UNIVERSITÄT
KOBLENZ · LANDAU
Fachbereich 4: Informatik

Entwicklung einer echtzeitfähigen Videokompression auf der GPU

Bachelorarbeit

zur Erlangung des Grades eines Bachelor of Science (B.Sc.)
im Studiengang Computervisualistik

vorgelegt von
Sebastian Karst

Erstgutachter:	Prof. Dr.-Ing. Stefan Müller (Institut für Computervisualistik, AG Computergraphik)
Zweitgutachter:	Gerrit Lochmann Deutsches Institut für Normung, Abteilung Moderne Technologien

Koblenz, im Mai 2015

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☐	☐
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	☐	☐

.....
(Ort, Datum) (Unterschrift)

Institut für Computervisualistik
AG Computergraphik
Prof. Dr. Stefan Müller
Postfach 20 16 02
56 016 Koblenz
Tel.: 0261-287-2727
Fax: 0261-287-2735
E-Mail: stefanm@uni-koblenz.de

UNIVERSITÄT
KOBLENZ · LANDAU

Fachbereich 4: Informatik

Aufgabenstellung für die Bachelorarbeit
Sebastian Karst
(Mat. Nr. 210 100 178)

Thema: Entwicklung einer echtzeitfähigen Videokompression auf der GPU

Es gibt heute verschiedene Ansätze, welche sich dem Problem widmen, um grafisch anspruchsvolle Applikationen auch auf Smartphones oder ähnlichen Endgeräten mit geringer Leistung darstellen zu können. Dabei spielt die Videokompression eine wichtige Rolle, um einen Hauptteil der Rechenlast von mobilen Endgeräten auf Computer oder Server auslagern zu können.

Die Live Übertragung von Videos stellt aufgrund der Ausgangsgröße des Bildmaterials eine große Herausforderung dar. Zudem bestehen besondere Anforderungen an den Kompressionsalgorithmus: Erstens muss das Verfahren performant und echtzeitfähig sein, zweitens dürfen, um die Latenz zwischen Enkodierung und Dekodierung möglichst gering zu halten, nur das aktuelle und bereits vergangene Frames zur Kompression verwendet werden.

Das Aufstellen eines Frameworks zum Testen verschiedener Kompressionstechniken soll als Ansatz für diese Arbeit dienen. Ziel der Arbeit ist es, die verschiedenen Arten der Videokompression zu evaluieren und sie auf ihre Praktikabilität hinsichtlich ihrer Nutzung auf der GPU zu überprüfen. Des Weiteren soll im Rahmen der Arbeit geklärt werden, ob es möglich ist Informationen aus dem Bildsynthesevorgang zu nutzen, um noch effizienter computergenerierte/synthetische Bilder zu komprimieren.

Die inhaltlichen Schwerpunkte der Arbeit sind:

1. Recherche sowie Einarbeitung in die Videokompression und ihre gängigsten Verfahren
2. Erstellen eines geeigneten Testframeworks
3. Implementation verschiedener Videokompressionsverfahren
4. Evaluation der einzelnen Algorithmen und Gegenüberstellung mit vorhandenen Lösungen
5. Dokumentation der Ergebnisse

Koblenz, 02.10.2014

– Sebastian Karst –

– Prof. Dr. Stefan Müller –

Inhaltsverzeichnis

Abbildungsverzeichnis

Zusammenfassung

Das Thema dieser Arbeit ist die Entwicklung einer hardwarebeschleunigten Einzelbildkompression zur Videoübertragung. Verfahren zur Einzelbildkompression existieren bereits seit längerer Zeit. Jedoch genügen die gängigen Verfahren nicht den Anforderungen der Echtzeit und Performanz, um während einer Videoübertragung ohne spürbare Latenz zum Einsatz zu kommen. In dieser Arbeit soll einer der geläufigsten Algorithmen zur Bildkompression auf Parallelisierbarkeit, unter zu Hilfenahme der Grafikkarte, untersucht werden, um Echtzeitfähigkeit während der Kompression und Dekompression von computergenerierten Bildern zu erreichen. Die Ergebnisse werden evaluiert und in den Rahmen aktueller Verfahren parallelisierter Kompressionstechniken eingeordnet.

Abstract

The subject of this thesis is the developement of a hardware-accelerated image compression technique with the ability for video streams in real-time. Image compression techniques have existed for some time now. However they do not meet the real-time and performance requirements needed for video streaming whithout a noticeable latency. One of the most common algorithms for image compression will be analysed with regard to parallelization capability with the aid of the graphics card to accomplish real-time ability during the compression and decompression process. The results are evaluated and compared to state-of-the-art compression techniques.

1 Einleitung

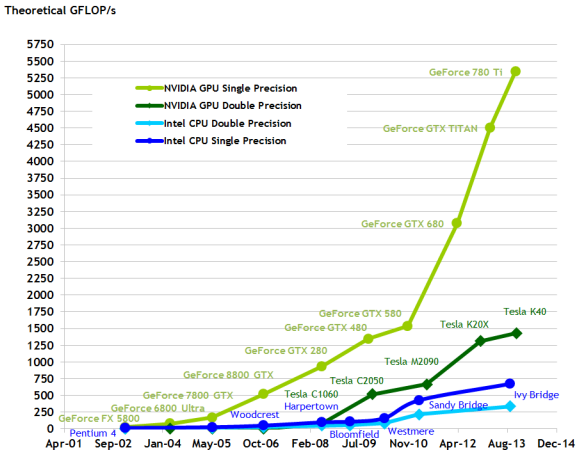
Die rasante Entwicklung der Hardware im Bereich der Grafikkarten ermöglicht es, Arbeiten zu erledigen, bei denen ein hohes Maß an Parallelität gefordert ist.
Noch vor wenigen Jahren war es üblich Daten nur in eine Richtung weiterzuleiten - von der CPU zur GPU, welche dann wiederum Bilder generierte, um sie auf einem Ausgabemedium darzustellen. Diese Anforderung resultierte schließlich in einer enormen Steigerung der parallel arbeitenden Prozessoren auf den Grafikkarten. Da sich die Arbeit der Grafikkarte dabei meist auf einfache arithmetische Operationen reduzieren lässt, die sehr häufig (d.h. pro Pixel oder Vertex) angewendet werden, liegt eine parallele Architektur der Prozessoren nahe.
Es gibt jedoch noch zahlreiche Problemstellungen außerhalb der Computergrafik, die ebenfalls eine hohe Anzahl von parallel arbeitenden Prozessen voraussetzen. Unter Zuhilfenahme der so genannten "General Purpose Graphics Processing Units"können große Datenmengen parallel verarbeitet werden und die Grafikkarte kann dadurch als stark parallelisierte Processing Unit, neben dem Hauptprozessor genutzt werden.

Im Bereich der Bild oder Videokompression finden zahlreiche Verfahren Anwendung, jedoch selten mit der Fähigkeit in Echtzeit komprimieren bzw. dekomprimieren zu können. Durch Parallelisierung der Verfahren und die Nutzung der Grafikkarte als Processing Unit kann die Performanz deutlich gesteigert werden - Echtzeitfähigkeit ist somit möglich.
Diese Arbeit soll die Möglichkeiten der Parallelisierung erarbeiten und mit Hilfe von OpenGL Compute Shadern eine parallelisierte Version des Kompressions- und Dekompressionsalgorithmus implementiert werden.

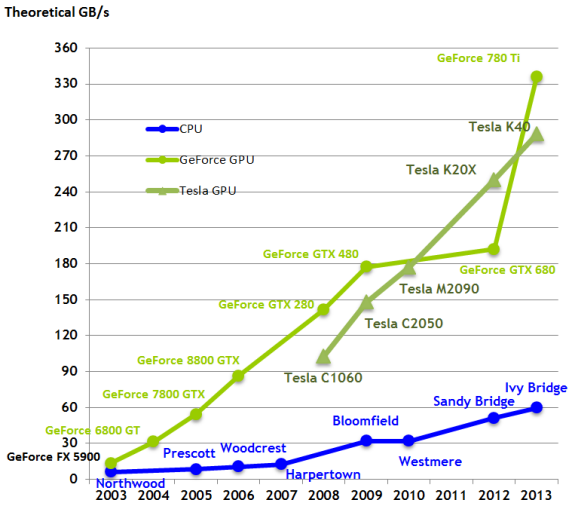
2 Grundlagen

2.1 General Purpose GPU und OpenGL Compute Shader

Der Begriff „General Purpose Graphics Processing Unit“ (kurz: GPGPU) bezeichnet die Verwendung der Grafikkarte um „generelle“ Berechnungen durchzuführen, also solche, die nicht in den klassischen Arbeitsbereich der Computergrafik fallen.
Gängige Einsatzgebiete sind physikalische Simulationen, Segmentation in 2D und 3D oder das Machine Learning, also Gebiete, in denen viele arithmetische Operationen in kurzer Zeit auf großen Datenmengen erfolgen, um größtmöglichen Nutzen aus der parallelen Datenverarbeitung zu ziehen[NVI14]. Wie groß dieser Nutzen theoretisch gegenüber sequentieller Verarbeitung auf einer CPU ausfallen kann, hängt stark von der Architektur der Grafikkarte ab und ist in Abbildung 1(a) verdeutlicht.



(a)



(b)

Abbildung 1: 1(a):Vergleich der theoretischen GFlop/s von GPU und CPU von 2001 bis 2014
1(b):Vergleich der theoretischen Gigabyte/s von GPU und CPU von 2001 bis 2013

http://docs.nvidia.com/cuda/cuda-c-programming-guide/graphics/floating-point-operations-per-second.png

http://docs.nvidia.com/cuda/cuda-c-programming-guide/graphics/memory-bandwidth.png

Es ist auf den Grafiken zu sehen, dass heute, unter optimalen Bedingungen, mehr als zehn mal so viele Floating Point Operationen pro Sekunde auf der GPU durchgeführt werden können, als auf einer vergleichsweise aktuellen CPU, auch wenn die Bedingungen selten so optimal sind, dass diese Ergebnisse erzielt werden können. Ebenso verhält es sich mit der Speicherbandbreite, sie ist auf heutigen GPUs immer noch fünf mal höher als auf

vergleichbar aktuellen CPUs. Durch diese Vorzüge sind aber komplexe Cachingverfahren auf der GPU nicht möglich, sodass die Programme ohne viele Kontrollmechanismen auskommen müssen.
Um GPGPU-Programme zu erstellen, stehen dem Programmierer mehrere Bibliotheken zur Verfügung. Die wichtigsten sind:

- CUDA, von Nvidia entwickelt, bietet die vermutlich bekannteste und bestdokumentierte API. Allerdings wird eine Grafikkarte von NVIDIA vorrausgesetzt.
- OpenCL, welches von der Khronos Group entwickelt wird, besitzt die Möglichkeit mit jeder Art von GPU-Hardware zu operieren.
- Direct Compute, als Teil von Microsoft's DirectX Sammlung entwickelt, findet diese API in der Spieleentwicklung Anwendung, ist jedoch nicht ausreichend dokumentiert.
- OpenGL's Compute Shader, seit der Version 4.3 gehören sie zu den Standardfunktionen von OpenGL und stellen das Pendant zu Microsoft's Direct Compute Shadern dar.

Da sich die Terminologie der Bibliotheken teils stark unterscheiden, werden für den Rest dieser Arbeit die Bezeichnungen der OpenGL Compute Shader übernommen, um Einheitlichkeit zu garantieren.
Ein Compute Shader unterscheidet sich nur in wenigen Aspekten von anderen Shadern wie Fragment- oder Vertex-Shader. Sie werden in der gleichen Sprache GLSL (OpenGL Shading Language) verfasst, mit den selben Befehlen erstellt und gebunden, was es für OpenGL Programmierer einsteigerfreundlich macht. Jedoch unterliegen sie nicht den selben Einschränkungen, nur an einer bestimmten Position der Renderpipeline operieren zu können[Gro15]. Tatsächlich liegt hier die eigentliche Stärke von Compute Shadern, da sie jederzeit Ausgabedaten einer bestimmten Shaderstage als Eingabe erhalten können, um Ihre Berechnungen als Eingabe an folgenden Shaderstages weiterzuleiten.

Bei der Ausführung von Compute Shadern muss der Berechnungsraum an die Aus- oder Eingabe des Compute Shaders angepasst werden. Die Ausführung der Compute Shader wird in Gruppen zusammengefasst. Eine solche Gruppe wird Work Group genannt. Eine einzelne Ausführung eines Compute Shaders wird Invocation genannt. Invocations werden in sog. local Work Groups zusammengefasst. Eine Local Workgroup beschreibt gleichzeitig die minimale Anzahl von parallel ausgeführten Invocations. Außerdem werden alle Local Work Groups als Global Workgroup zusammengefasst[Shr13].

Lokale und Globale Work Groups haben, wie in Abbildung 2 zu sehen, einen dreidimensionalen Aufbau. Die Dimensionen der Local Work Group werden

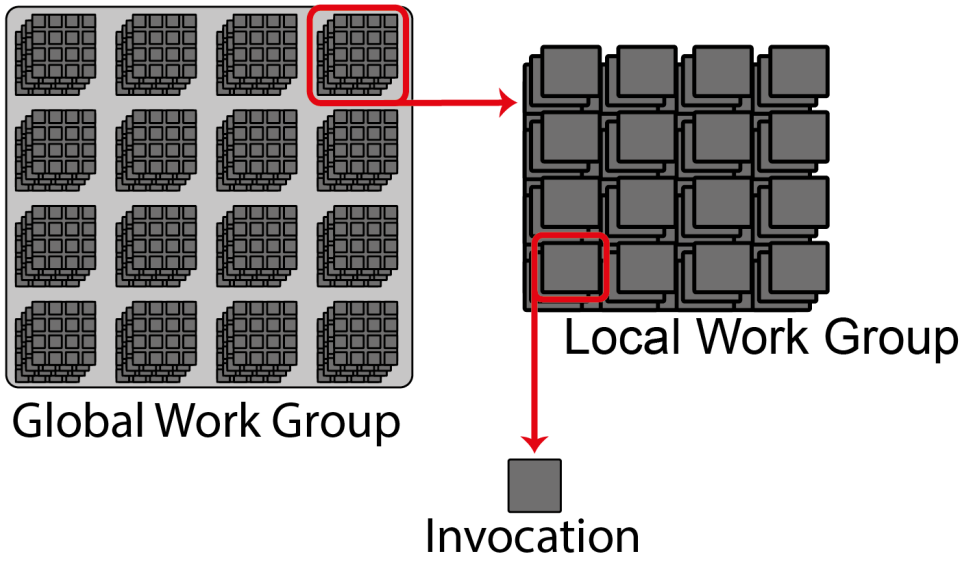


Abbildung 2: Dreidimensionale Anordnung von Work Groups

im Compute Shader selbst mit Built-in Variablen deklariert. Die Dimensionen der Global Work Group werden im OpenGL Code zur Laufzeit deklariert. Es ist also möglich Folgendes zu definieren, um eine zweidimensionale Work Group zu erstellen:

```
1 layout (local_size_x = 8, local_size_y = 8, local_size_z =
      1) in;
```

Durch diese Vorgehensweise eignen sich Compute Shader, um Bildverarbeitung zu realisieren, welche meist auf zweidimensionalen Datensätzen wie Texturen bzw. Bildern arbeiten. Dieser Ansatz dient als grundlegender Baustein für die Bildkompression in den folgenden Kapiteln und wird in Kapitel 3 näher beschrieben. Während die einzelnen Invocations einer Local Work Group gleichzeitig ausgeführt werden, kann man keine Vorhersage machen, in welcher Reihenfolge die Local Work Groups selbst ausgeführt werden[Gro15]. Um zu garantieren, dass alle Work Groups ihre Arbeit beendet haben, muss der Programmierer selbst dafür Sorge tragen.

2.1.1 Limitierungen und Grenzen von Parallelisierbarkeit

Auch wenn die Ausführungen in Kapitel 2.1 darauf schließen lassen, dass Programme nur auf genügend Work Groups verteilt werden müssen, um beliebig beschleunigt zu werden, stößt jede Strategie zur Parallelisierung zwangsläufig an gewisse Grenzen. Die Grenzen der Hardware, in diesem Fall eher die begrenzte Anzahl von Prozessoren, stellt eine dieser Grenzen dar, welche durch die Nutzung der Grafikkarte mit ihren vielen tausend Prozessoren irrelevant werden soll.

Eine andere Grenze wird durch das Amdahlsche Gesetz formuliert. Es sagt aus, dass ein Algorithmus immer einen sequenziellen und einen parallelisierbaren Teil besitzt. Der parallelisierbare Teil des Programms kann theoretisch beliebig beschleunigt werden, wenn er nur auf genügend Prozessoren verteilt wird. Der sequentielle Teil dagegen stellt die Grenze der Parallelisierbarkeit dar[HM08]. Mathematisch wird der maximale Berechnungszeitgewinn in Amdahls Gesetz folgendermaßen definiert:

$$(1 - P) + \frac{P}{N} \tag{1}$$

Die Berechnungszeit des Algorithmus ohne Parallelisierung wird auf eins normiert. $(1 - P)$ ist dabei der nicht parallelisierbare Anteil des Algorithmus. P ist der parallelisierbare Anteil des Algorithmus und $\frac{P}{N}$ ist der parallelisierbare Anteil auf N Prozessoren verteilt. Wenn man nun sehr viele Prozessoren nutzt, wird der Term $\frac{P}{N}$ vernachlässigbar klein, aber der sequenzielle Teil $(1 - P)$ bleibt bestehen.

2.2 Bildkompression

Die Kompression von digitalen Bildern unterscheidet sich kaum von der Kompression anderer Daten, wie Audiodateien, Textdokumenten oder Quellcodes. Das Ziel einer jeden Datenkompression ist es, Informationen so kompakt wie möglich zu repräsentieren, um Speicherplatz zu sparen oder die Daten schneller übertragen zu können[Nel]. Dabei muss darauf geachtet werden, dass die Ursprungsdaten wieder aus der komprimierten Repräsentation rekonstruiert werden können. Gerade bei der Datenkompression zur Übertragung ist dies besonders wichtig. Insbesondere wenn die komprimierten Informationen versendet werden, unterscheidet man die Datenverarbeitung auf der Sender- und Empfängerseite. Die Verarbeitung auf Senderseite wird Encodierung oder Encoder genannt. Auf der Empfängerseite wird das Verfahren Decodierung oder Decoding genannt. Beides zusammen wird häufig als Codec bezeichnet[Str].

Bei der Bildkompression ist das vornehmliche Ziel ebenfalls die reine Anzahl an Bits zu reduzieren, die benötigt werden, um das Bild abzuspeichern, jedoch unter der Prämisse, dass die subjektive Qualität nicht zu sehr darunter leidet[Say]. Um dies bewerkstelligen zu können, haben sich einige Algorithmen und Verfahren etabliert, die besonders für Bilddaten überzeugende Ergebnisse liefern. In diesem Kapitel werden sie vorgestellt und erläutert, warum sich diese Verfahren für die Bilddatenkompression eignen.

2.2.1 Gängige Verfahren

Es gibt heute unzählige Vorgehensweisen um diese Ziele erreichen zu können. Einige Verfahren zielen darauf ab, Redundanzen zu eliminieren. Sie wer-

den dann verlustfreie Verfahren (engl.: *lossless compression*) genannt[Nel]. Andere wiederum versuchen irrelevante Informationen zu entfernen. Sie werden verlustbehaftete Verfahren (engl.: *lossy compression*) genannt[Nel]. Diese Unterscheidung lässt sich auf fast alle Arten der Datenkompression anwenden.

2.2.1.1 Verlustfreie Verfahren

Bei den verlustfreien Verfahren handelt es sich um Algorithmen, welche garantiert nach erfolgreicher Encodierung und Decodierung ein exaktes Duplikat der Ausgangsdaten erzeugen. Gegeben sei ein Encoder, welcher einen Datensatz $\boldsymbol{X}$ als Eingabe erhält und aus diesem eine komprimierte Repräsentation der Daten $\boldsymbol{X}_C$ generiert. Außerdem erstellt ein Decoder aus $\boldsymbol{X}_C$ den rekonstruierten Datensatz $\boldsymbol{X}_R$.

$$\begin{aligned} \boldsymbol{X} \;&\rightarrow\; \boldsymbol{X}_C \;\rightarrow\; \boldsymbol{X}_R \\ \text{mit:}\;\; \boldsymbol{X} \;&=\; \boldsymbol{X}_R \end{aligned} \tag{2}$$

Verlustfreie Kompression findet hauptsächlich dort Anwendung, wo kein Datenverlust hinnehmbar wäre, wie z.B. bei Textdateien, Quellcodes oder beim Packen in eine Zip-Datei[Say]. Gerade bei letzterem wäre ein Datenverlust von wenigen Bits bereits kritisch, da es beim Entpacken zu Fehlern führen würde, welche nicht vom entsprechenden Algorithmus kompensiert werden könnten. Das Resultat wäre eine fehlerhaft entpackte Datei, welche nicht mehr verwendet werden könnte.

2.2.1.2 Verlustbehaftete Verfahren

Kompressionsverfahren die einen Informationsverlust in Kauf nehmen, werden als verlustbehaftet bezeichnet. Sie erreichen häufig deutlich höhere Kompressionsraten als verlustlose Verfahren. Daten, die mit dieser Technik komprimiert wurden, lassen sich nicht vollständig rekonstruieren, sodass folgendes gilt:

$$\begin{aligned} \boldsymbol{X} \;&\rightarrow\; \boldsymbol{X}_C \;\rightarrow\; \boldsymbol{X}_R \\ \text{mit:}\;\; \boldsymbol{X} \;&\neq\; \boldsymbol{X}_R \end{aligned} \tag{3}$$

Für viele Anwendungen spielt der Informationsverlust eine untergeordnete Rolle. Gerade wenn es sich um Daten handelt, die von Menschen interpretiert werden, wie bspw. Bild oder Ton[Str].
Entscheidend für die Qualität der rekonstruierten Daten ist hier maßgeblich die Effizienz, mit welcher Informationen als „irrelevant“ erkannt werden. Sollten die falschen Informationen als irrelevant erkannt werden, kann es sein, dass die Kompressionsrate hoch, aber der subjektive Qualitätsverlust nicht mehr hinnehmbar wäre.

Welche Informationen relevant sind und welche nicht, lässt sich anhand der Daten selbst nur schwer erkennen. Doch Erkenntnisse über die menschliche Wahrnehmung können Aufschluss darüber geben, wie man verlustbehaftet visuelle Medien wie Videos oder Bilder komprimieren kann.

2.2.2 Ausnutzen von psychovisuellen menschlichen Schwächen

Um bei einer verlustbehafteten Kompressionstechnik zu entscheiden, welche Bildinformationen von Relevanz sind, hilft es die Physiologie des menschlichen Auges zu untersuchen, denn die visuelle Wahrnehmung des Menschen wird durch den Aufbau des Auges maßgeblich bestimmt.

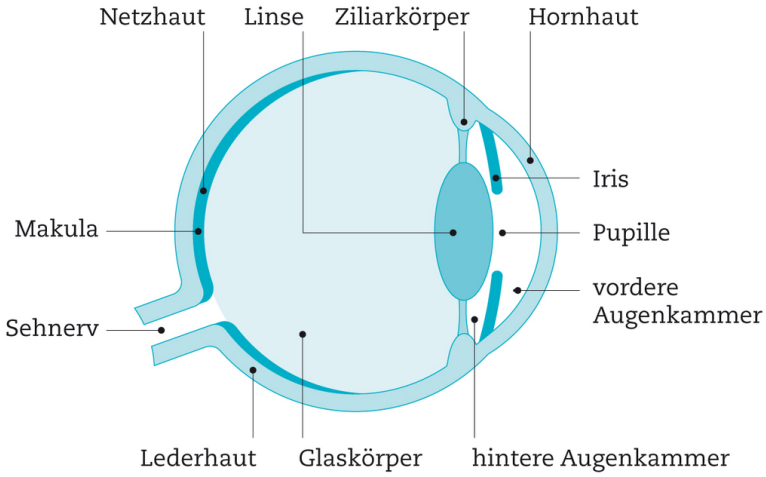


Abbildung 3: Schematischer Aufbau eines Auges

http://www.amvz-lausitz.de/media/intemplate/27_Auge_Anatomie.jpg

In Abbildung 3 ist ein Auge schematisch dargestellt. Das einfallende Licht fällt durch die Pupille auf die Netzhaut. Auf der Netzhaut sitzen Rezeptoren, die das Licht in neuronale Signale übersetzen. Diese Signale werden dann über den Sehnerv ins Gehirn übertragen, wo letztendlich das Bild entsteht.
Die Rezeptoren auf der Netzhaut unterscheiden sich in zwei Typen. Die sog. Stäbchen sind besonders für die Wahrnehmung der Lichtintensität zuständig. Sie sind jedoch unfähig Farben zu unterscheiden und sind besonders aktiv, wenn wenig Licht ins Auge fällt, z.B. bei Nacht.
Zapfen nennt man die Rezeptoren, die uns die Farbwahrnehmung ermöglichen. Sie unterscheiden Farben und sind im Stande Helligkeit wahrzunehmen, jedoch reagieren sie empfindlicher auf Farben. Im Auge befinden sich ca. 20-mal so viele Stäbchen wie Zapfen (6 Millionen Zapfen, 120 Millionen Stäbchen)[Mal]. Dieser Umstand resultiert in einer deutlich geringeren Auflösung beim Farbsehen im Gegensatz zum Helligkeitssehen, sodass Farbkanten deutlich schlechter wahrgenommen werden. In einer Bildkompression kann dies ausgenutzt werden. Voraussetzung ist, dass die Bilddaten in einem

Farbraum vorliegen, der die Helligkeit von der Farbgebung trennt.
Solch ein Farbraum ist bspw. der YCbCr-Farbraum, wobei Y die Helligkeitskomponente darstellt, Cb steht für *chrominance blue* und Cr für *chrominance red*. Im Gegensatz zum gebräuchlichen RGB-Farbmodell, bei welchem die Helligkeitskomponente nur durch eine Kombination aus Rot-, Blau- und Grünkanal, errechnet wird, existiert im YCbCr-Modell eine Dekorrelation der Farbkomponenten. Die Chrominanz wird dabei in zwei Kanälen gesondert gespeichert. Der Cb-Kanal interpoliert dabei zwischen blau und gelb. Der Cr-Kanal interpoliert zwischen rot und grün. Abbildung 4 zeigt ein Bild im RGB-Farbraum, und nachfolgend die einzelnen Kanäle des gleichen Bildes im YCbCr-Raum.

Abbildung 4: Oben links: Bild in RGB ; Oben rechts: Helligkeitskanal (Y)
Unten links: Farbkanal blau (Cb); Unten rechts: Farbkanal rot (Cr)

Wenn die Auflösung der beiden Farbkanäle Cb und Cr horizontal oder vertikal halbiert wird, stellt das für die meisten Menschen keinen sichtbaren

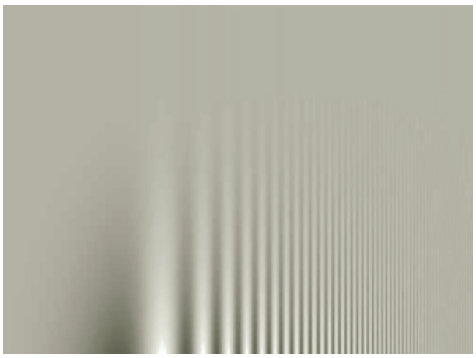
Unterschied dar. Abbildung 5 zeigt links ein Bild ohne Unterabtastung des Farbraumes und rechts das gleiche Bild mit vertikal halbierter Auflösung im Cb- und im Cr-Kanal.

Abbildung 5: Links: Bild in YCbCr-Farbraum ohne Unterabtastung der Farbkanäle (4:4:4)
Rechts: Bild in YCbCr-Farbraum mit Unterabtastung der Farbkanäle (4:2:2)

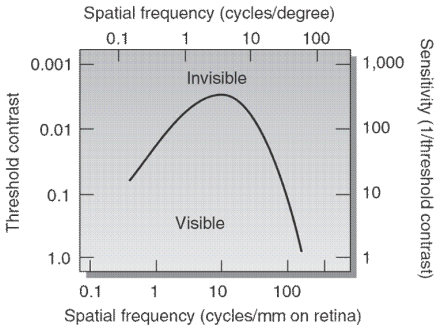
Dieses Verfahren wird Farbunterabtastung oder engl. Chroma subsampling genannt. Durch diese Technik wurde der Speicherbedarf für zwei von drei Kanälen bereits auf die Hälfte reduziert. Die verlorengegangenen Informationen sind nicht rekonstruierbar, was die Vorgehensweise verlustbehaftet macht. Jedoch ist der Informationsverlust nicht vom Menschen detektierbar[Str]. Anders sähe es aus, wenn man dieses Verfahren auf den Helligkeitskanal anwenden würde. Ein Mensch würde den Qualitätsverlust sofort bemerken.

Das bedeutet nicht, dass der Helligkeitskanal nicht komprimiert werden sollte. Es kommt viel mehr darauf an, die Kompressionstechnik der Helligkeitsinformation an die Wahrnehmung des Menschen anzupassen[WBP].
Wie viele Details ein Mensch beim Sehen wahrnimmt, wird maßgeblich durch zwei Größen bestimmt - den Kontrast und die Ortsfrequenz[Str].
Kontrast bedeutet in diesem Zusammenhang, die Helligkeitsänderung zu benachbarten Flächen, und die Ortsfrequenz beschreibt die Anzahl der Linien eines Rechteckgitters oder die Anzahl der Perioden eines Sinusgitters.
Der Unterschied der beiden Größen wird deutlich, wenn man sich ein Bild mit abwechselnd hellen und dunklen, vertikalen Linien vorstellt. Die Frequenz beschreibt dabei die Anzahl der sichtbaren Linien bei einem fest definierten Blickwinkel. Der Kontrast beschreibt die Helligkeitsänderung von einer Linie zur nächsten.

Wenn nun der Abstand vom Betrachter zum Bild vergrößert wird, verändert sich der Kontrast nicht, da die Helligkeit der Linien unverändert bleibt. Die Frequenz wird aber bei gleichbleibendem Blickwinkel erhöht, da mehr Linien in den Blickwinkel fallen.
Resultat des Experimentes ist, dass die optische Unterscheidung von einer hellen zu einer dunklen Linie erschwert wird. Erhöht man die Frequenz bzw. den Abstand noch weiter, wird es zunehmend schwieriger die Linien voneinander zu unterscheiden, bis zu dem Moment, wo der Betrachter keine einzelnen Linien mehr ausmachen kann und das für ihn sichtbare Bild einen einheitlichen Grauwert zu haben scheint[Mal].
Abbildung 6(a) visualisiert von links nach rechts ansteigende Ortsfrequenzen und von oben nach unten steigende Kontrastwerte.



(a) Abhängigkeit der Erkennbarkeit von Kontrast und Frequenz



(b) Erkennbarkeit bei verschiedenen Ortsfrequenzen

Abbildung 6: 6(a): Kontrast nimmt von unten nach oben zu, Frequenz steigt von links nach rechts
6(b): Unterhalb der Linie ist der sichtbare Bereich, darüber nicht sichtbar

http://www.cns.nyu.edu/~david/courses/perception/lecturenotes/channels/csf.gif

http://www.cns.nyu.edu/~david/courses/perception/lecturenotes/channels/sweep.gif

Setzt man den Kontrast in Abhängigkeit zur Ortsfrequenz, lässt sich eine Funktion definieren, die Modulationsübertragungsfunktion genannt wird. Sie beschreibt die Sichtbarkeit von Kontrasten in Korrelation zur Periodenanzahl eines Sinusgitters. Untersuchungen in der Wahrnehmungspsychologie haben ergeben, dass Kontraste bei niedrigen Ortsfrequenzen (in Abbildung 6(a) links) besser erkennbar sind, als bei hohen (in Abbildung 6(a) rechts), also dass niedrige Frequenzen detailliertes Sehen begünstigen. Diese Abhängigkeit ist in Abbildung 6(b) dargestellt.
Diese Informationen können ausgenutzt werden um für das menschliche Auge schlecht zu erkennende Frequenzen zu eliminieren. Das Problem dabei ist, dass die Helligkeitsinformationen im Helligkeitskanal nicht im Frequenzbereich vorliegen, sondern im Ortsbereich. Bevor für das menschliche Auge

irrelevante Frequenzen entfernt werden können, muss also erst eine Transformation der Helligkeitswerte in den Frequenzbereich vorgenommen werden. Im Bereich der Bildverarbeitung haben sich für diesen Zweck besonders die diskreten Transformationen etabliert, da sie eine Dekorrelation der Frequenzen garantieren. Am Beispiel der diskreten Kosinustransformation soll diese Transformation und die nachfolgende Elimination der irrelevanten Frequenzen im Kapitel 2.2.3 erklärt werden.

2.2.3 Die JPEG Pipeline

Die Joint Photographic Experts Group, kurz JPEG, definierte bereits 1992 ein grundlegendes und standardisiertes Verfahren zur Kompression von Einzelbildern, den sog. Baseline Algorithm. Das standardisierte JPEG Verfahren unterstützt die verlustbehaftete Kompression von Einzelbildern mit einer Genauigkeit von acht Bit pro Bildpunkt in jedem Kanal. Abbildung 7 zeigt den schematischen Ablauf einer JPEG-Kompression.

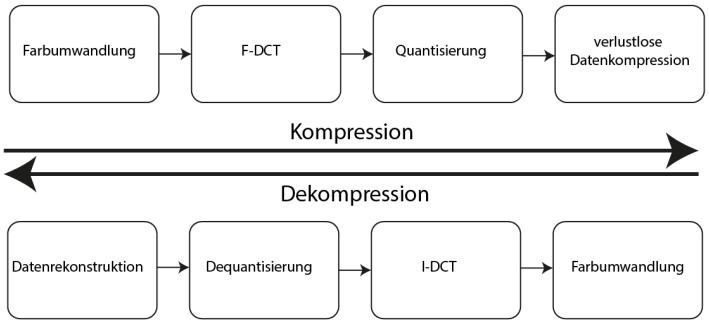


Abbildung 7: Reihenfolge der Operationen während der Kompression und Dekompression

In Abbildung 7 steht F-DCT für die Diskrete Kosinustransformation (engl.: forward cosine transformation) und I-DCT ihre Rücktransformation (engl.: inverse cosine transformation).

Der RGB-Farbraum ist zwar heute der bekannteste und weitverbreitetste Farbraum, eignet sich aber, wie in Kapitel 2.2.2 beschrieben, nicht für die Bildkompression, da man die Besonderheiten der Farb- und Helligkeitswahrnehmung nicht in die Datenverarbeitung mit einbeziehen kann. Also müssen die Pixeldaten zuerst in einen Farbraum übersetzt werden, der einen dedizierten Helligkeitskanal besitzt.
Deswegen beginnt der Algorithmus mit der Umwandlung der Bilddaten von RGB in den YCbCr-Farbraum [WBP]. Diese Transformation lautet:

$$\begin{pmatrix} Y \\ Cb \\ Cr \end{pmatrix} = \begin{pmatrix} 0.2990 & 0.5870 & 0.1140 \\ -0.1687 & -0.3313 & 0.5000 \\ 0.5000 & -0.4187 & -0.0813 \end{pmatrix} \cdot \begin{pmatrix} R \\ G \\ B \end{pmatrix} \tag{4}$$

Die Transformation ist bis auf vernachlässigbare Rundungsfehler verlustlos, da die Rücktransformation durch die Inverse zur Transformationsmatrix in Gleichung 4 verwirklicht wird:

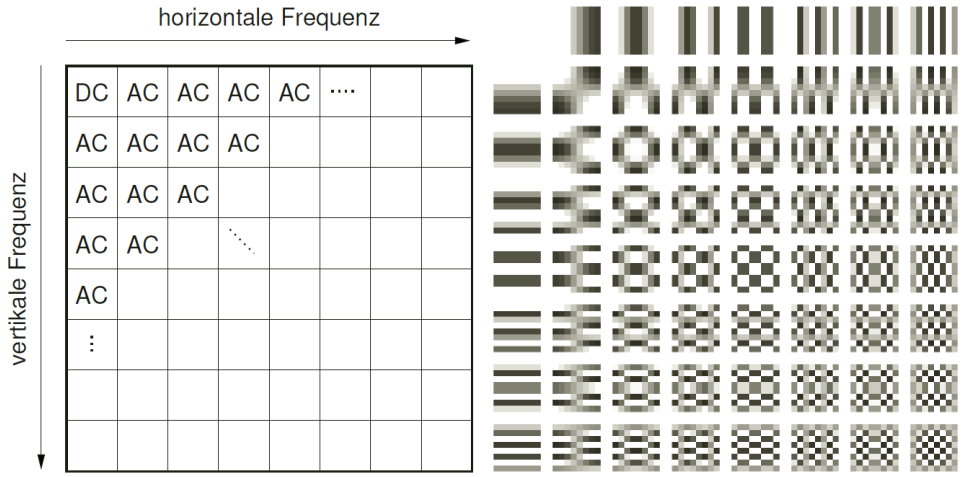
$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} 1.0 & 0.0 & 1.4020 \\ 1.0 & -0.3441 & -0.7141 \\ 1.0 & 1.7720 & -0.0001 \end{pmatrix} \cdot \begin{pmatrix} Y \\ Cb \\ Cr \end{pmatrix} \tag{5}$$

An dieser Stelle des Algorithmus liegen die Farbwerte in separaten Kanälen und es gibt einen dedizierten Helligkeitskanal. Es ist also jetzt möglich, das in Kapitel 2.2.2 beschriebene Chroma Subsampling durchzuführen, sodass der Ortsbereich der Farbkanäle auf die Hälfte reduziert wird.
Nun beginnt das eigentliche Herzstück des JPEG Kompressionsvorgangs - die diskrete Kosinustransformation. Diese ist definiert durch:

$$D(i,j) = \frac{1}{\sqrt{2N}} C(i)\, C(j) \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} p(x,y) \cos\left[\frac{(2x+1)\, i\pi}{2N}\right] \cos\left[\frac{(2y+1)\, j\pi}{2N}\right]$$

$$C(u) = \begin{cases} \frac{1}{\sqrt{2}} \text{ if } u = 0 \\ 1 \text{ if } u > 0 \end{cases} \tag{6}$$

Wobei p(x,y) der Pixelwert ist und N die Blockgröße auf dem die DCT operiert. In JPEG wird die DCT typischerweise mit einer Blockgröße von acht ausgeführt[WBP]. Wenn die Blockgröße mit acht angegeben ist, werden 8x8 Grauwerte in 8x8 DCT-Koeffizienten transformiert. Die 64 daraus resultierenden Koeffizienten lassen sich in einen Gleichanteil, der die durchschnittliche Helligkeit des Blockes definiert (in Abbildung 8(a) mit DC dargestellt) und 63 Wechselanteile (in Abbildung 8(a) mit AC dargestellt), welche die vertikalen und horizontalen Frequenzen repräsentieren, unterscheiden. Abbildung 8(a) zeigt den schematischen Aufbau der Ausgabe der Transformation eines 8x8 Grauwertblocks.



(a) DC ist der Gleichanteil, AC der Wechselanteil

(b) Basisbilder einer 8x8 DCT

Abbildung 8: In 8(a) steigen die Frequenzen von links oben nach rechts unten an[Str];
In 8(b) zeigen weiße Pixel einen hohen und dunkle Pixel einen niedrigen Wert an

https://upload.wikimedia.org/wikipedia/commons/2/24/DCT-8x8.png

Die Koeffizienten in 8(a) repräsentieren eine Gewichtung der in 8(b) dargestellten Basisbilder/Frequenzen. Ein niedriger Koeffizient bedeutet also, dass die zugehörige Frequenz im Ursprungsbild kaum präsent war. Ein hoher Wert steht für eine sehr präsente Frequenz im Ursprungsbild. Zusammenfassend lässt sich sagen: Hohe Koeffizienten sind für die Rekonstruktion wichtig, niedrige Koeffizienten sind für die Rekonstruktion irrelevant[Nel]. Die Rekonstruktion der Grauwerte erfolgt später durch die Überlagerung der Basisbilder (eigentlich eine Überlagerung der Frequenzen), gewichtet durch ihre zugehörigen Werte im Koeffizientenblock. Die Rücktransformation ist definiert durch:

$$p(x,y) = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} \frac{C(i)\,C(j)}{\sqrt{2N}} D(i,j) \cos\left[\frac{(2x+1)\,i\pi}{2N}\right] \cos\left[\frac{(2y+1)\,j\pi}{2N}\right] \tag{7}$$

Durch die Hintransformation liegen die Bildwerte nun im Frequenzbereich vor. Das ermöglicht nun die Elimination irrelevanter Frequenzen, wie in Kapitel 2.2.2 beschrieben. Dies geschieht im Falle der JPEG-Kompression durch Quantisierung der DCT-Koeffizienten. Ziel der Quantisierung ist es, den Wertebereich der Koeffizienten zu verkleinern und niedrige Koeffizienten auf null zu setzen, um im späteren Codierungsverfahren Redundanzen auszunutzen [Str]. Hier geschieht die eigentliche verlustbehaftete Datenreduktion; verlustbehaftet, weil die auf Null gerundeten Werte nicht mehr korrekt rekonstruiert

werden können. Die Rekonstruktion des ursprünglichen Grauwertblock ist also nur eine Approximation durch die Frequenzen, die nach der Quantisierung übrig bleiben. Die Quantisierung ist definiert durch:

$$Q(i,j) = \lfloor \frac{D(i,j)}{q(i,j)} + 0.5 \cdot \text{ sign } (D(i,j)) \rfloor \tag{8}$$

In Kapitel 2.2.2 wurde beschrieben, dass hohe Frequenzen für den Menschen schlecht wahrnehmbar sind und niedrige Frequenzen zum Detailreichtum der Wahrnehmung beitragen. Dieses Prinzip findet in der Quantisierungsmatrix des JPEG-Algorithmus Anwendung. Die Quantisierung wird nicht gleichmäßig auf alle Koeffizienten angewandt, sondern hohe Frequenzen werden stärker quantisiert als die für den Menschen wichtigen niedrigen Frequenzen[WBP]. Abbildung 9 zeigt die von der ITU-T empfohlenen Quantisierungstabelle für den Luminanzkanal:

16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

Abbildung 9: 8x8 Quantisierungsmatrix für Luminanz der ITU-T T.81

http://www.w3.org/Graphics/JPEG/itu-t81.pdf

Verschiedene Kompressionsraten erreicht man durch Skalierung der Quantisierungsmatrix. Skaliert man diese Tabelle um den Faktor zwei, werden bereits deutlich mehr Werte durch Rundungsfehler auf null gesetzt. Skaliert man jedoch um den Faktor 0,1 werden kaum noch Werte auf null gesetzt und es wird eine geringere Kompressionsrate erzielt. Nachdem die quantisierten Werte codiert wurden, lässt sich die Rekonstruktion durch folgende Dequantisierung erreichen:

$$D(i,j) = Q(i,j) \cdot q(i,j) \tag{9}$$

Durch die in Formel 8 durchgeführte Rundung, werden die quantisierten Werte in einer reduzierten Präzision gespeichert. Dies soll den Wertebereich zusätzlich verkleinern, um die Daten später mit weniger Bits abspeichern zu können. Die Rundungsfehler setzen sich so jedoch für den Rest des Kompressionsvorganges fort, sodass die subjektive Qualität der Bilder deutlich sinkt. Nach der Quantisierung werden die Koeffizienten umsortiert. Aus der zweidimensionalen 8x8 Repräsentation wird eine eindimensionale 1x64 Repräsentation erstellt. Dafür wird die quantisierte 8x8 Koeffizientenmatrix in einem

Zick-Zack-Muster abgearbeitet, welches in Abbildung 10 zu sehen ist. Der Zweck des sog. Zig-Zag-Scans ist es, die hohen Frequenzen, die mit einer erhöhten Wahrscheinlichkeit auf null gesetzt wurden möglichst redundant hintereinander zu schreiben, um diese Redundanz beim Kodieren ausnutzen zu können.

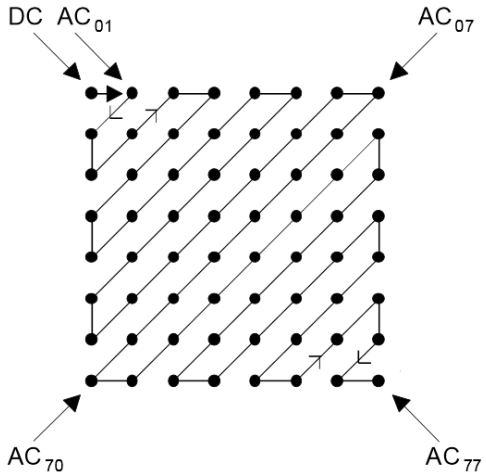


Abbildung 10: Reihenfolge der Koeffizientenumsortierung

http://www.w3.org/Graphics/JPEG/itu-t81.pdf

Die niedrigen Frequenzen stehen nun am Anfang und die hohen am Ende. Nach der Umsortierung werden die quantisierten Werte verlustlos komprimiert. Hierfür eignen sich eigentlich alle Arten der verlustlosen Kompression. JPEG bietet allerdings die Kodierung durch den Huffman-Algorithmus, arithmetisches Codieren und die Darstellung als sog. variable-length-integers. Um den Rahmen dieser Arbeit nicht zu sprengen, soll nicht weiter auf die verschiedenen Formen der Kodierung eingegangen werden. Es wird im Kapitel 3.1.6 eine Lauflängenkodierung vorgestellt, die in der Implementation benutzt wurde, um die quantisierten Werte verlustlos zu komprimieren.

3 Entwurf & Implementation

Die Kompressionstechnik, die im Rahmen dieser Arbeit entwickelt wurde, orientiert sich stark an dem, in Kapitel 2.2.3 beschriebenen, Baseline Algorithmus des JPEG-Standards. Die Gemeinsamkeiten und etwaige Abweichungen des entwickelten Programms zur JPEG-Kompression sollen beschrieben werden. Die einzelnen Schritte der Kompression werden in eigenen Kapiteln behandelt, um hervorzuheben, dass die Bildkompression verschiedene Techniken (verlustlose, sowie verlustbehaftete) miteinander kombiniert, um eine möglichst hohe Kompressionsrate bei gleichzeitig hoher subjektiver Bildqualität zu erreichen.

Die einzelnen Kapitel 3.1.1 bis 3.1.6 behandeln die Schritte der Bildkompression in der Reihenfolge, wie sie während des Kompressionsvorgangs ausgeführt werden. In jedem dieser Kapitel wird die Rücktransformation ebenfalls beschrieben.

Zur besseren Unterscheidung wird bei Codefragmenten, welche auf der GPU ausgeführt werden, ein Rand auf der linken Seite zu sehen sein

```
1 example code in GLSL
```

und Code, welcher auf dem Hauptprozessor läuft wird ganz eingerahmt:

```
1 example code in C++
```

Genutzte Mittel:

- **OpenGL**: Eine offene Grafikbibliothek zur Grafikprogrammierung in der Version 4.3
- **Framework**: Ein bereitgestelltes OpenGL Framework mit funktionierender Renderpipeline
- **GLSL**: OpenGL Shading Language zur Programmierung der Shader, ebenfalls in der Version 4.3
- **C++**: Programmiersprache in der Version elf
- **GLFW**: Bibliothek für OpenGL zur Erstellung von Fenstern und zum Behandeln von Tastatureingaben
- **GLM**: Eine Mathematikbibliothek für Grafikprogramme

3.1 Vorstellung der Kompressionspipeline

Die implementierte Kompression für Einzelbilder unterscheidet sich nur in wenigen Aspekten von der Standard JPEG-Kompression. Die Unterschiede resultieren hauptsächlich aus einer fundamental anderen Repräsentation der Datentypen im JPEG-Standard im Gegensatz zur Repräsentation von Datentypen in OpenGL.

Während in JPEG mit einer Genauigkeit von 8 Bit pro Integerwert pro Kanal gearbeitet wird, bietet OpenGL die Möglichkeit permanent mit 32 Bit Floatwerten zu arbeiten, um mehr Genauigkeit zu garantieren, was gerade in der Bildkompression von computergenerierten Bildern vorteilhaft ist. Da synthetisierte Bilder immer bloß eine approximierte Simulation der Wirklichkeit darstellen, fallen Ungenauigkeiten sofort auf. Außerdem wird in dieser Implementation eine andere verlustfreie Kompressionstechnik genutzt, als im

JPEG-Pendant: Die Lauflängenkodierung.

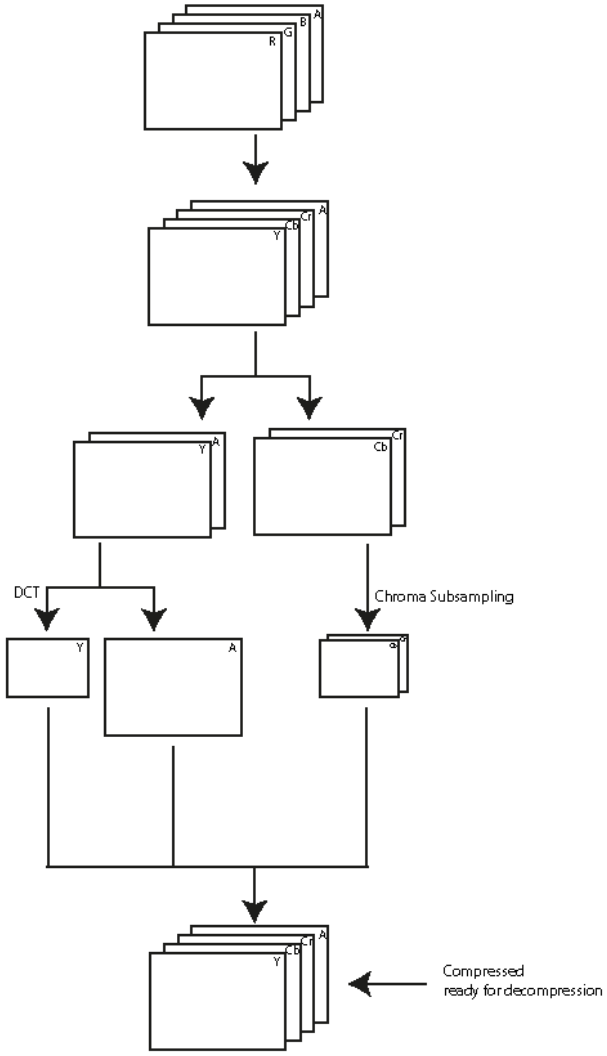


Abbildung 11: Flowgraph der verwendeten Texturen
Überlappende Texturen symbolisieren mehrkanalige Texturen

3.1.1 Umrechnen der Farbräume

Außer der Lauflängenkodierung, sind alle Schritte des Algorithmus als OpenGL Compute Shader implementiert worden, um die Parallelisierung des Verfahrens zu bewerkstelligen. Da die Compute Shader auf Texturen operieren, wird die Work Group Größe, wie in Kapitel 2.1 erklärt, auf eine zweidimensionale Größe gesetzt. Ausgangspunkt für die Implementation ist die Ausgabe des bereitgestellten Frameworks. Die Ausgabe ist eine vierkanalige Textur, mit

Rot-, Grün-, Blau- und Alphakanal, welche wiederum als Eingabe für den ersten Schritt der Implementation dient. Abbildung 11 zeigt die verwendeten Texturen für den gesamten Ablauf, außer Texturen, die bloß Zwischenergebnisse speichern.
Der Alphakanal wird in der gesamten Berechnung ignoriert, da er für die Kompression des Bildes und dessen Qualität eine untergeordnete Rolle spielt. Die Umrechnung der Farbe von RGB zu YCbCr wird nach der Transformationsvorschrift durchgeführt, welche in Kapitel 2.2.2 beschrieben wurde. Zu diesem Zweck wird im Compute Shader zunächst eine zweidimensionale lokale Work Group definiert und zwei Texturen gebunden. *inImg* dient hier als Eingabetextur im RGBA-Format. *outImg* dient als Ausgabetextur im YCbCrA-Format:

```
1 layout(binding = 0, rgba32f) readonly uniform image2D inImg;
2 layout(binding = 1, rgba32f) writeonly uniform image2D
      outImg;
3
4 layout(local_size_x = 16, local_size_y = 16) in;
```

Anschließend wird nach Transformationsvorschrift 4 jeder einzelne Farbwert umgerechnet und in die Ausgabetextur *outImg* geschrieben.

```
1  ivec2 index = ivec2 (gl_GlobalInvocationID.xy);
2
3  vec4 texel_color = imageLoad(inImg, index);
4
5  float y = (0.299f * texel_color.r) + (0.587f * texel_color.g
       ) + (0.114f * texel_color.b);
6  float cb = 0.5f -(0.168736f * texel_color.r) - (0.331264f *
       texel_color.g) + (0.5f * texel_color.b);
7  float cr = 0.5f + (0.5f * texel_color.r) - (0.418688f *
       texel_color.g) - (0.081312f * texel_color.b);
8
9  vec4 resulting_color = vec4( y, cb, cr, texel_color.a);
10
11 imageStore(outImg, index, resulting_color);
```

Zur Orientierung, an welcher Stelle des Eingabebildes der Texelwert gelesen werden muss, stellt OpenGL eine vordefinierte Funktion bereit:
gl_ GlobalInvocationID gibt einen dreidimensionalen Vektor zurück, welcher die Position innerhalb der globalen Work Group repräsentiert. In diesem Fall wurde die Größe der Work Group in der dritten Dimension auf eins gesetzt, sodass eine zweidimensionale Work Group definiert wird um die Repräsentation des Compute Shaders an die zweidimensionale Textur anzupassen. Also beschreibt der Vektor *ivec2 index* die genaue Position des zu lesenden Texels auf der Eingabetextur.

Während der Rücktransformation wird die Farbe wieder zurück in das gängige RGBA Format umgewandelt. Dies geschieht mit der in Kapitel 2.2.2 beschriebenen Transformation 5:

```
1 float r = texel_color.r + (1.402f * (texel_color.b - 0.5f));
2
3 float g = texel_color.r - (0.34414f * (texel_color.g - 0.5f)
      ) - (0.71414f * (texel_color.b - 0.5f));
4
5 float b = texel_color.r + (1.772f * (texel_color.g -0.5f));
6
7 vec4 resulting_color = vec4(r, g, b, texel_color.a);
8
9 imageStore(outImageRGBA, index, resulting_color);
```

Die bisher gezeigten Code-Ausschnitte beschreiben den Code, der auf der GPU pro Invocation des Shaders ausgeführt wird. Der C++ Code wird im Folgenden beschrieben.

Der Compute Shader, welcher die Farbtransformation ausführt, besitzt, wie bereits beschrieben, eine zweidimensionale Struktur um adäquat auf zweidimensionalen Strukturen arbeiten zu können.
Die Ausgabetextur des vorherigen Rendering Vorganges besitzt eine Breite von 1280 Pixeln und eine Höhe von 720 Pixeln. Der Compute Shader ist mit einer Breite und einer Höhe von 16 definiert worden. Zeile 18 des nachfolgenden Code-Ausschnittes sorgt dafür, dass exakt so viele globale Work Groups ausgeführt werden, dass jede Invocation des Shaders einem Texel zugeordnet werden kann. Außerdem startet Zeile 18 gleichzeitig die Ausführung der definierten Work Groups.

```
1  RGBtoYCbCr->use();
2  glBindImageTexture( 0,
3                      pass->get("fragColor"),
4                      0,
5                      GL_FALSE,
6                      0,
7                      GL_READ_ONLY,
8                      GL_RGBA32F);
9
10 glBindImageTexture( 1,
11                     texYCbCrHandle,
12                     0,
13                     GL_FALSE,
14                     0,
15                     GL_WRITE_ONLY,
16                     GL_RGBA32F);
17
18 glDispatchCompute(int(1280/16), int(720/16), 1);
19 glMemoryBarrier(GL_SHADER_STORAGE_BARRIER_BIT);
```

Zeile zwei bis neun bindet die Ausgabetextur des vorangegangen Rendering Vorgangs an den Shader. Zeile zehn bis 16 bindet eine Textur mit den gleichen Ausmaßen als Ausgabetextur an den Shader. In Zeile 19 wird eine Speichergrenze eingesetzt, die garantieren soll, dass der weitere Programmablauf so lange pausiert, bis alle Invocations in allen Work Groups ihre Arbeit beendet haben. Damit wird sichergestellt, dass alle Farbwerte im YCbCr-Format vorliegen, bevor weitere Berechnungen durchgeführt werden.

3.1.2 Reduktion der Chrominanzauflösung

Der Shader, welcher die Unterabtastung durchführt, teilt die vierkanalige Eingabetextur auf eine zweikanalige und zwei einkanalige Ausgabetexturen auf. Eine Textur speichert die zwei Farbkanäle Cb und Cr in geringerer Auflösung. Die beiden einkanaligen Texturen werden benutzt um den Helligkeitskanal vom Alphakanal zu trennen. Die Helligkeitsinformationen werden benutzt um im nächsten Schritt die diskrete Kosinustransformation durchzuführen. Der Alphakanal wird nur isoliert um ihn später wieder hinzufügen zu können, sollte er für andere Zwecke außer der Bildkompression noch benötigt werden.
Das sog. Chroma Subsampling oder zu Deutsch Farbunterabtastung, lässt sich theoretisch in allen möglichen Unterabtastungen in vertikaler und horizontaler Richtung realisieren. In dieser Implementation wurde eine horizontale Unterabtastung gewählt, da das Framework Ausgabetexturen im Breitbildformat generiert. Dafür wird an den Shader, welcher die Unterabtastung ausführt, eine Ausgabetextur gebunden, die nicht mit den Maßen der Eingabetextur übereinstimmt. Die Textur wird folgendermaßen erstellt:

```
1 glGenTextures(1, &texCbCrHandle);
2 glActiveTexture(GL_TEXTURE0);
3 glBindTexture(GL_TEXTURE_2D, texCbCrHandle);
4 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
      GL_NEAREST);
5 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
      GL_NEAREST);
6 glTexImage2D(GL_TEXTURE_2D, 0, GL_RG32F, width/2, height, 0,
       GL_RG, GL_FLOAT, NULL);
```

In Zeile sechs wird die horizontale und vertikale Größe der Textur bestimmt. Da sie nur noch die Hälfte der ursprünglichen Farbwerte speichern soll, muss sie auch nur noch halb so breit sein wie die Eingabetextur. Der Shader Code selbst sieht folgendermaßen aus:

```
1 #version 430
2 layout(binding = 0, rgba32f) readonly uniform image2D inImg;
3 layout(binding = 1, rg32f) writeonly uniform image2D
      outImgCbCr;
4 layout(binding = 2, r32f) writeonly uniform image2D outImgY;
```

```
5  layout(binding = 3, r32f) writeonly uniform image2D outImgA;
6
7  layout(local_size_x = 16, local_size_y = 16) in;
8  void main() {
9          ivec2 index = ivec2 (gl_GlobalInvocationID.x,
               gl_GlobalInvocationID.y);
10         vec4 texel_color = imageLoad(inImg, index);
11
12         if((index.x % 2)==0)
13                 imageStore(outImgCbCr, ivec2(index.x / 2,
                       index.y), vec4(vec2(texel_color.gb), 0,
                       0));
14         imageStore(outImgY, index, vec4(texel_color.r, 0, 0,
               0));
15         imageStore(outImgA, index, vec4(texel_color.a, 0, 0,
               0));
16 }
```

Dabei garantiert Zeile 12, dass nur jeder zweite Pixel in horizontaler Richtung in der Farbtextur abgespeichert wird.
Der Shader wird auf die gleiche Weise gebunden und ausgeführt wie der vorherige Shader.
Da der Helligkeitskanal in eine separate Textur geschrieben wurde (Zeile 14), stehen nun die Helligkeitsinformationen isoliert zur Verfügung, um sie mit Hilfe der diskreten Kosinusfunktion in den Frequenzbereich zu transformieren.
Das Ergebnis eines Chromina Subsamplings ist in Kapitel 2.2.2 auf Abbildung 5 zu sehen.

3.1.3 Diskrete Cosinus Transformation

Die diskrete Kosinustransformation, wie in Formel 6 definiert, besteht aus zu vielen Operationen um performant auf einer GPU implementiert zu werden. Eine Erste Optimierung kann bereits geschehen, wenn man die DCT zuerst spaltenweise und anschließend zeilenweise ausführt. Die eindimensionale DCT ist wie folgt definiert:

$$D(i) = \sqrt{\frac{2}{N}} C(i) \sum_{x=0}^{N-1} p(x) \cos\left[\frac{(2x+1)\, i\pi}{2N}\right] \tag{10}$$

Zur eindimensionalen DCT existieren heute Optimierungen, die den Berechnungsaufwand um ein vielfaches verkleinern. Der Arai-Agui-Nakajima-Algorithmus kurz AAN-Algorithmus ist die bis Heute effizienteste Optimierung der DCT mit lediglich 29 Additionen und 5 Multiplikationen [WBP].
Die Berechnungen wurden in einem sog. Flow-Graph verdeutlicht und werden in Abbildung 12 gezeigt.

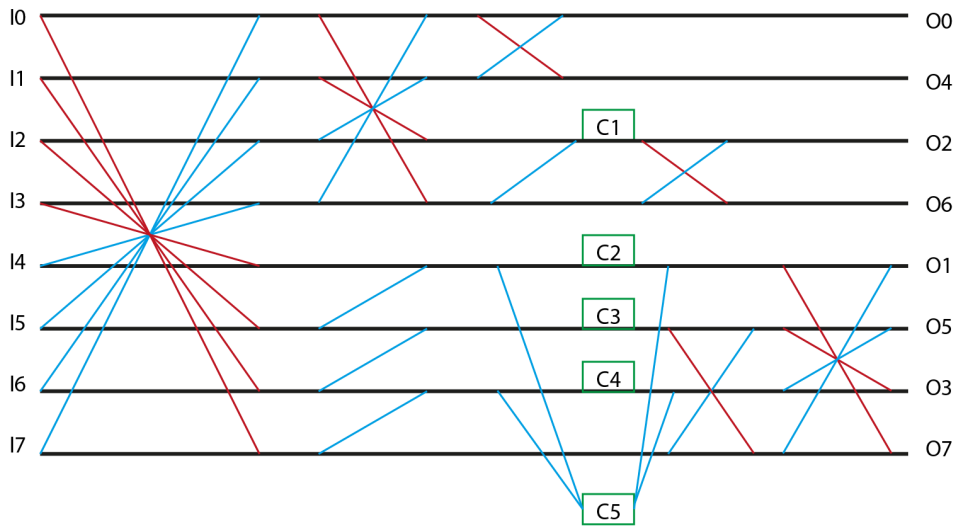


Abbildung 12: Flowgraph von Arai, Agui & Nakajima [WBP];
Jede horizontale Line beschreibt eine Input variable (links: I0 bis I7) und eine Outputvariable (rechts: O0 bis O7)
Blaue Linien bedeuten Additionen, rote Linien bedeuten Subtraktionen und Kästen symbolisieren Multiplikationen mit konstanten Skalierungsfaktoren

Der AAN-Algorithmus wurde benutzt um die DCT als Compute Shader zu implementieren. Dafür wird zunächst eine lokale Work Group erstellt, die keine dimensionale Ausbreitung besitzt, da jede Invocation eine einzelne Zeile des 8x8 Blocks transformieren soll:

```
1 layout(local_size_x = 1, local_size_y=1, local_size_z=1) in;
```

Anschließend muss der Index der Invocation an die Eingabetextur angepasst werden:

```
1 ivec2 index = ivec2 (gl_GlobalInvocationID.x * 8,
2                      gl_GlobalInvocationID.y);
```

Nun kann jede Zeile des 8x8 Blocks einzeln eingelesen werden. Die eingelesenen Werte entsprechen den Eingabewerten I0 bis I7 aus der, in Abbildung 12 gezeigten, Flowchart.

```
1 for(int i = 0; i < 8; i++){
2     inputValues[i] = imageLoad(inImg,
3                                ivec2(index.x + i, index.y)).x;
4         }
```

Da nun jede Invocation acht Werte besitzt, kann der Rest des Shader Programms die DCT-Berechnung nach Abbildung 12 umsetzen.

```
1 float temporaryValue0 = inputValues[0] + inputValues[7];
2 float temporaryValue7 = inputValues[0] - inputValues[7];
```

```
3  float temporaryValue1 = inputValues[1] + inputValues[6];
4  float temporaryValue6 = inputValues[1] - inputValues[6];
5  float temporaryValue2 = inputValues[2] + inputValues[5];
6  float temporaryValue5 = inputValues[2] - inputValues[5];
7  float temporaryValue3 = inputValues[3] + inputValues[4];
8  float temporaryValue4 = inputValues[3] - inputValues[4];
9
10 float tmp1 = temporaryValue0 + temporaryValue3;
11 float tmp2 = temporaryValue1 + temporaryValue2;
12 float tmp3 = temporaryValue1 - temporaryValue2;
13 float tmp4 = temporaryValue0 - temporaryValue3;
14
15 imageStore(outImg, ivec2(index.x + 0, index.y), vec4(tmp1 +
       tmp2, 0, 0, 0));     //write Ouptut O0
16 imageStore(outImg, ivec2(index.x + 4, index.y), vec4(tmp1 -
       tmp2, 0, 0, 0));     //write Output O4
17
18 float mul1 = (tmp3 + tmp4) * 0.707106781;
19
20 imageStore(outImg, ivec2(index.x + 2, index.y), vec4(tmp4 +
       mul1, 0, 0, 0));     //write Output O2
21 imageStore(outImg, ivec2(index.x + 6, index.y), vec4(tmp4 -
       mul1, 0, 0, 0));     //write Output O6
22
23 tmp1 = temporaryValue4 + temporaryValue5;
24 tmp2 = temporaryValue5 + temporaryValue6;
25 tmp3 = temporaryValue6 + temporaryValue7;
26
27 float mul5 = (tmp1 - tmp3) * 0.382683433;
28 float mul2 = 0.541196100 * tmp1 + mul5;
29 float mul4 = 1.306562965 * tmp3 + mul5;
30 float mul3 = tmp2 * 0.707106781;
31
32 float mul6 = temporaryValue7 + mul3;
33 float mul7 = temporaryValue7 - mul3;
34
35 imageStore(outImg, ivec2(index.x + 5, index.y), vec4(mul7 +
       mul2, 0, 0, 0));     //write Output O5
36 imageStore(outImg, ivec2(index.x + 3, index.y), vec4(mul7 -
       mul2, 0, 0, 0));     //write Output O3
37 imageStore(outImg, ivec2(index.x + 1, index.y), vec4(mul6 +
       mul4, 0, 0, 0));     //write Output O1
38 imageStore(outImg, ivec2(index.x + 7, index.y), vec4(mul6 -
       mul4, 0, 0, 0));     //write Output O7
```

Durch die Ausführung des Shaders ist der vertikale Schritt der 2D-DCT vollendet. Die vertikale Transformation geschieht analog, außer, dass über die Spalten iteriert werden muss:

```
1 ivec2 index = ivec2 (gl_GlobalInvocationID.x,
```

```
2                                       gl_GlobalInvocationID.y * 8);
3
4   for(int i = 0; i < 8; i++){
5           temp[i] = imageLoad(inImg,
6                                   ivec2(index.x, index.y + i)).x;
7   }
```

Um die Shaderausführung zu starten ist es notwendig im Hauptprogramm den Shaderaufruf an die Work Group Größe anzupassen:

```
1  horizontalDCT->use();
2  glBindImageTexture(0, texYHandle, 0, GL_FALSE, 0,
       GL_READ_ONLY, GL_R32F);
3  glBindImageTexture(1, texHDCTHandle, 0, GL_FALSE, 0,
       GL_WRITE_ONLY, GL_R32F);
4  glDispatchCompute(int(width/8), height, 1);
5  glMemoryBarrier(GL_SHADER_STORAGE_BARRIER_BIT);
```

Analog geschieht dies für den Shader, welcher die Spalten transformiert:

```
1  verticalDCT->use();
2  glBindImageTexture(0, texHDCTHandle, 0, GL_FALSE, 0,
       GL_READ_ONLY, GL_R32F);
3  glBindImageTexture(1, texYHandle, 0, GL_FALSE, 0,
       GL_WRITE_ONLY, GL_R32F);
4  glDispatchCompute(width, int(height/8), 1);
```

Nachdem die eindimensionale DCT auch die Spalten transformiert hat, ist die gesamte 2D-DCT vollendet. Die Ausgabetextur des vertikalen Compute Shaders, also das Ergebnis der zweidimensionalen DCT, ist in Abbildung 13 verdeutlicht.

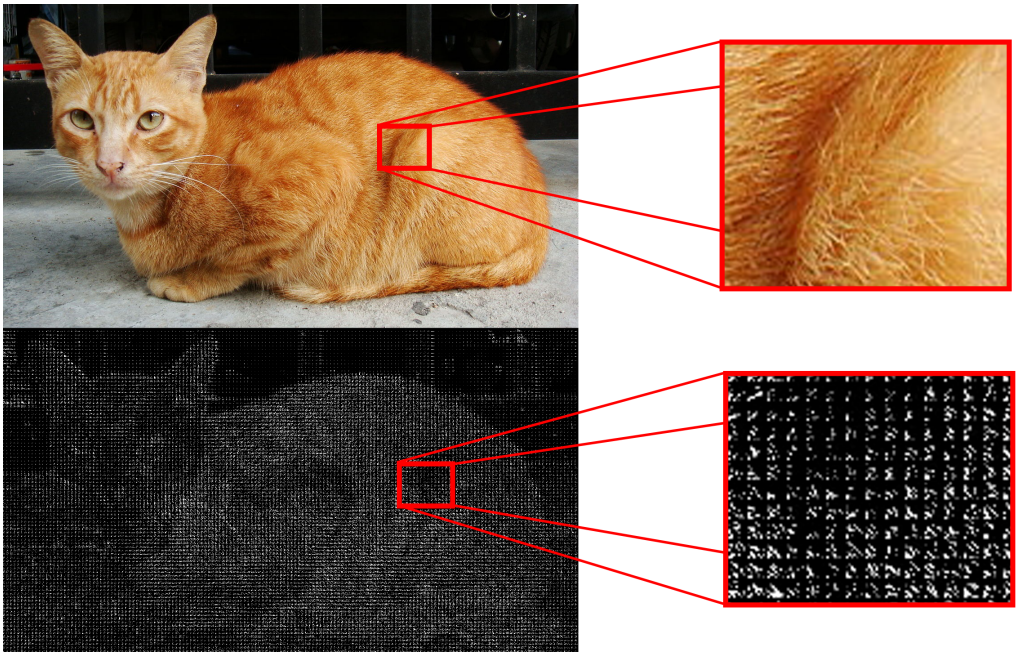


Abbildung 13: Oben: Bild im Ortsbereich, vor der DCT
Unten: das gleiche Bild transformiert in den Frequenzbereich, nach der DCT

3.1.4 Quantisierung

Die Quantisierung wird unmittelbar nach der diskreten Kosinustransformation durchgeführt. Es sollen, wie in Kapitel 2.2.2 beschrieben, die Koeffizienten der DCT auf null gesetzt werden. Die in dieser Implementierung benutzte Quantisierungsmatrix, ist die von JPEG empfohlene Quantisierungsmatrix für Luminanz, also den Helligkeitskanal. Sie ist in Abbildung 9 zu sehen. Dass einzelne Koeffizienten der DCT während der Quantisierung auf null gesetzt werden, wird in der ursprünglichen Kompression von JPEG durch das Runden auf den nächsten ganzzahligen Wert erzielt. JPEG benutzt für seine Berechnungen ganzzahlige Eingabetypen mit einer Genauigkeit von 8 Bit. Da die Eingabetexturen für diese Implementation aus einem Rendering Vorgang stammen, liegen die Werte jedoch als 32 Bit Gleitkommazahl vor. Diese Genauigkeit soll erhalten bleiben. Aus diesem Grund gestaltet sich die Quantisierung in dieser Kompression geringfügig anders.
Die Quantisierungsvorschrift 8 muss dahingehend verändert werden, dass nicht mehr gerundet wird:

$$Q(i,j) = \frac{D(i,j)}{q(i,j)} \tag{11}$$

Da nun nicht mehr gerundet wird, werden hinreichend kleine Koeffizienten

allerdings nicht mehr auf null gesetzt. Für diesen Zweck wird ein unterer Grenzwert definiert, den ein Koeffizient überschreiten muss um nicht auf null gesetzt zu werden. Überschreitet ein Koeffizient diesen Schwellwert, wird er in seiner ganzen Genauigkeit von 32 Bit abgespeichert. Nachfolgend der Shader Code für die Quantisierung:

```
1  ivec2 index = ivec2 (gl_GlobalInvocationID.xy);
2  float barrier = 0.02;
3  float scaling = 1.0;
4
5  ivec2 newIndex;
6  newIndex.x = int (mod(index.x, 8));
7  newIndex.y = int (mod(index.y, 8));
8
9  vec4 texel_color = imageLoad(inImg, index);
10
11 texel_color = texel_color / (scaling * qTable[newIndex.x][
       newIndex.y]);
12
13 if(abs(texel_color.r)<= barrier)
14         imageStore(outImg, index, vec4(0, 0, 0, 0));
15 else
16         imageStore(outImg, index, vec4(texel_color.r, 0, 0,
               0));
```

Die Quantisierungsmatrix qTable wird als shared Variable deklariert und steht somit jeder Invocation zur Verfügung. Um den Wert zu ermitteln, durch den ein Koeffizient geteilt werden soll, muss ein lokaler Index errechnet werden. Das ist in Zeile 5-7 zu sehen. Die Entscheidung ob der Wert abgespeichert wird, oder ob der Koeffizient zu unrelevant für das menschliche Auge ist, wird in Zeile 13 getroffen. Der Wert scaling aus Zeile 3 skaliert die gesamte Quantisierungsmatrix und ermöglicht so variable Kompressionsraten. Das Hauptprogramm startet den Quantisierungsshader auf Grund der Work Group Größe wie den Shader zur Farbraumunterabtastung:

```
1  quantize->use();
2  glBindImageTexture(0, texYHandle, 0, GL_FALSE, 0,
       GL_READ_ONLY, GL_R32F);
3  glBindImageTexture(1, texQuantHandle, 0, GL_FALSE, 0,
       GL_WRITE_ONLY, GL_R32F);
4  glDispatchCompute(int(width/8), int(height/8), 1);
5  glMemoryBarrier(GL_SHADER_STORAGE_BARRIER_BIT);
```

Ohne die Quantisierung wäre der Vorgang verlustfrei und die Speicherplatzersparnis gering. Durch die Quantisierung wird eine höhere Redundanz erreicht (auf Grund der vielen Nullen), welche anschließend während der Lauflängenkodierung ausgenutzt wird, um die Speicherplatzersparnis drastisch zu erhöhen. Aus diesem Grund stellt die verlustfreie Kompressionstechnik den letzten Schritt der implementierten Pipeline dar. Sie wird in Kapitel 3.1.6

näher beschrieben. das Ergebnis der Quantisierung ist in Abbildung 14 zu sehen.

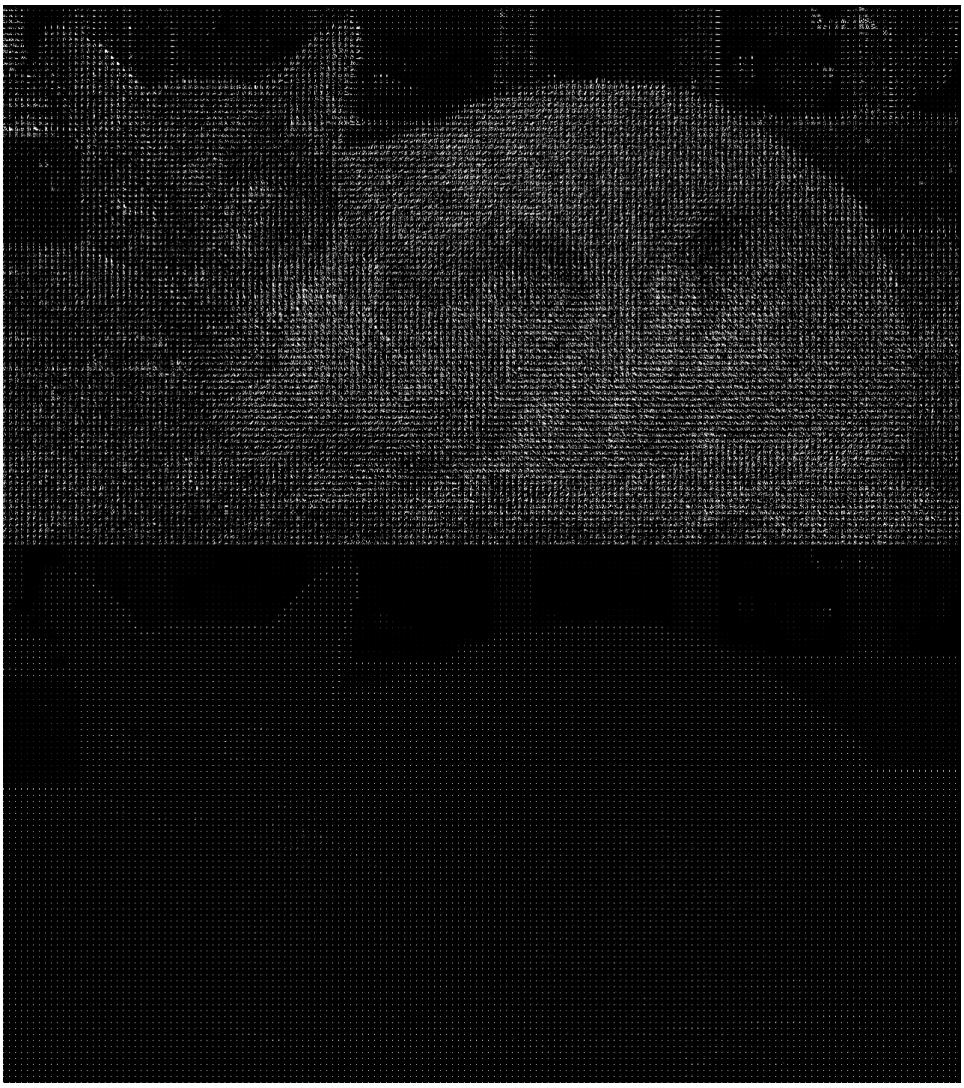


Abbildung 14: Oben: Vor der Quantisierung
Unten: Nach der Quantisierung

3.1.5 Zig-Zag-Scan

Bevor die mit vielen Nullen versehene Textur effektiv per Lauflängenkodierung komprimiert werden kann, müssen die quantisierten Koeffizienten in eine adäquate Reihenfolge gebracht werden. Durch den Aufbau der Quantisierungsmatrix werden Koeffizienten höherer Frequenzen stärker quantisiert als die von niedrigeren Frequenzen. Es ist also sinnvoll, die Koeffizienten anhand steigender Frequenzen zu sortieren, da die Ergebnisse der Quantisie-

rung so bestmöglich ausgenutzt werden können. Um dies zu bewerkstelligen, wird die Koeffizientenmatrix in einem Zickzackmuster traversiert, wie bereits in Kapitel 2.2.3 auf Abbildung 10 zu sehen ist.
Die Implementation des sog. Zig-Zag-Scans benötigt eine Ausgabetextur mit veränderten Ausmaßen, um als Compute Shader zu funktionieren. Jeder 8x8 Block wird in einen 64x1 Block übersetzt. Also muss die Ausgabetextur acht Mal so breit sein wie die Eingabetextur. Die Höhe beträgt dabei nur noch ein Achtel der ursprünglichen Höhe. Eine solche Textur wird erstellt und an den Shader gebunden. Der Shader Code ist dem der Quantisierung sehr ähnlich. Die Position der neuen Reihenfolge wird in Form einer Lookuptable als shared Variable deklariert, damit alle Invocations einer lokalen Work Group Zugriff darauf haben.

```
1  layout (local_size_x = 8, local_size_y = 8) in;
2
3  shared int orderTable[8][8] = {
4                  {0, 1, 5, 6, 14, 15, 27, 28},
5                  {2, 4, 7, 13, 16, 26, 29, 42},
6                  {3, 8, 12, 17, 25, 30, 41, 43},
7                  {9, 11, 18, 24, 31, 40, 44, 53},
8                  {10, 19, 23, 32, 39, 45, 52, 54},
9                  {20, 22, 33, 38, 46, 51, 55, 60},
10                 {21, 34, 37, 47, 50, 56, 59, 61},
11                 {35, 36, 48, 49, 57, 58, 62, 63}};
12
13 void main() {
14
15         ivec2 index = ivec2 (gl_GlobalInvocationID.xy);
16
17         ivec2 orderIndex;
18         orderIndex.x = int (mod(index.x, 8));
19         orderIndex.y = int (mod(index.y, 8));
20
21         ivec2 zigZagIndex;
22         zigZagIndex.x = int(floor(index.x/8) * 64 ) +
               orderTable[orderIndex.x][orderIndex.y];
23         zigZagIndex.y = int(floor(index.y/8));
24
25         vec4 texel_color = imageLoad(inImg1, index) ;
26
27         imageStore(outImg, zigZagIndex, vec4(texel_color.r,
               0, 0, 0));
```

Jede Invocation ermittelt nun zuerst einen Index für die Lookuptable, zu sehen in Zeile 17 bis 19. Anhand dieses Index ist es nun möglich die finale Position in der umsortierten Ausgabetextur zu ermitteln (Zeile 21 bis 23).

Die während der Dekompression erforderliche Umsortierung ist ähnlich im-

plementiert. Sie benutzt ebenfalls eine Lookuptabelle. Diese hat zwei Spalten: Eine für den Index der Spalten und eine für den Index der Zeilen der 8x8 Matrix.

```
shared int orderTable[2][64] = {
        {0,1,0,0,1,2,3,2,1,0,0,1,2,3,4,5,4,3,2,1,0,
        0,1,2,3,4,5,6,7,6,5,4,3,2,1,0,1,2,3,4,5,6,
        7,7,6,5,4,3,2,3,4,5,6,7,7,6,5,4,5,6,7,7,6,7},

        {0,0,1,2,1,0,0,1,2,3,4,3,2,1,0,0,1,2,3,4,5,6,
        5,4,3,2,1,0,0,1,2,3,4,5,6,7,7,6,5,4,3,2,1,2,
        3,4,5,6,7,7,6,5,4,3,4,5,6,7,7,6,5,6,7,7}};
```

Die Eingabetextur repräsentiert für diesen Shader jeden ursprünglichen 8x8 Block als 64x1 Block. Da sich die Dimensionen der Workgroup an die Eingabetextur anpassen lassen,

```
layout (local_size_x = 64, local_size_y = 1) in;
```

lässt sich auch in diesem Fall wieder ein adäquater Index errechnen:

```
        ivec2 index = ivec2 (gl_GlobalInvocationID.xy);

        int newIndex;
        orderIndex = int(mod(index.x, 64));

        ivec2 zigZagIndex;
        zigZagIndex.x = orderTable[1][newIndex];
        zigZagIndex.y = orderTable[0][newIndex];

        vec4 texel_color = imageLoad(inImg1, index);

        ivec2 finalIndex;
        finalIndex.x = int(floor(index.x/64)) * 8 ;
        finalIndex.y = index.y * 8;

        finalIndex += zigZagIndex;
```

Auf Seiten des Hauptprogrammes wird die Umsortierung der 8x8 Blöcke genauso gestartet wie bei der Quantisierung. Jede Work Group sortiert einen 8x8 Block:

```
forwardZigZag->use();
glBindImageTexture(0, texQuantHandle, 0, GL_FALSE, 0,
    GL_READ_ONLY, GL_R32F);
glBindImageTexture(1, texZigZagHandle, 0, GL_FALSE, 0,
    GL_WRITE_ONLY, GL_R32F);
glDispatchCompute(int(width / 8), int(height / 8), 1);
glMemoryBarrier(GL_SHADER_STORAGE_BARRIER_BIT);
```

Da die Workgroup der späteren Umsortierung auf Decoder-Seite eine 64x1 Work Group definiert, wird der Aufruf leicht abgewandelt:

```
1 rZigZag->use();
2 glBindImageTexture(0, texZigZagHandle, 0, GL_FALSE, 0,
      GL_READ_ONLY, GL_R16F);
3 glBindImageTexture(1, texQuant2Handle, 0, GL_FALSE, 0,
      GL_WRITE_ONLY, GL_R16F);
4 glDispatchCompute(int(width * 8 / 64), int(height / 8), 1);
5 glMemoryBarrier(GL_SHADER_STORAGE_BARRIER_BIT);
```

In Zeile 4 des oben stehenden Codefragmentes wird der Aufruf der Work Groups an die neue Eingabetextur angepasst.

3.1.6 Lauflängenkodierung

Die Lauflängenkodierung (engl.: "Run Length Encoding"kurz: RLE) stellt den letzten Schritt dieser Implementation auf Encoder-Seite dar. Demnach stellt die Lauflängendekodierung den ersten Schritt der Implementation der Decoder-Seite dar. Das Prinzip dieser verlustlosen Kompressionstechnik ist simpel -
Jede Sequenz von identischen Symbolen wird ersetzt durch einmaliges Speichern des Symbols und einmaliges Speichern der Häufigkeit dieses Symbols: z.B.: Aus *AAABAAAAABB* wird *3A 1B 5A 2B*.

Da während der Quantisierung zahlreiche Koeffizienten den Wert 0 besitzen, ist durch die Lauflängenkodierung eine hohe Speicherplatzersparnis zu erwarten.
Bevor die verlustlose Kompression jedoch beginnen kann, muss die Textur aus dem GPU-Speicher auf den Hauptspeicher übertragen werden. Für diesen Zweck stellt OpenGL die Funktion *glGetTexImage* zur Verfügung. Zunächst muss Speicher für die Textur allokiert werden:

```
1 float* data;
2
3 glBindTexture(GL_TEXTURE_2D, texRLEHandle);
4 glGetTexLevelParameteriv(GL_TEXTURE_2D, 0, GL_TEXTURE_WIDTH,
      &tWidth);
5 glGetTexLevelParameteriv(GL_TEXTURE_2D, 0, GL_TEXTURE_HEIGHT
      , &tHeight);
6
7 data = (float*)malloc( sizeof(float) * tHeight * tWidth);
```

In Zeile vier und fünf werden Höhe und Breite der Textur ermittelt, um mit ihrer Hilfe den benötigten Speicherplatz zu errechnen. Der Speicher wird dann in Zeile 7 allokiert. Dies kann zum Programmstart einmal getan werden und muss danach nicht pro Bild wiederholt werden, da sich die Maße der Textur zur Laufzeit nicht ändern. Aus diesem Grund muss der allokierte Speicher auch erst zum Programmende wieder freigegeben werden.

Wenn nun alle Schritte, einschließlich der Umsortierung, durch den Zick-Zack-Scan beendet sind, kann die Textur in den Hauptspeicher geladen werden:

```
1 glBindTexture(GL_TEXTURE_2D, texRLEHandle);
2 glGetTexImage(GL_TEXTURE_2D, 0, GL_RED, GL_FLOAT, data);
3 glBindTexture(GL_TEXTURE_2D, 0);
```

Die benötigte Textur wird gebunden und in den zuvor allokierten Speicher übertragen. Die Funktion aus Zeile zwei speichert die Textur nun als einfache Gleitkommawerte mit vier Byte pro Wert zeilenweise in das Array data. Der Pointer, der auf dieses Array zeigt, kann nun an die Lauflängenkodierung übergeben werden. Die Funktion der Lauflängenkodierung erwartet noch zwei andere Parameter. Einen Container für die zu speichernden Werte und einen für die Anzahl der Werte. Diese Container werden erstellt und an die Funktion übergeben:

```
1 vector<float>* val = &values;
2 vector<char>* coun = &counts;
3
4 runlengthEncoding(data, val, coun);
```

Die Funktion runlengthEncoding() führt nun die Lauflängenkodierung durch:

```
1  void runlengthEncoding(float *array, vector<float>* outValue
       , vector<char>* outCounts){
2  float colorOld = array[0];
3  float color;
4  char count = 1;
5
6  for(int i = 1; i < (tWidth * tHeight) ; i++){
7          color = array[i];
8          if(colorOld != color){
9                  outValue->push_back(colorOld);
10                 outCounts->push_back(count);
11                 colorOld = color;
12                 count = 1;
13         }
14         else{
15                 count++;
16         }
17 }
18         outValue->push_back(colorOld);
19         outCounts->push_back(count);
```

Da durch die Quantisierung viele Koeffizienten auf null gesetzt wurden, wird der Fall, dass zwei benachbarte Werte gleich sind, deutlich öfter eintreten, was die Kompressionsrate erhöht. Es wird über alle Werte des Arrays data iteriert. Der Wert des aktuell betrachteten Pixels wird in der Variable color in Zeile sieben zwischengespeichert. In Zeile acht wird ermittelt, ob die

Werte sich unterscheiden. Wenn dies der Fall ist, wird die aktuelle Anzahl und der dazugehörige Wert in den beiden Vektoren outValue und outCounts gespeichert (Zeile neun und zehn).
Wenn die Werte jedoch gleich sind, wird nur die Zählervariable count erhöht (Zeile 15). Nach der Lauflängenkodierung ist die Kompression abgeschlossen. Die Dekompression auf Decoderseite sieht folgendermaßen aus:

```
1 void rleDecodeFromVec(vector<float>* value, vector<char>*
    counts, float* array){
2       int i,j,k = 0;
3       for(i = 0 ; i < counts->size(); i++){
4               for(j = 0; j < counts->at(i); j ++){
5                       array[k+j] = value->at(i);
6               }
7               k+=j;
8       }
```

Wie in den Parametern der Funktion zu sehen ist, benötigt die Funktion drei Eingabewerte. Der erste Parameter vector<float>* value ist das Ausgabearray. Es muss genauso viel Speicher fassen können, wie zuvor für die Ausgabetextur des Zick-Zack-Scans benötigt wurde. Die beiden anderen Parameter sind eben die Ausgabecontainer der Lauflängenkodierung. Es wird nun jeder Eintrag der beiden Container betrachtet und für jeden Eintrag der Wert aus value so oft in das Ausgabe-Array geschrieben, wie es der Eintrag aus counts vorgibt. Nachdem alle Einträge aus den Vektoren abgearbeitet sind, steht wieder ein Array zur Verfügung, welches mit Hilfe des OpenGL-Befehls glTexSubImage2D wieder als Textur auf den GPU-Speicher übertragen werden kann.

```
1 glBindTexture(GL_TEXTURE_2D, tex9Handle);
2 glTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0,
3                 tWidth, tHeight, GL_RED,
4                 GL_FLOAT, data2);
5 glBindTexture(GL_TEXTURE, 0);
```

Die Zieltextur wird in Zeile eins gebunden und anschließend in Zeile zwei mit den Inhalten des Arrays data2 beschrieben. Sie steht nun wieder als normale OpenGL-Textur zur Verfügung.
Die beiden Container für die Ausgabe der Lauflängenkodierung müssen an dieser Stelle gelöscht werden, damit der Speicher wieder freigegeben werden kann. Ansonsten würde für jedes Bild zwei neue Container erstellt werden und der Speicher würde volllaufen, was einen Programmabsturz zur Folge hätte:

```
1 values.clear();
2 counts.clear();
```

4 Ergebnisse

4.1 Kompression und Bildqualität

Die Kompression eines Bildes lässt sich auf unterschiedliche Weise ermitteln. Die einfachste Möglichkeit stellt das Kompressionsverhältnis C_R dar. Es wird auch oft Kompressionsrate oder Kompressionsfaktor genannt.
Es stellt die Datenmenge des ursprünglichen Bildes in Relation zu der des komprimierten Bildes[Str]:

$$C_R = \frac{Datenmenge\,(Originalbild)}{Datenmenge\,(komprimiertesBild)} \tag{12}$$

Die Notation des errechneten Quotienten C_R ist oftmals nicht einheitlich. Er kann als Dezimalbruch, als prozentuales Verhältnis oder als Quotient in der Form 1:X (gesprochen: eins zu x) angegeben werden. Um Einheitlichkeit zu erreichen, wird im Folgenden das Kompressionsverhältnis als prozentuales Verhältnis angegeben. Wenn ein Ursprungsbild also 3600 KB Speicherplatz benötigt und das zugehörige komprimierte Bild 360 KB, so ergibt sich eine Kompressionsrate von 10%. Anders ausgedrückt beschreibt die Kompressionsrate, wie viel Prozent des ursprünglichen Speicherplatzbedarfs die komprimierte Bilddatei benötigt.
Zusätzlich ist es bei der Kompression von Bilddaten üblich, die Kompressionsleistung mit Hilfe der Bitrate anzugeben. Hierzu wird die Datenmenge N_B zur Anzahl der Symbole N_A in Relation gesetzt:

$$R = \frac{N_B}{N_A} \tag{13}$$

Dieser Quotient beschreibt also wie viele Bits durchschnittlich für die Darstellung eines Symbols, in diesem Fall eines Pixels, benötigt werden. In der Beurteilung der implementierten Kompression sollen die beiden Werte Kompressionsrate und Kompressionsleistung bewertet werden.

Um die Bildqualität der komprimierten Bilder zu bewerten, sollen Qualitätsmaße eingeführt werden, die eine objektive Bewertung ermöglichen. Qualitätsmaße bauen meist auf sog. Verzerrungsmaßen auf. Ein solches Maß für die Verzerrung ist bspw. der mittlere quadratische Fehler, engl. *mean square error*:

$$MSE = \frac{1}{N}\sum_{i=1}^{N}\left(x_i - x_i'\right)^2 \tag{14}$$

In Formel 14 steht x_i für den Pixelwert und x_i' für den rekonstruierten Pixelwert. Die Differenz zwischen einem Pixel im Originalbild und dem zugehörigen Pixel im rekonstruierten Bild wird quadriert, damit größere Differenzen stärker ins Gewicht fallen. Das Qualitätsmaß, welches zur Bewertung der

Bildqualität herangezogen werden soll, ist das sog. Spitzen-Signal-Rausch-Verhältnis (engl. peak-signal-to-noise-ratio kurz PSNR).
Es ist definiert durch:

$$PSNR = 10 \cdot \log_{10} \frac{255^2}{MSE} \tag{15}$$

für einen Wertebereich von $[0, 255]$. Dieses Qualitätsmaß bezieht das Verzerrungsmaß MSE in die Berechnung mit ein. MSE und PSNR haben ein antiproportionales Verhältnis - während ein hoher MSE für eine schlechte Bildqualität steht, steht ein hoher PSNR für eine gute Bildqualität und ein hoher MSE bedeutet ein niedriges PSNR [Str].
Da die Bilddaten einen Wertebereich von $[0, 1]$ besitzen, ist es notwendig die Werte in den Wertebereich von $[0, 255]$ zu transformieren, bevor der MSE berechnet wird.

Typischerweise werden Verzerrungs- und Qualitätsmaße nur für den Helligkeitskanal errechnet und verglichen [Str]. Der Grund dafür ist, die in Kapitel 2.2.2 behandelte besondere Wahrnemungsfähigkeit des Menschen bezüglich Helligkeit. Außerdem ist der Qualitätsverlust sowie die Kompressionsrate dieser Implementation für die beiden Farbkanäle konstant - es werden nur die Hälfte der Farbinformationen gespeichert, was bedeutet, dass die Kompressionsrate 50% für die Farbkanäle beträgt und der subjektive Qualitätsverlust ist nicht vom Menschen feststellbar.

Nachfolgend werden drei Testbilder bei unterschiedlicher Kompressionsrate mit ihren Originalbildern verglichen. Die Originalbilder haben unterschiedliche Auflösungen, um zu zeigen, dass die Verzerrungs- und Qualitätsmaße keine subjektive Bildqualität wiederspiegeln, sondern lediglich einen Richtwert darstellen, der im besten Fall mit der subjektiven Wahrnehmung korreliert.

(a) Kompressionsrate: 50%; Bitrate: 16.04; MSE: 3.48; PSNR: 42.71 dB

(b) Kompressionsrate: 25%; Bitrate: 8.01 MSE: 9.35; PSNR: 38.42 dB

(c) Kompressionsrate: 10%; Bitrate: 3.31; MSE: 29.55; PSNR:33.42 dB

(d) Kompressionsrate: 5%; Bitrate: 3.31; MSE: 99.5; PSNR: 28.15 dB

(e) Kompressionsrate: 3%; Bitrate: 0.99; MSE: 993.8; PSNR 18.15 dB

Abbildung 15: Originalbild 512x512 mit einer Bitrate von 32

In Abbildung 15(a) bis 15(e) ist zu sehen, dass der PSNR bei zunehmendem MSE abnimmt. Erste optische Abstriche sind spätestens ab 5% Kompressionsrate zu sehen. Bis zu einer Kompression auf 10% sind Unterschiede zum Original nicht auszumachen. Ab einer Kompression von 5% benötigt man durchschnittlich nur noch 3.31 Bit, um einen Pixel des Ausgangsbildes zu kodieren.

(a) Kompressionsrate: 50%; Bitrate: 16.00; MSE: 4.74; PSNR: 41.35 dB

(b) Kompressionsrate: 25%; Bitrate: 8.00 MSE: 55.15; PSNR: 30.71 dB

(c) Kompressionsrate: 10%; Bitrate: 3.21; MSE: 14.66; PSNR: 27.17 dB

(d) Kompressionsrate: 5%; Bitrate: 1.62; MSE: 169; PSNR: 25.85 dB

(e) Kompressionsrate: 3%; Bitrate: 0.99; MSE: 232; PSNR 24.4 dB

(f) Kompressionsrate: 2.89%; Bitrate: 0.92; MSE: 513.7; PSNR 21.02 dB

Abbildung 16: Originalbild 1280x720 mit einer Bitrate von 32

(a) Kompressionsrate: 50%; Bitrate: 15.99; MSE: 66.6; PSNR: 29.8 dB

(b) Kompressionsrate: 20%; Bitrate: 6.4 MSE: 303.9; PSNR: 23.3 dB

(c) Kompressionsrate: 10%; Bitrate: 3.19; MSE: 522; PSNR: 20.9 dB

(d) Kompressionsrate: 4%; Bitrate: 1.48; MSE: 807; PSNR: 19.0 dB

(e) Kompressionsrate: 2.56%; Bitrate: 0.82; MSE: 2061.1; PSNR 14.99 dB

Abbildung 17: Originalbild 1920x1080 mit einer Bitrate von 32

Bei den anderen Testbildern werden gängigere Auflösungen gewählt, wie sie heute in vielen Fernsehern und Computerbildschirmen Anwendung finden. Der Wert des MSE steigt schneller an, wenn das Ursprungsbild eine höhere Auflösung besitzt. Demnach sinkt auch der PSNR-Wert schneller als bei niedrig aufgelösten Bildern. Ein PSNR von ca. 30 dB ist in Abbildung 15(d) bei einer Kompressionsrate von 5%, in Abbildung 16(b) bereits bei einer

Kompressionsrate von 25% und in Abbildung 17(a) schon bei einer Kompressionsrate von 50% erreicht. Diese Unterschiede des PSNR zur subjektiven Wahrnehmung untermauern, dass Verzerrungsmaße und Qualitätsmaße nicht als objektive Mittel zu Bewertung der Bildqualität herangezogen werden sollten. Eine Bewertung durch Menschen kann nicht ersetzt werden, aber sie können Anhaltspunkte darüber liefern, ob sich die Qualität unverhältnismäßig zur Kompressionsrate entwickelt.
Dies war bspw. der Fall in einer alternativen Implementation, die parallel zu der hier vorgestellten getestet wurde. Der Grund für die zweite Implementation war es, Die Herangehensweise der Standard JPEG Kompression während der Quantisierung zu simulieren, um Vergleichswerte zu erhalten.
Da GLSL keine Festkommzahlen als Variablentypen erlaubt, wurden Festkommazahlen folgendermaßen simuliert:

```
1 float x = 0.123456789f;
2 x *= 10000000;   // x = 1234567.89
3 x = round(x);    // x = 1234568.00
4 x /= 10000000;   // x = 0.1234568
```

Die Kompressionsrate war nahezu identisch zu der Kompressionsrate der hier vorgestellten Implementation. Die Werte für MSE sowie PSNR entwickelten sich bereits bei einer Kompression von 50% deutlich schlechter als die Implementation mit Fließkommazahlen, was durch den subjektiven Eindruck bestätigt wurde. Nachfolgend ein Bild der Festkommasimulation bei 10% Kompressionsrate, 3.12 Bitrate, einem MSE von 1031 und einem PSNR von 18 dB, zu sehen in Abbildung 18. Ein Direktvergleich ist in Abbildung 19 zu sehen.

Abbildung 18: Mit simulierter Festkommaquantisierung komprimiertes Bild

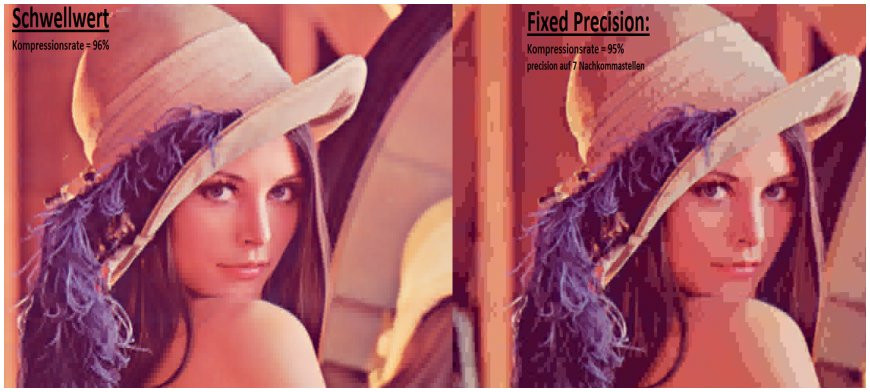


Abbildung 19: Vergleich der Bildqualität bei Kompression mit Festkommazahlen und Fließkommazahlen

4.2 Echtzeitfähigkeit und Performance

Konfiguration des Testsystems

- Grafikkarte: Nvidia GTX 570
- Prozessor: Intel i7-2600K
- Arbeitsspeicher: 8 GB
- Betriebssystem: Windows 7 Professional 64-Bit

Die Implementation soll unabhängig von der Bildqualität auf Effizienz getestet werden. Zu diesem Zweck werden alle Operationen der Kompression einzeln untersucht. Die gemessenen Zeiten werden über 300 Durchläufe protokolliert und gemittelt.
Während den Messungen fällt auf, dass die Texturgröße keinen Einfluss auf die Berechnungszeit der Compute Shader nimmt. Die kleinsten Texturen (512x512 Pixel) werden genau so schnell verarbeitet, wie die größten (1920x1080 Pixel).
Der Durchschnitt pro Compute Shader beträgt dabei 0,51636 μs. Dabei sollte beachtet werden, dass es ebenfalls Zeit benötigt die Shader in OpenGL zu binden. Es wäre also möglich die Rechenzeit dahingehend zu optimieren, dass man die Shader in einem Compute Shader zusammenfasst. Die Zeiten werden inklusive Binden der Shader gemessen.
Ebenfalls fällt auf, dass die Operationen, die nicht parallelisierbar sind, die meiste Rechenzeit in Anspruch nehmen - die Übertragung der Texturen, sowie die Lauflängenkodierung und die Lauflängendekomprimierung.
Betrachtet man die Kompression getrennt von der Dekompression, dann machen die Compute Shader im gesamten Kompressionsvorgang zwischen 0,057% (bei 512x512) und 0,009% (bei 1920x1080) der Berechnungszeit aus.

Operation	ØBerechnungszeit 512x512	1280x720	1920x1080
Umrechnung der Farbräume	ca. 2,408 μs	siehe 512x512	siehe 512x512
Kompression der Farbkanäle	ca. 0,502 μs	siehe 512x512	siehe 512x512
horizontale DCT	ca. 0,498 μs	siehe 512x512	siehe 512x512
vertikale DCT	ca. 0,509 μs	siehe 512x512	siehe 512x512
Quantisierung	ca. 0,450 μs	siehe 512x512	siehe 512x512
Zig-Zag-Scan	ca. 0,497 μs	siehe 512x512	siehe 512x512
Übertragung der Textur von GPU	ca. 8,431 ms	ca. 22,805 ms	ca. 55,726 ms
Lauflängen-kodierung	ca. 0,085 ms	ca. 0,226 ms	ca. 0,566 ms
Lauflängen-dekodierung	ca. 0,709 ms	ca. 2,605 ms	ca. 6,091 ms
Übertragen der Textur auf GPU	ca. 1,09 ms	ca. 3,30 ms	ca. 8,09 ms
Inverser Zig-Zag-Scan	ca. 0,516 μs	siehe 512x512	siehe 512x512
Dequantisierung	ca. 0,555 μs	siehe 512x512	siehe 512x512
Vertikale iDCT	ca. 0,784 μs	siehe 512x512	siehe 512x512
Horizontale iDCT	ca. 0,421 μs	siehe 512x512	siehe 512x512
Farbumrechnung	ca. 0,944 μs	siehe 512x512	siehe 512x512

Die gesamte Kompressionszeit beträgt:

- 8,520864 ms für Texturen einer Größe von 512x512 Pixeln
- 23,035864 ms für Texturen einer Größe von 1280x720 Pixeln
- 56,296864 ms für Texturen einer Größe von 1920x1080 Pixeln

Dagegen beträgt die gesamte Dekompressionszeit:

- 1,802216 ms für Texturen einer Größe von 512x512 Pixeln
- 5,908216 ms für Texturen einer Größe von 1280x720 Pixeln
- 14,184216 ms für Texturen einer Größe von 1920x1080 Pixeln

Es wird ebenfalls getestet ob eine Echtzeitfähigkeit möglich ist, wenn Coder und Decoder des Codecs auf getrennten Systemen ausgeführt werden. Der Versuchsaufbau ist in Kompression und Dekompression unterteilt.

Die Animation eines sich drehenden und bewegenden Würfels wird gerendert und pro Frame wird die Ausgabetextur an die implementierte Kompressionspipeline übergeben. Jedes Mal, wenn der letzte Schritt des Coders fertiggestellt ist, werden die komprimierten Texturen auf die Festplatte geschrieben. Dieser Vorgang wird so oft wiederholt, bis exakt 10 Sekunden Animation als komprimierte Textur auf der Fesplatte gespeichert sind.

Anschließend wird das Programm beendet und der Decoder beginnt mit seiner Arbeit. Bevor die Lauflängendekodierung startet, wird der erste Satz komprimierter Texturen eingelesen. Dieser Satz besteht aus:

- 1 Helligkeitstextur, komprimiert auf durchschnittlich 10%.
- 2 Farbtexturen, komprimiert auf exakt 50%.

Anschließend beginnt die normale Dekompressionspipeline.

Die Kompression kann auf dem Testsystem mit 30 Bildern pro Sekunde bei einer Auflösung von 1280x720 Pixeln ausgeführt werden. Die Dekompression läuft konstant mit 60 Bildern pro Sekunde, weswegen die Animation auf Decoderseite doppelt so schnell abläuft.
Insgesamt fallen während der 10 sekündigen Animation 300 komprimierte Datensätze an, welche einen Speicherplatz von 1100 MB in Anspruch nehmen. Bei einer Datenübertragungsrate von 110 MB pro Sekunde, könnte die Animation mit 30 Bildern pro Sekunde auf Decoderseite ablaufen.

5 Fazit & Ausblick

In dieser Arbeit wurde ein gängiges Verfahren der Einzelbildkompression mit Hilfe von OpenGL Compute Shadern auf Grafikkarten umgesetzt. Fast alle Teile der Kompression, sowie der Dekompression wurden durch Parallelisierung hardware-beschleunigt, sodass echtzeitfähige Kompression zur Videoübertragung möglich gemacht wurde. Durch die unabhängige Bearbeitung der 8x8 Blöcke, während der diskreten Kosinustransformation und den strukturellen Aufbau der Work Groups der Compute Shader, profitiert die Kompression sehr von einer Parallelisierung.
Die quantitativen Werte aus Kapitel 4 zeigen, dass es durchaus im Bereich des Möglichen liegt, 30 Bilder pro Sekunde auf beiden Seiten des Codecs zu erreichen, und somit einen flüssigen Bewegungseindruck zu erzeugen.
Die qualitativen Ergebnisse der Kompression entsprechen denen komerzieller Kompressionstechniken und weisen sogar, auf Grund der Verwendung von Gleitkommazahlen im Bereich der Rekonstruktionsfehler, bessere Ergebnisse auf.
Im Bereich der Performance liegt noch deutliches Optimierungspotenzial. Darauf soll im Ausblick in Kapitel 5.1 bis 5.3 eingegangen werden.

5.1 Parallelisierbarkeit des Run Length Encodings

Die Ergebnisse aus dem vorangegangenen Kapitel haben deutlich gezeigt, dass die Lauflängenkodierung nach der Texturübertragung den teuersten Schritt in der Kompressionspipeline darstellt. Während das Übertragen von Daten zwischen Graffikkartenspeicher und Arbeitsspeicher nur durch das Aufrüsten der Hardware beschleunigt werden kann, existieren bereits Implementationen des RLE, welche teilweise parallelisiert wurden [Rut11]. Jedoch ist der Algorithmus der Lauflängenkodierung schon durch sein reines Prinzip der sequenzielle Teil, des in Kapitel 2.1.1 beschriebenen Amdahlschen Gesetzes.
Während den durchgeführten Tests ist ebenfalls aufgefallen, dass nur einer von vier zur Verfügung stehenden physischen Kernen die Rechenlast zu mehr als 80% schulterte. Es wäre folglich bereits ein großer Leistungszuwachs zu erwarten, wenn man die Arbeit gleichmäßiger auf die zur Verfügung stehenden Kerne verteilen würde.
Auch wenn vermutlich die von JPEG genutzte Huffman Kodierung bereits effizienter implementiert werden könnte als ein so inhärent sequentieller Algorithmus wie die Lauflängenkodierung, wäre es interessant eine parallelisierte Version des RLE mit anderen verlustlosen Kompressionsalgorithmen zu vergleichen. Der Grund, warum die RLE für den verlustlosen Teil der Kompression gewählt wurde, war, dass durch die Umsortierung des Zig-Zag-Scans eine höhere Ausnutzung der Redundanz erwartet wurde, was bessere Bitraten versprach. Es stellte sich jedoch heraus, dass die Bitraten mit dem des JPEG2000 Standards nicht mithalten können. Eine Implementation der Lauflängenkodierung war leider mit Compute Shadern nicht möglich. Diese Problematik soll in Kapitel 5.2 näher beschrieben werden. Mögliche Lösungsstrategien werden in Kapitel 5.3 vorgestellt. Aber genau aus diesem Grund wurde die gesamte Kompressionspipeline nur auf dem Helligkeitskanal angewandt. Es wäre zu unperformant dies auch für die beiden Farbkanäle durchzuführen. Es gibt spezielle Quantisierungstabellen für die Chrominanzkanäle. Aber um die Farbkanäle ebenfalls mit der diskreten Kosinustransformation sinnvoll nutzen zu können, muss eine Optimierung des verlustlosen Kompressionsteils durchgeführt werden.

5.2 Grenzen von OpenGL Compute Shadern

In diesem Kapitel soll kurz ausarbeitet werden, warum die Wahl der OpenGL Compute Shader nicht optimal für alle Herausforderungen dieser Arbeit war. Die Compute Shader, in der jetzigen Version 4.5, besitzen nicht die Möglichkeit Output mit variabler Länge zu generieren. Dies ist ein Nachteil, den vergleichbare API's zur GPGPU-Programmierung nicht aufweisen. Dadurch war es nicht möglich, das RLE mit Compute Shadern zu implementieren. Jedoch erweisen sich Compute Shader als außerordentlich effizient für die Teile

der implementierten Kompression, welche einen fest definierten Output generiert. Die Möglichkeit auf Texturen zu arbeiten und einfach die Work Group Größe an die Ein- oder Ausgabetextur anzupassen ist ebenfalls ein Vorteil, wenn die Eingabedaten der Kompression ohnehin eine Textur sind. Es lässt sich festhalten, dass sich Compute Shader für die meisten Tätigkeiten der Bildverarbeitung sehr gut eignen.

5.3 Interframe Kompression & andere API's

Auf Grund der Unzulänglichkeiten der OpenGL Compute Shader liegt die Überlegung nahe andere API's zur GPGPU-Programmierung zu nutzen. Auch wenn Compute Shader großartige Berechnungszeiten aufweisen (nichtmal eine Microsekunde während den Tests), lässt sich keine variable Ausgabe mit ihnen erzeugen. Es wurden bereits Variable Length Coding Argorithmen auf der GPU implementiert, die durch die Parallelisierung einen Geschwindigkeitszuwachs von 35x erreichen[App09]. Diese und andere Implementationen nutzen häufig die Nvidia CUDA API. Anstatt die Compute Shader Implementation komplett neu in einer anderen API wie CUDA zu implementieren, könnte man den verlustfreien Kompressionspart in CUDA schreiben und mit dieser Implementation kombinieren. OpenGL Programme können ohne Weiteres mit CUDA oder OpenCL kombiniert werden.

Eine zusätzliche Erweiterung dieser Implementation könnte die sog. Interframe-Kompression sein. Gerade das Varialble Length Coding scheint für diese Technik gute Ergebnisse erzielen zu können. Die Idee ist, neue Bilder nur durch die Veränderungen zum vorangegangen Bild zu kodieren. Besitzt ein Pixel bspw. den Wert 0.6 und im nächsten Bild den Wert 0.7, müsste man bloß eine Veränderung von +0.1 abspeichern. Eine solche Erweiterung, könnte die Kompressionsrate zusätzlich steigern.

Literatur

[App09] Appel, Arthur: Parallel Variable-Length Encoding on GPGPUs, University of Stuttgart, 2009. – http://tesla.rcub.bg.ac.rs/~taucet/docs/papers/PAVLE-AnaBalevic09.pdf zuletzt eingesehen am 02.04.2015

[Gro15] Group, Khronos: *OpenGL Specification V4.4*. 2015. – https://www.opengl.org/registry/doc/glspec44.core.pdf" Eingesehen am 08.03.2015

[HM08] Hill, Mark D. ; Marty, Michael R.: Amdahl's Law in the Multicore Era, IEEE Computer, 2008. – http://research.cs.wisc.edu/multifacet/papers/tr1593_amdahl_multicore.pdf zuletzt eingesehen am 02.04.2015

[Mal] Mallot, Hanspeter A.: *Sehen und die Verarbeitung visueller Information. Eine Einführung.* Vieweg Verlagsgesellschaft

[Nel] Nelson, Mark: *The Data Compression Book*. DG Books Worldwide, Inc.

[NVI14] NVIDIA: *CUDA C Programming Guide*. 2014. – http://docs.nvidia.com/cuda/cuda-c-programming-guide/" Eingesehen am 04.09.2014

[Rut11] Rutter, Ruth: Run-length Encoding on Graphics Hardware, University of Alaska, 2011. – https://www.cs.uaf.edu/media/filer_public/2013/08/27/ms_cs_ruth_rutter.pdf zuletzt eingesehen am 02.04.2015

[Say] Sayood, Khalid: *Introduction to Data Compression*. Morgan Kaufmann

[Shr13] Shreiner, Kessenich Licea-Kane Sellers: *OpenGL Programming Guide*. Addison Wesley, 2013. – Eighth Edition

[Str] Strutz, Tilo: *Bilddatenkompression Grundlagen, Codierung, Wavelets, JPEG, MPEG, H.264*. ViehwegVieweg + Teubner

[WBP] William B. Pennebaker, Joan L. M.: *JPEG: Still Image Data Compression Standard*. Springer