h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

Hochschule Darmstadt

– Fachbereich Informatik –

Erweiterung eines fraktalen Videokompressionsverfahrens durch die Referenz derselben Fraktale

Abschlussarbeit zur Erlangung des akademischen Grades

Bachelor of Science (B.Sc.)

vorgelegt von

Xaver Himmelsbach

Referent	:	Prof. Dr. Oliver Weissmann
Korreferent	:	Björn Bär

ERKLÄRUNG

Ich versichere hiermit, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die im Literaturverzeichnis angegebenen Quellen benutzt habe.

Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder noch nicht veröffentlichten Quellen entnommen sind, sind als solche kenntlich gemacht.

Die Zeichnungen oder Abbildungen in dieser Arbeit sind von mir selbst erstellt worden oder mit einem entsprechenden Quellennachweis versehen.

Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

Darmstadt, 19. April 2021

Xaver Himmelsbach

ABSTRACT

Audiovisual media are compressed with so-called codecs to facilitate transmission within a network. This process can be lossy or lossless. The coding procedures used for this purpose are constantly being developed further. The codecs used today often work with compression methods such as the Discrete Cosine Transform, which stores the complete image content as parameters of mathematical functions. However, another interesting method has been largely ignored by the industry: Fractal coding.

Fractal image codecs exploit the self-similarity of images in order to only have to store small image areas, which are called domain images. Using those, the rest of the image can be reconstructed through affine transformations. Although this approach allows for high-quality compression, it has not been widely used, mainly because of the large amount of computation required during encoding.

Since today's processors and graphics cards have significantly better performance, it makes sense to reconsider the potential of fractal encoding of images and videos.

In this paper, after an introduction to classical and fractal image and video coding, an existing fractal video codec is extended. FIASCO, the codec used as a basis, supports fractal frame coding and block-based motion compensation. The quality of the motion compensation will be optimised by referencing the same domain images in several frames. For this purpose, a sequence of video frames is first combined in an image file and encoded with FIASCO, whereby the domain images used are identical in all frames of this image group. During decoding, the individual images are separated again and stored one after the other as frames of a video.

The speed and quality of the codec extended in this way is compared with the widely used codecs H.264 and VP9. The potential of using fractal compression in the network is assessed and evaluated on the basis of the preceding measurements. The investigation result is that the extension developed in the work does not offer any direct advantages over the established codecs. However, especially hybrid approaches using classical and fractal coding seem to be promising for further research.

ZUSAMMENFASSUNG

Audiovisuelle Medien werden mit sogenannten Codecs komprimiert, um die Übertragung in einem Netzwerk zu erleichtern. Dieser Vorgang kann sowohl verlustbehaftet als auch verlustfrei ablaufen. Die zu diesem Zweck genutzten Coding-Verfahren werden stetig weiterentwickelt. Während die heute eingesetzten Codecs oft mit Kompressionsverfahren wie der Diskreten Cosinustransformation arbeiten, die den kompletten Bildinhalt als Parameter mathematischer Funktionen speichern, ist ein weiteres interessantes Verfahren von der Industrie größtenteils ignoriert worden: Das fraktale Coding.

Fraktale Bildcodecs nutzen die Selbstähnlichkeit von Bildern aus, um nur kleine Bildbereiche, sogenannte Domain-Images, speichern zu müssen. Aus diesen lässt sich wiederum der Rest des Bildes durch affine Transformationen rekonstruieren. Obwohl dieser Ansatz eine qualitativ hochwertige Kompression ermöglicht, wurde er, vor allem aufgrund des großen Rechenaufwands während des Encodings, nicht in großem Maße angewandt.

Da heutige Prozessoren und Grafikkarten über bedeutend bessere Leistung verfügen, bietet es sich an, das Potenzial fraktaler Codierung von Bildern und Videos erneut zu untersuchen.

In dieser Arbeit wird, nach einer Einführung in die klassische und die fraktale Bild- und Videocodierung, ein bestehendes fraktales Videocodec erweitert. Bei diesem Codec handelt es sich um FIASCO, das eine fraktale Einzelbildcodierung und eine blockbasierte Bewegungskompensation unterstützt. Die Qualität der Bewegungskompensation soll durch die Referenz derselben Domain-Images in mehreren Einzelbildern optimiert werden. Hierfür wird zunächst eine Folge von Video-Frames in einer Bilddatei zusammengefasst und mit FIASCO codiert, wodurch die verwendeten Domain-Images in allen Einzelbildern dieser Bildgruppe identisch sind. Beim Decoding werden die einzelnen Bilder wieder separiert und nacheinander als Frames eines Videos gespeichert.

Die Geschwindigkeit und Qualität des so erweiterten Codecs wird mit den etablierten Codecs H.264 und VP9 verglichen. Das Potenzial des Einsatzes fraktaler Kompression im Netz wird anhand dieser Messungen eingeschätzt und evaluiert. Das Ergebnis der Untersuchungen ist, dass die in der Arbeit entwickelte Erweiterung keine direkten Vorteile gegenüber den etablierten Codecs bietet. Vor allem hybride Ansätze, die klassisches und fraktales Coding einsetzen, scheinen allerdings vielversprechend für weitere Forschung zu sein.

INHALTSVERZEICHNIS

ABBILDUNGSVERZEICHNIS

ABKÜRZUNGSVERZEICHNIS

CRF Constant Rate Factor

DCT Discrete Cosine Transform

FIASCO Fractal Image And Sequence Codec

GOP Group of Pictures

PNM Portable Any Map

WFA Weighted Finite Automaton

1 EINFÜHRUNG

Die Geschwindigkeit von Internetanbindungen ist in den vergangenen Jahren enorm gestiegen. Doch auch mit Highspeed-Downloads nehmen Bild- und Videodateien, neben dynamischen Skripten und Anfragen an Webschnittstellen, einen Großteil der übertragenen Datenmenge im Netz ein.

Ein Video besteht in unkomprimierter Form aus einer Folge von Einzelbildern. Diese Einzelbilder sind ihrerseits aus mehreren zweidimensionalen Matrizen aufgebaut, die Farb- und Helligkeitswerte der einzelnen Bildpunkte speichern.[1] Würde man eine so aufgebaute Videodatei in einer Auflösung von 1920×1080 Pixeln mit drei Farbkanälen mit einer Farbtiefe von acht Bit und einer Framerate von 25 Bildern pro Sekunde über ein Netzwerk schicken, würde ein Verkehr von etwa 155 Megabyte pro Sekunde Videomaterial anfallen. [2] Bedenkt man, dass die durchschnittliche Breitband-Downloadgeschwindigkeit in Deutschland im Dezember 2020 bei 113.19 Megabit pro Sekunde lag[3], was umgerechnet 14,15 Megabyte pro Sekunde entspricht, zeigt sich, dass es für den durchschnittlichen deutschen Internetnutzer nicht möglich wäre, ein konventionelles Full-HD Video flüssig zu streamen. In der Praxis ist es bei Streaming-Anbietern wie YouTube, Netflix und Co. allerdings möglich, Videos in dieser und höheren Auflösungen mit vergleichbaren Anbindungen ohne Aussetzer abzuspielen. Diese Dateneinsparung wird durch den Einsatz von Codecs erreicht.

Im Laufe dieser Arbeit wird ein bestehendes Bild- und Video-Codec, das seine Kompression durch den Einsatz und die Referenz sogenannter Fraktale erreicht, weiterentwickelt. Des Weiteren wird diese Weiterentwicklung auf ihr Potenzial bezüglich des Einsatzes zum Versenden von Video-Dateien in Netzwerken untersucht.

1 16, S. 73.
2 $1920 * 1080 * 3 * 8$ Bit $* 25 = 1.244.160.000$ Bit $= 155.520.000$ Byte
3 20.

2 VIDEOCODIERUNG

2.1 CODIERUNG

Eine Codierung bildet die Zeichen eines Eingangsalphabets auf die Zeichen eines Ausgangsalphabets ab. Im Bereich der Datenkompression wird durch den Einsatz einer Codierung das Ziel verfolgt, die Größe des rohen Datenstroms einer Datei durch die Abbildung der Daten auf ein neues Alphabet zu reduzieren. Erreicht wird dies mithilfe eines Codecs, was eine Wortkomposition der beiden Hauptbestandteile des Coding-Prozesses ist. Diese sind der Encoder und der Decoder.[1]

2.1.1 *Encoder*

Der Encoder ist ein Programm, das einen eingehenden Datenstrom codiert und somit komprimiert.[2] Dies wird durch die Anwendung unterschiedlicher Source-Coding-Verfahren, wie Entropie-Coding, Quantisierung, Vorhersage oder lineare Transformation, erreicht. Die angewendeten Verfahren können entsprechend der Datenstrukturen, auf die ein Encoder ausgelegt ist, variieren.[3]

Encoder sind in Betriebssystemen sowie in anderen Anwendungen und Programmen, zum Beispiel in Bild- und Videobearbeitungssuites, wo sie den Export in verschiedene Dateiformate ermöglichen[4], oder in Multimediaframeworks, wie FFmpeg[5], eingebettet. Gängige Codecs können zudem spezielle Hardwarechips[6] oder die Architektur der Grafikkarte des Systems, auf dem sie ausgeführt werden, ausnutzen und so schneller als reine Softwarecodecs, die nur auf dem Prozessor ausgeführt werden, encoden.[7] Durch die Festlegung auf eine spezielle Architektur sind Hardware-Encoder allerdings auf bestimmte Qualitätseinstellungen limitiert und somit weniger konfigurierbar als Software-Encoder.

2.1.2 *Decoder*

Der Decoder kehrt die durch den Encoder ausgeführte Komprimierung eines Datenstroms um. Die decodeten Daten können daraufhin auf ein Speichermedium geschrieben, von anderen Programmen weiter verarbeitet oder

1 16, S. 5.
2 16, S. 4.
3 16, S. 75 f.
4 1.
5 4.
6 2, S. 549 ff.
7 12.

einem Endnutzer angezeigt werden.[8] In den meisten Codecs liegt ein Großteil der Implementationskomplexität im Encoder, während das Wiederherstellen der Daten im Decoder mit deutlich weniger Aufwand verbunden ist. Die Entscheidung für diese Verteilung des Ressourceneinsatzes ist für einen Großteil der Gebiete, auf denen Codecs zum Einsatz kommen, sinnvoll. Den Empfängern, die den Datenstrom dekomprimieren, stehen meist weniger Ressourcen zur Verfügung als den Absendern, besonders bei der Kommunikation zwischen Clients und Servern im Internet.[9]

Auch Decoder werden mit Betriebssystemen, Verarbeitungsprogrammen oder Multimediaframeworks mitgeliefert, wobei sie vor allem zum Import und zur Anzeige diverser Dateiformate eingesetzt werden. Parallel zu Encodern können auch sie von dedizierter Hardware profitieren.[10]

2.1.3 *Verlustbehaftung*

Die von Encodern angewendeten Source-Coding-Verfahren unterscheiden sich grundsätzlich in ihrer Verlustbehaftung.

Verlustfreie Verfahren erlauben die exakte Rekonstruktion des ursprünglichen Datenstroms durch den Decoder aus einem codierten Datenfluss. Kompression kann in diesem Fall nur durch das Ausnutzen von Abhängigkeiten in den Daten oder aufgrund statistischer Eigenschaften erfolgen.[11,12]

Verlustbehaftete Verfahren können den ursprünglichen Datenstrom nicht eins zu eins wiederherstellen und sind somit durch einen unumkehrbaren Informationsverlust charakterisiert. Dadurch ist es aber möglich, Details zu entfernen, die für den Betrachter irrelevant sind, und so eine weit bessere Kompressionsrate als bei verlustfrei komprimierten Dateien zu erzielen. Verlustbehaftete Komprimierung wird vor allem für Audio-, Video- und Bildinhalte eingesetzt, da menschliche Betrachter den Informationsverlust in diesen Medien gut kompensieren können.[13,14]

2.2 VIDEOKOMPRIMIERUNG

Beim Encoden von digitalen Videodaten, die grundsätzlich, im Gegensatz zu analog encodeten Videos, aus einer Folge vollständiger Einzelbilder, auch Frames genannt, bestehen, gibt es zwei Möglichkeiten zur Kompression. Zum einen können die einzelnen Videoframes separat komprimiert werden, zum anderen können Redundanzen zwischen verschiedenen Frames ausgenutzt werden.

8 16, S. 5.
9 16, S. 75.
10 17, S. 129 ff.
11 14, S. 6.
12 15, S. 20 f.
13 14, S. 6.
14 15, S. 22 ff.

2.2.1 *Kompression der Einzelbilder*

Die einzelnen Frames, aus denen ein unkomprimiertes Video aufgebaut ist, lassen sich mit herkömmlichen Source-Coding-Verfahren und speziell auf Einzelbilder angepassten Techniken codieren, die genauso beim Encoden einer einzelnen Bilddatei angewendet werden können.

2.2.1.1 *Farbunterabtastung*

Da das menschliche Auge auf Helligkeitsunterschiede in gleichfarbigen Bildbereichen sehr viel sensibler reagiert als auf Farbunterschiede in ähnlich hellen Bildbereichen, ist es möglich, die Farbqualität des Bildes bei gleichbleibender Luminanzqualität ohne große Qualitätseinbuße für den Betrachter zu reduzieren.[15]

Dieser Vorgang wird durch die Farbunterabtastung, im Englischen Chroma Subsampling, realisiert. Um die Farbunterabtastung anwenden zu können, muss das Eingabebild zunächst in einen Farbraum gebracht werden, der Helligkeit und Farbe in eigene Kanäle trennt. Dies ist im RGB-Farbraum, in dem unkomprimierte Einzelbilder oft dargestellt werden, nicht der Fall. Als Standardmodell zur Farbunterabtastung hat sich Y'CbCr herausgestellt[16], während theoretisch auch Modelle wie HSV oder YUV verwendet werden könnten. Im Y'CbCr-Modell wird der Y-Kanal für Helligkeitsinformationen genutzt, während der Cb-Kanal ein gelb-blaues Differentialsignal und der Cr-Kanal ein rot-grünes Differentialsignal beinhalten.[17] Diese Aufteilung entspricht in etwa dem menschlichen Sehen mit luminanzempfindlichen Stäbchen und zwei Arten von farbempfindlichen Zapfen im Auge.[18]

Die beiden Farbkanäle können nun in einer geringeren Auflösung als der Helligkeitskanal gespeichert werden. Häufig verwendete Formate umfassen Y'CbCr 4:4:4, in dem die Farbkomponenten nicht komprimiert werden, Y'CbCr 4:2:2, wobei die Farbkomponenten die gleiche vertikale Auflösung wie die Helligkeitskomponente haben, aber horizontal um den Faktor 2 gestaucht sind oder Y'CbCr 4:2:0, in dem die Farbkomponenten horizontal und vertikal um den Faktor 2 gestaucht werden.[19,20] So kann unter Verwendung von Y'CbCr 4:2:0 beim Encoden eines Bildes eine Kompressionsrate von 50% erreicht werden.[21] Aufgrund der Auflösungsreduktion der Farbkanäle handelt es hierbei sich um ein verlustbehaftetes Kompressionsverfahren.

15 16, S. 56.
16 16, S. 56.
17 16, S. 55.
18 16, S. 22.
19 16, S. 56 f.
20 15, S. 11.
21 Original aufgelöster Y-Kanal + 2 in X- und Y-Richtung um den Faktor 2 gestauchte Farbkanäle: $1 * \frac{1}{3} + \frac{1}{4} * \frac{1}{3} + \frac{1}{4} * \frac{1}{3} = \frac{1}{2}$

2.2.1.2 *DCT und Quantisierung*

Der Grundgedanke der Discrete Cosine Transform (DCT) ist das Transformieren der Pixel eines Bildes in einen Frequenzbereich, in dem hochfrequente, für den Betrachter irrelevante Details herausgefiltert werden können.[22]

Zu diesem Zweck wird das Eingabebild in gleich große quadratische Blöcke aufgeteilt. Auf jeden dieser Blöcke kann jetzt die DCT angewendet werden. Dabei wird ein Block als lineare Kombination aus waagrecht und senkrecht orientierten Cosinus-Funktionen gespeichert.[23] Es stehen genau so viele Cosinus-Funktionen zur Verfügung, wie der Block Pixel beinhaltet. Mit zunehmendem X-Index nimmt die Frequenz der senkrechten Cosinus-Funktion zu, mit zunehmendem Y-Index die der waagrechten. Bei allen X- und Y-Indizes größer Null werden die waagrechten und senkrechten Funktionen an der entsprechenden Stelle in der Matrix kombiniert. Die Koeffizienten für die Cosinus-Funktionen werden anschließend in einer Matrix gespeichert, deren X- und Y-Länge der des Bildes in Pixeln entspricht. Die Koeffizienten in der unteren rechten Ecke der Matrix stellen hochfrequente Bilddetails dar, deren Vollständigkeit zur Qualität des codierten Bildes nur einen geringen Beitrag leistet. Die Koeffizienten in der oberen linken Ecke repräsentieren dagegen das niedrigfrequente Gros des codierten Bildes, dessen Entfernung massive Qualitätseinbußen zur Folge hätte.[24]

Um Kompression zu erreichen, wird diese Koeffizientenmatrix quantisiert. Bei der Quantisierung wird jeder Koeffizient durch einen Quantisierer-Wert aus einer gleichgroßen Quantisierungsmatrix geteilt. Diese Matrix wird von jedem Encoder selbst vorgegeben. Im Allgemeinen werden die niedrigfrequenten Koeffizienten mit deutlich kleineren Werten als die hochfrequenten Koeffizienten quantisiert[25], wodurch sie deutlich weniger Genauigkeit verlieren. Die hochfrequenten Koeffizienten werden dagegen in den meisten Fällen zu einem Wert nahe Null quantisiert. Da bei der Quantisierung Informationen verloren gehen, ist sie, wie die in 2.2.1.1 beschriebene Methode der Farbunterabtastung, eine verlustbehaftete Technik.[26]

2.2.1.3 *Entropie-Coding*

Entropie-Codes sind ein Verfahren, um die quantisierten Koeffizienten der DCT effizient zu speichern. Sie weisen den Zeichen eines Eingangsalphabets ein Codewort mit variabler Länge zu. Die Länge dieses Codeworts richtet sich nach der Häufigkeit, mit der das jeweilige Zeichen im zu codierenden Text auftaucht. Sehr häufig auftretende Zeichen erhalten kurze Codewörter, während seltenere Zeichen mit längeren Wörtern codiert werden.[27]

Eines der bekanntesten Entropie-Coding-Verfahren ist das Huffman-Coding. Hierbei wird aus dem Eingabealphabet ein Binärbaum aufgebaut, in

22 8, S. 203.
23 15, S. 32.
24 8, S. 204 f.
25 13.
26 14, S. 78.
27 10, S. 13 f.

dem in jedem Schritt die beiden seltensten Zeichen unterhalb eines neuen Knotens gesetzt werden, der die kombinierte Wahrscheinlichkeit seiner beiden Kinder als Wert erhält. Im nächsten Schritt werden die beiden Zeichen bei der Wahrscheinlichkeitssortierung durch den neuen Knoten ersetzt.[28]

Weitere Entropie-Coding-Verfahren sind unter anderem der Rice-Golomb-Code oder der angepasste Binärcode.[29]

2.2.1.4 *Weitere Coding-Verfahren*

Als weitere Techniken zum Speichern quantisierter Koeffizienten sind das arithmetische Coding und die Lauflängencodierung zu nennen.

Arithmetisches Coding eignet sich besonders, wenn nicht die Koeffizienten selbst als Zeichen, sondern ihre Binärdarstellung komprimiert werden soll. Zu diesem Zweck wird das Intervall einer Fließkommazahl zwischen 0 und 1 gespannt. Die Grenzen dieses Intervalls werden, je nach dem Eingabezeichen, so weit verengt, bis die erste Nachkommastelle in beiden Grenzwerten gleich ist. Diese Ziffer wird vom Encoder herausgeschrieben und aus den Grenzwerten entfernt, sodass sich das Intervall wieder breiter verbreitert.[30]

Die Lauflängencodierung ist eines der simpelsten Coding-Verfahren, das gleichzeitig für die Kompression von quantisierten DCT-Koeffizienten gut geeignet ist. Bei der Lauflängencodierung werden aufeinanderfolgende Wiederholungen des gleichen Zeichens durch die Anzahl der Vorkommnisse des Zeichens ersetzt. Wenn die Koeffizienten des DCT nach einem Zickzackmuster aus der Matrix ausgelesen werden, lässt sich das volle Potenzial dieser Technik nutzen, da so zuerst die Koeffizienten der oberen linken Ecke, die im Allgemeinen größer als Null sind, verarbeitet werden. Ab einem bestimmten Punkt müssen dann nur noch Nullen encodet werden.[31]

2.2.2 *Aufbau*

Eine codierte Videodatei besteht in der Regel nicht nur aus einer Aneinanderreihung komprimierter Einzelbilder. Zwischen den aufeinanderfolgenden Frames eines Videos bleiben große Bildbereiche oftmals unverändert oder werden nur verschoben. Diese Dopplungen lassen sich durch eine Strukturierung der Frames in I-, P- und B-Frames aufheben.[32]

2.2.2.1 *I-Frames*

I- bzw. Intra-Frames sind vollwertige Einzelbilder, die mit den vorangegangenen Coding-Methoden komprimiert wurden. Sie dienen als Anhalts- und Synchronisationspunkte beim Abspielen des Videos. Ein Video wird in sogenannte Groups of Pictures (GoPs) aufgeteilt. Eine GoP beginnt immer mit

28 10, S. 14.
29 10, S. 15.
30 10, S. 17.
31 10, S. 18.
32 10, S. 61.

einem I-Frame, das als Einstiegspunkt für den Decoder dient. Darauf folgt eine vom Encoder festgelegte Menge an P- und B-Frames, die sich Motion-Compensation, also die Redundanz ähnlicher Bildausschnitte in aufeinander folgenden Frames, ausgehend vom initialen I-Frame, zunutze machen. Je länger die GoPs sind, je weiter also einzelne I-Frames voneinander entfernt sind, desto auffallender werden die durch die Motion-Compensation eingebrachten Bildfehler. Dafür steigt allerdings die Kompressionsrate, da I-Frames der speicherintensivste Frame-Typ sind.[33]

2.2.2.2 *P-Frames*

P- bzw. Prediction-Frames speichern nur die Veränderungen, die sie gegenüber einem vorherigen I- oder P-Frame aufweisen, und neue Bildinhalte, die nicht aus den vorangegangenen Frames vorhergesagt werden können. Sie sind ausschließlich zu Forward-Prediction in der Lage, können also keine Inhalte aus späteren Frames referenzieren. Die Vorhersage erfolgt über Displacement-Vektoren, die die Bewegung von Pixelblöcken aus vorangegangenen Frames beschreiben. Durch das Verschieben von Inhalten aus anderen Frames ist der Einsatz von P-Frames verlustbehaftet, da dabei Informationen im vorhergesagten Frame verloren gehen können. Blöcke, zu denen kein Displacement-Vektor gefunden wird, werden wie in einem I-Frame gespeichert.[34]

2.2.2.3 *B-Frames*

B- bzw. Bidirectional-Prediction-Frames können zusätzlich zur Forward-Prediction auch Inhalte aus nachfolgenden Frames referenzieren und sind aus diesem Grund zur Backward-Prediction in der Lage. Auch in diesem Fall kommen Displacement-Vektoren und explizit gespeicherte, nicht vorhersagbare Bildinhalte zum Einsatz. B-Frames sind vor allem zum Minimieren von Folgefehlern der Motion-Compensation aus der Anwendung mehrerer P-Frames nacheinander nützlich und ermöglichen einen sanften Übergang zwischen ihnen. Sie sind ebenso wie P-Frames verlustbehaftet, nehmen aber noch weniger Speicherplatz ein. Jedoch ist ihre Berechnung beim En- und Decoden im Vergleich mit den anderen Frame-Arten am aufwendigsten.[35]

33 10, S. 61 f.
34 10, S. 62 f.
35 10, S. 62.

3 FRAKTALE BILDKOMPRESSION

3.1 FRAKTALE

Die fraktale Bildkompression hat keine große Nutzerbasis erreicht. Der Grund hierfür liegt vor allem im hohen Rechen- und Speicheraufwand, den das Encoden eines Bildes mit Fraktalen verursacht, auch wenn das Decoden dafür schnell und einfach umsetzbar ist, wie im Folgenden aufgezeigt wird.

3.1.1 *Gedankenexperiment*

Das folgende Gedankenexperiment erlaubt ein grundsätzliches Verständnis über die Anwendung von Fraktalen in der Bildkompression.

Man stelle sich einen Fotokopierer vor, der die Vorlage um die Hälfte verkleinert und dreimal, jeweils an einer anderen Position, auf die Kopie abbildet. Wird diese angefertigte Kopie direkt wieder in den Fotokopierer eingelegt, in einem sogenannten Feedback-Loop, ergibt sich mit der Zeit ein spezifisches Konvergenzbild, das unabhängig vom ursprünglichen Originalbild ist.[1]

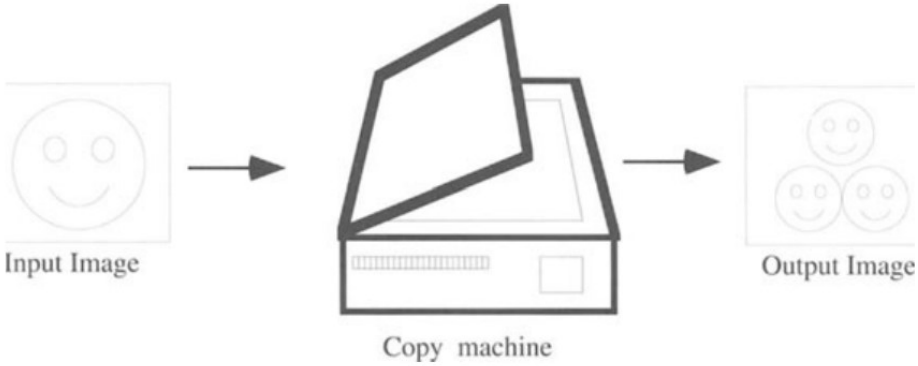

Abbildung 3.1: Schematischer Aufbau des Gedankenexperiments. [6, S. 3]

Dieses Konvergenzbild wird von der Art, wie die reduzierten Eingabebilder auf der Kopie angeordnet werden, beeinflusst. Verschiedene Transformationen führen, solange sie kontraktiv sind, zu verschiedenen Konvergenzbildern. Die Anordnung auf der Kopie wird über affine Transformationen, nach einem speziellen Schema aufgebaute Funktionen, gesteuert.[2]

Es zeigt sich, dass das Konvergenzbild, auch Attraktor genannt, unabhängig vom initialen Eingabebild ist. In diesem Fall werden affine Transforma-

1 6, S. 2.
2 6, S. 2.

tionen eingesetzt, um ein sogenanntes Sierpinski-Dreieck, eines der bekanntesten Fraktale, iterativ zu erzeugen.[3]

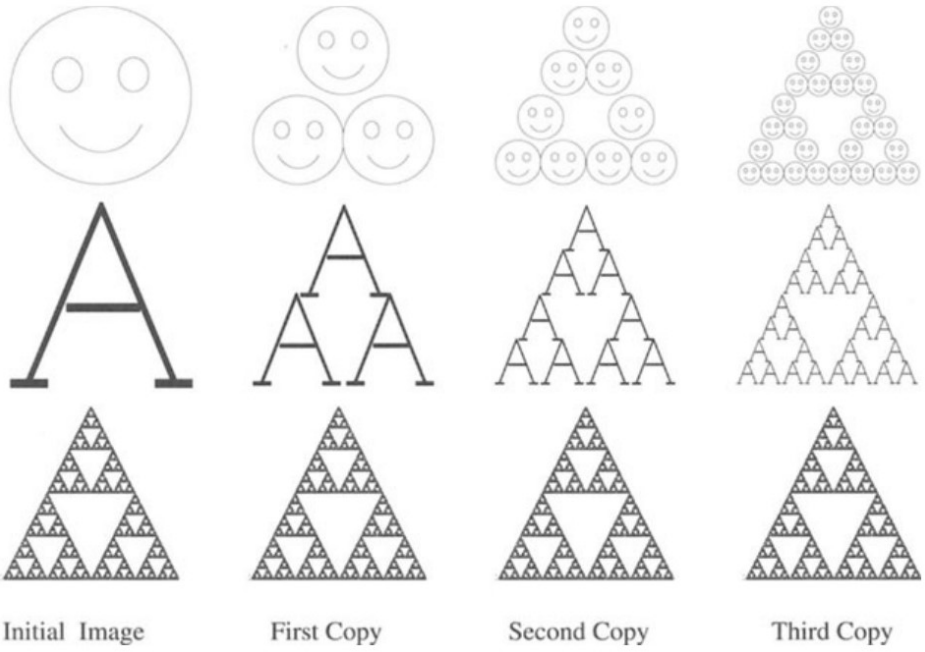

Abbildung 3.2: Attraktoren mit gleichen affinen Transformationen und unterschiedlichen Eingangsbildern. [6, S. 4]

3.1.1.1 *Affine Transformationen*

Eine affine Transformation ist eine Abbildung von einem Vektor auf einen anderen Vektor in derselben Dimension. Im Kontext dieser Arbeit werden ausschließlich zwei- und dreidimensionale Vektoren verwendet. Eine affine Transformation lässt sich schreiben als $w = Ax + b$, wobei w der transformierte Vektor, A eine $n \times n$-Matrix, x der ursprüngliche Vektor und b ein Translations-Vektor ist. Diese Formel ermöglicht die Rotation, Skalierung und Translation des Vektors x im n-dimensionalen Raum.[4]

Bei einer dreidimensionalen affinen Transformationen, wie sie im vorangegangenen Gedankenexperiment zum Einsatz kommt, stellen die X- und Y-Koordinate die Bildkoordinaten auf der Fotokopie dar, während die Z-Koordinate den Grauwert am jeweiligen Bildpunkt repräsentiert. Letztere wird durch die affinen Transformationen in diesem Beispiel nicht verändert.[5]

3.1.1.2 *Kontraktivität*

Eine Abbildung ist kontraktiv, wenn die Werte, die sie zurückgibt, um einen Faktor kleiner eins gestaucht werden und sich somit bei jeder Iteration ein-

3 6, S. 3 f.
4 6, S. 46.
5 6, S. 11.

ander annähern. Der Fixpunkt der Kontraktion, also der Punkt, auf den alle Werte zustreben, ist dabei für jede kontraktive Funktion unterschiedlich.[6]

Ist die Kontraktivität für eine Funktion nicht gegeben, ist sie für den Einsatz in der fraktalen Bildkompression nicht geeignet, da die Werte bei jeder Iteration weiter wachsen und somit ins Unendliche streben, wenn eine Streckung vorliegt oder sich nicht verändern, wenn der Stauchfaktor gleich eins ist.[7]

3.1.1.3 *Iterierte Funktionssysteme*

Ein iteriertes Funktionssystem ist eine endliche Menge kontraktiver Funktionen. Diese Funktionen bilden eine Menge an Punkten aus einem n-dimensionalen Raum wieder auf diesen Raum ab. Im Zusammenhang mit dem Gedankenexperiment ist jede Funktion, die in dieser Menge enthalten ist, eine affine Transformation, die das Eingabebild auf einen Bildbereich des Ausgabebilds abbildet.[8]

Wenn alle Funktionen des Funktionssystems kontraktiv sind, existiert für dieses eine Menge an Punkten, die Attraktor genannt wird. Wenn das iterierte Funktionssystem mit dem Attraktor als Eingabe ausgeführt wird, bringt es als Ergebnis wieder den Attraktor hervor. Der Attraktor ist der Grenzwert des iterierten Funktionssystems. Wenn das Funktionssystem also mit einer beliebigen initialen Punktemenge in einem Feedback-Loop ausgeführt wird, wird irgendwann der Attraktor erreicht. Für jedes iterierte Funktionssystem existiert genau ein Attraktor. Diese Eigenschaften sind auch als Contractive-Mapping-Fixed-Point-Theorem bekannt und werden beim fraktalen Encoden ausgenutzt.[9]

3.1.2 *Eigenschaften*

Der Begriff „Fraktal“ ist schwer fassbar und wird üblicherweise anhand der Eigenschaften, die ihn ausmachen, abgegrenzt. Die folgenden Merkmale sind maßgebliche Charakteristika zur Bestimmung der Fraktalseigenschaft.[10]

3.1.2.1 *Detail in jedem Maßstab*

Unabhängig davon, wie stark das Fraktal vergrößert wird, hat es immer dieselbe Detailfülle, da es aus unendlich vielen Kopien seiner selbst besteht.[11]

6 6, S. 2.
7 6, S. 2.
8 6, S. 6.
9 6, S. 6 f., 28 ff.
10 6, S. 25.
11 6, S. 26.

3.1.2.2 *Selbstähnlichkeit*

Da Fraktale immer aus den gleichen Bausteinen, nämlich aus Kopien ihrer selbst, aufgebaut sind, verfügen sie über Selbstähnlichkeit. Das bedeutet, dass sowohl die vollständige Menge an Bildpunkten, als auch alle Untermengen, die durch die iterierte Anwendung der Funktionen erzeugt wurden, annähernd identisch sind.[12]

3.1.2.3 *Fraktale Dimension*

Wenn die fraktale Dimension einer Menge größer als ihre topologische Dimension ist, handelt es sich bei ihr um ein Fraktal.

Der einfachste Erklärungsansatz für die fraktale Dimension ist die sogenannte Boxdimension. Dabei wird eine Menge mit Quadraten der Größe x abgedeckt. Wird nun die Größe dieser Quadrate um die Hälfte reduziert, so steigt die Anzahl der benötigten Quadrate zur Abdeckung an. Bei einer topologisch eindimensionalen Kurve geschieht dieser Anstieg um den Faktor 2^1, bei einem topologisch zweidimensionalen Quadrat um den Faktor 2^2 und bei einem topologisch dreidimensionalen Würfel um den Faktor 2^3. Diese Mengen haben also eine fraktale Dimension, die ihrer topologischen Dimension entspricht. Wendet man dieses Verfahren aber auf das fraktale Sierpinski-Dreieck an, so erhält man den Faktor $2^{\frac{log3}{log2}}$. $\frac{log3}{log2} \approx 1,585$ ist größer als die topologische Dimension des Sierpinski-Dreiecks, die 1 beträgt, da es theoretisch nach unendlicher Iteration über keine Fläche mehr verfügt.[13,14]

3.2 ANWENDUNG IN DER BILDKOMPRESSION

3.2.1 *Nötige Anpassungen*

Das mathematische Konzept der Fraktale bedarf einiger Anpassungen, um in der Bildcodierung angewendet werden zu können.

Ein typisches mit einer Kamera aufgenommenes oder an einem PC erzeugtes Bild lässt sich nicht praktikabel aus vollständigen Kopien seiner selbst aufbauen, da es nicht selbstähnlich genug ist.[15] Es ist aber möglich, das Bild in einzelne Partitionen aufzuteilen, die jeweils aus anderen partitionierten Bildausschnitten aufgebaut sind. Um dies zu erreichen, kann man den Fotokopierer aus 3.1.1 für jede affine Transformation mit einer Maske ausstatten, die entscheidet, welche Partition für die jeweilige Abbildung verwendet wird. Besteht zusätzlich noch die Möglichkeit, Kontrast, Sättigung und Helligkeit der verwendeten Partition anzupassen, ist es möglich, ein Grauwertbild durch ein bestimmtes iteriertes Funktionssystem mit dem Fo-

12 6, S. 9 f.
13 6, S. 26 f.
14 7.
15 6, S. 9.

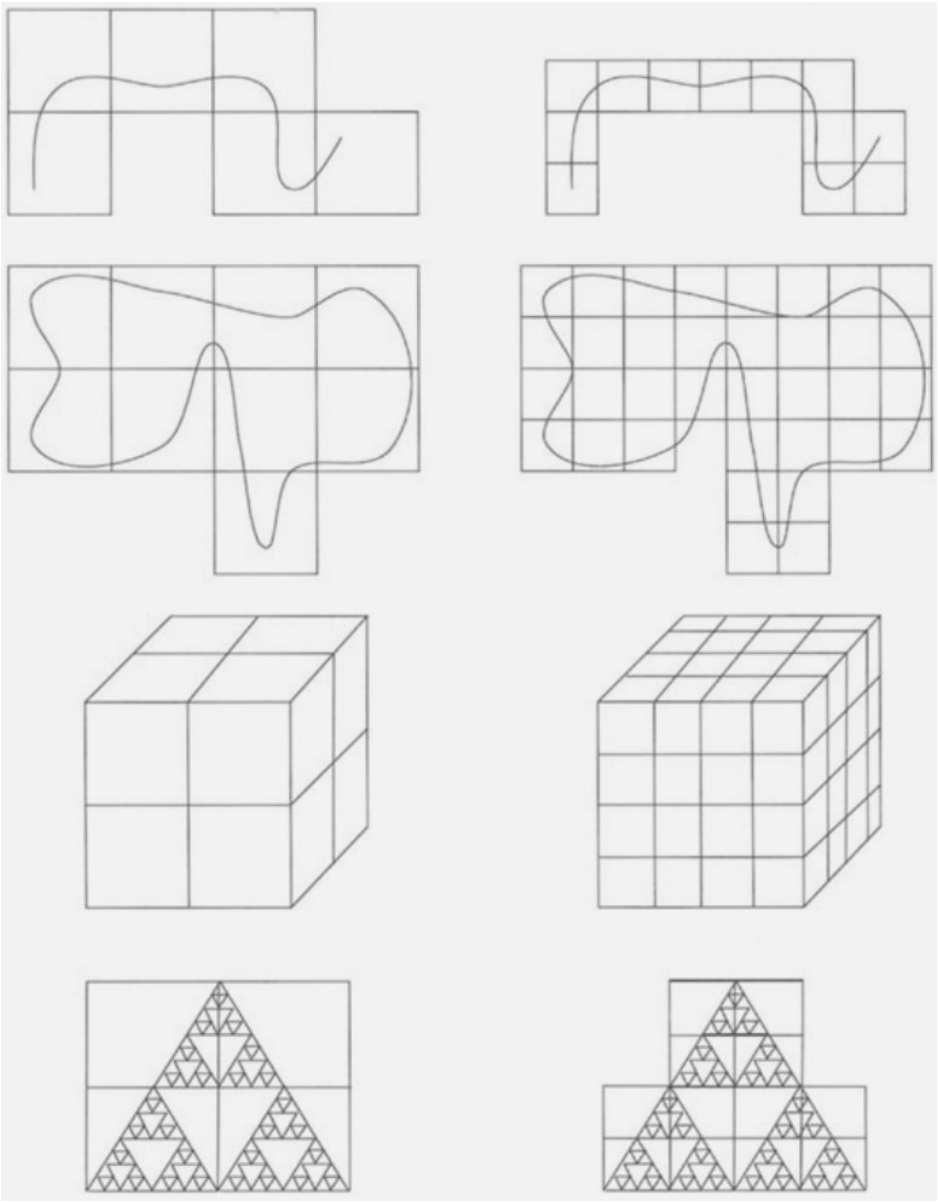

Abbildung 3.3: Wachstum der Boxdimension bei verschiedenen Formen. [6, S. 27]

tokopierer zu encoden. Zum Decoden kann ein beliebiges Eingangsbild verwendet werden.[16]

3.2.2 *Encoding*

Das fraktale Encoden eines Bildes erfolgt durch die Suche nach einer Menge an affinen Transformationen, deren Attraktor das zu codierende Bild ist. Dafür wird das Bild in einzelne Partitionen aufgeteilt, die Range-Blöcke, für deren Erzeugung jeweils die benötigten affinen Transformationen gesucht werden, die auf sogenannte Domain-Blöcke verweisen.[17] Die Partitionierung kann auf verschiedene Weisen erfolgen. Bei der Quadtree-Partitionierung wird jede Partition bei der Unterteilung in vier gleich große Unterpartitionen geteilt.[18] Die HV-Partitionierung arbeitet ähnlich, erlaubt aber die Erzeugung unterschiedlich großer Unterpartitionen, um beispielsweise die Hypotenuse eines rechtwinkligen Dreiecks in möglichst kleine Stufen abzugrenzen.[19] Weitere Partitionen teilen das Bild beispielsweise in Dreiecke, um eine größere Flexibilität im Einsetzen der Fraktale zu erreichen, da die Domain-Blöcke mit diesem Vorgehen auch um andere Winkel als um 90°, 180° und 270° rotiert werden können.[20]

Da es vom Rechenaufwand für ein gewöhnliches Bild nicht realistisch ist, einen komplett identischen Attraktor zu finden, gibt man sich beim Encoden damit zufrieden, dass der Attraktor dem zu encodenden Bild möglichst ähnlich ist. Dies wird durch das Ergebnis einer Differenzfunktion zwischen dem Originalbild und dem Attraktor überprüft, das unterhalb einer Toleranzgrenze, je nach eingestellter Qualität des Encoders, liegen muss. Aus diesem Grund sind fraktale Codecs im Regelfall verlustbehaftet.[21]

Die Koeffizienten der gefundenen affinen Transformationen werden daraufhin in eine Datei geschrieben. Dabei werden die erzeugten Daten mit Quantisierung oder den Entropie-Coding-Methoden aus 2.2.1.3 komprimiert. Die Rotation, Position und Größe der Range- und Domain-Blöcke wird ebenfalls gespeichert, wobei manche Partitionsmethoden weniger Speicherplatz benötigen. Die Quadtree-Partitionierung ermöglicht es zum Beispiel, die Position zu unterschlagen, da durch die Größe der Range-Blöcke, die Reihenfolge der affinen Transformationen und den Fakt, dass alle Unterpartitionen einer Partition gleich groß sind, die Position eines Blocks beim Decoden abgeleitet werden kann.[22]

16 6, S. 10.
17 6, S. 12 f.
18 6, S. 16.
19 6, S. 17 f.
20 6, S. 18, 16.
21 6, S. 13.
22 6, S. 21 f.

3.2.3 *Decoding*

Um ein fraktal codiertes Bild zu decoden, genügt es, nach dem Auslesen der affinen Transformationen, der Kontrast- und Sättigungswerte und der Range- und Domain-Blöcke die Transformationen in einem Feedback-Loop anzuwenden. Als Startbild kann dabei ein beliebiges, vom Decoder vorgegebenes Bild gewählt werden. Es hat zwar keinen Einfluss auf das schlussendliche Aussehen des decodeten Bildes, kann aber, wenn es intelligent gewählt wird, zu einem schnelleren Decoding-Prozess führen.[23] Im Regelfall nähert sich das decodete Bild nach zehn Iterationen zur Genüge an den Attraktor des iterierten Funktionssystems an. Wird der Prozess nach weniger Iterationen abgebrochen, hat das erzeugte Bild eine niedrigere Auflösung und enthält eventuell visuelle Artefakte.[24]

3.3 WEIGHTED FINITE AUTOMATA

Weighted Finite Automata (WFA) sind eine spezielle Ausprägung fraktaler Bildkompressionsverfahren. Das Bild- und Videocodec, das im Laufe dieser Arbeit erweitert werden soll, baut auf dieser Technik auf.

3.3.1 *Automat*

Ein WFA ist charakterisiert durch

1. Q, eine endliche Menge an Zuständen
2. Σ, ein endliches Alphabet, bspw. $\Sigma = 0, 1$
3. $W_a : Q \times Q \to \mathbb{R}$, eine Funktion, die für jedes $a \in \Sigma$ die Gewichtung des Übergangs zwischen zwei Zuständen zurückgibt
4. $I \in Q$, die Startzustände
5. $F \in Q$, die Endzustände

Ein Übergang zwischen zwei Zuständen unter dem Zeichen a besteht, wenn W_a für diese beiden Zustände eine Gewichtung größer als Null zurückgibt.[25]

3.3.2 *Encoding*

Das Kernstück eines WFA-Codecs ist der gewichtete endliche Automat. Auch hier wird das Bild beim Encoding-Prozess in Range-Blöcke partitioniert, diesmal allerdings nach dem Adaptive-Bintree-Schema, wobei immer nur zwei Unterpartitionen erstellt werden. Deren Teilung erfolgt abwechselnd horizontal und vertikal. Nach jedem Schritt wird für die neu erzeugten Partitionen geprüft, ob sie sich als lineare Kombination der derzeit bekannten

23 6, S. 164.
24 6, S. 20.
25 6, S. 246.

Domain-Blöcke beschreiben lassen. Ist dies nicht der Fall, wird erneut partitioniert.[26]

Bei jeder Iteration der Partitionierung werden neue Zustände in den gewichteten Automaten eingefügt. Von jedem Zustand aus führt der Übergang beim Zeichen 0 zu seinem linken, beziehungsweise oberen, Kind, je nachdem ob eine horizontale oder eine vertikale Trennung vollzogen wurde und der Übergang beim Zeichen 1 zu seinem rechten, oder unteren, Kind. Die vorab zu treffende Richtungszuweisung von 0 und 1 zu Richtungen kann abweichen und von jedem En- und Decoder selbst festgelegt werden.[27]

Ist es möglich, eine Partition unterhalb einer gewissen Kostengrenze aus einer linearen Kombination existierender Domain-Images zu erzeugen, wird die Gewichtung angepasst. Bis zu diesem Zeitpunkt wurde bei allen Übergängen des Automaten als Gewichtung $1,0$ angegeben. Wenn eine Partition nun beispielsweise aus der Kombination dreier anderer Partitionen gebildet werden kann, wobei eine doppelt so hoch wie die beiden anderen gewichtet wird, können vom Zustand der betrachteten Partition 3 Übergänge zu den Zuständen der zugrundeliegenden Partitionen erstellt werden. Dabei hat einer der Übergänge die Gewichtung $0,5$, die beiden anderen dagegen $0,25$. Der Zustand übernimmt somit beim Durchlauf des WFA die Werte seiner Kinder und gewichtet sie unterschiedlich stark. Dadurch wird der zugrundeliegende Automat nichtdeterministisch. Der WFA wird in der Regel mit einigen Basiszuständen initialisiert, die grundlegende Muster abdecken, zum Beispiel ein komplett schwarzer Zustand, ein Zustand mit einem Schachbrettmuster und einer mit Linienmuster.[28]

Der durch diesen Prozess erzeugte Automat repräsentiert das encodete Bild. Er wird abschließend in eine Datei geschrieben oder vom Encoder ausgegeben. Auch hier können die Entropie-Coding-Methoden aus 2.2.1.3 angewendet werden, um die geschriebenen Daten zu komprimieren.

3.3.3 *Decoding*

Zur Anzeige des Bildes wird der WFA mit den entsprechenden Gewichtungen aus der codierten Datei geladen und mit allen darzustellenden Bildpunkten durchlaufen. Die Bildpunkte werden dabei als binäre Folgen angegeben, die der Partitionierung im Adaptive-Bintree-Schema entsprechen. Eine 0 bedeutet, dass die linke oder obere Unterpartition ausgewählt wird, eine 1 entsprechend die rechte oder untere. Auch hier kann die Zuordnung von 0 und 1 vom Encoder anders festgelegt worden sein. Sobald einer der Basiszustände erreicht wird, die nur noch über Übergänge auf sich selbst verfügen und so nach dem vollständigen Lesen der Eingabe einen Wert zurückgeben, kann die entsprechende Partition vom Decoder in Grauwerte auf dem Ausgabebild umgesetzt werden. Die Grauwerte werden über die Gewichtungen

26 10, S. 30 ff.
27 10, S. 10 ff.
28 10, S. 10 ff.

der einzelnen Übergänge angepasst. Beim Decoden von Farbbildern wird für jeden Farbkanal ein eigener WFA erstellt und decodet.[29]

3.3.4 *Unterschiede*

Zwischen dem fraktalen Coding mit affinen Transformationen und mit WFA bestehen einige grundlegende Unterschiede.

Da die Partitionierung nur so lange fortgeführt wird, bis ein Range-Block durch eine lineare Kombination aus Domain-Blöcken beschrieben werden kann, verfügen mit WFA codierte Bilder nicht über Details in jedem Maßstab. Decodet man ein so codiertes Bild in einem vergrößerten Maßstab, so ergeben sich keine neuen Details, sondern die Kontrastkanten bleiben in ihrer ursprünglichen Auflösung bestehen.

Beim Decoden wird, im Gegensatz zu Codecs, die auf iterierte Funktionssysteme setzen, nur eine Iteration des Decoding-Prozesses durchgeführt. Sobald alle Partitionen vom Decoder abgearbeitet wurden, ergibt sich das fertige Bild, ohne dass ein Feedback-Loop zum Einsatz kommt.

29 10, S. 93 ff.

4 KONZEPTION EINES FRAKTALEN VIDEOCODECS

4.1 ZIELSETZUNG

4.1.1 *Verbesserung eines fraktalen Videocodecs*

Im Folgenden wird ein Programm konzipiert, das die Coding-Fähigkeiten des FIASCO als Basis hat und darauf aufbaut. Zweck des Programms ist es, Videodateien mit einer besseren Kompressionsrate bei gleichbleibender Qualität zu encoden. Nach der Konzeption wird in Kapitel 5 der Entwicklungsprozess und der Quellcode des Programms vorgestellt und in Kapitel 6 das Programm mit anderen etablierten Codecs verglichen.

4.1.2 *Referenz derselben Fraktale*

Die verbesserte Kompression soll durch ein Referenzieren derselben Domain-Blöcke in mehreren Einzelbildern erfolgen. FIASCO persistiert die verfügbaren Domain-Blöcke beim Laden eines neuen Einzelbildes nicht, sondern erstellt einen neuen WFA, der mit einigen Basiszuständen initiiert wird. Dieses Verhalten soll mit dem entwickelten Programm erweitert werden, sodass mehrere Einzelbilder eines Videos mit demselben WFA und somit auch mit denselben referenzierten Zuständen encodet werden. Durch den Zugriff auf dieselben Partitionen, besonders in aufeinanderfolgenden Einzelbildern, die sich häufig stark ähneln, wird eine Datenersparnis erwartet, die mit dem Einsatz von Prediction-Frames mithalten kann.

4.2 FIASCO

FIASCO ist ein Bild- und Videokompressionssystem, das auf fraktalen Coding-Verfahren mithilfe von WFA aufbaut. Es wurde im Zuge der 1999 erschienenen Dissertation "Low Bit-Rate Image and Video Coding with Weighted Finite Automata"[1] von Ullrich Hafner entwickelt.[2]

4.2.1 *Aufbau*

Beim Kompilieren von FIASCO werden einige Binaries erstellt, die unterschiedliche Anwendungen zur Interaktion mit codierten Dateien bereitstellen, beziehungsweise Bilddateien codieren.

---

1 10.
2 9, README.

4.2.1.1 *cfiasco*

cfiasco ist ein Kommandozeilenprogramm, das es ermöglicht, Dateien von einer Portable Any Map (PNM) zu FIASCO-Dateien zu encoden. PNM ist ein einfach auszulesendes und zu verarbeitendes abstraktes Bildformat[3], das Farbwerte, im Falle des PPM-Formats, beziehungsweise Grauwerte, im Falle des PGM-Formats, pro Pixel als Raster, entsprechend der Höhe und Breite des encodeten Bildes, speichert.[4,5,6]

4.2.1.2 *dfiasco*

dfiasco ist ein Kommandozeilenprogramm zum Decoden von Bildern und Videos, die im FIASCO-Format vorliegen. Die Ausgabebilder werden im PNM-Format geschrieben.[7,8]

4.2.1.3 *Weitere Module*

Weitere Module, die bei der Kompilierung erzeugt werden, sind libfiasco, eine C-Bibliothek, die das En- und Decoden von FIASCO-Dateien in anderen Anwendungen ermöglicht, afiasco, eine X11-Applikation, die codierte Dateien analysieren und visualisieren kann, bfiasco, ein Tool, das den binären Partitionsbaum einer FIASCO-Datei visualisiert, efiasc,o ein Programm zum Bearbeiten und Verketten von FIASCO-Streams, und gimpplugin, eine Erweiterung für das GNU Image Manipulation Program zum Anzeigen und Bearbeiten von FIASCO-Dateien.[9]

4.2.2 *Bildcodierung*

Das Encoding von Bildern in FIASCO erfolgt nach den in 3.3 erläuterten Prinzipien. Ein WFA wird mit den Ausschnitten eines partitionierten Bildes befüllt, die sich gegenseitig mit Gewichtungen referenzieren können und so die Selbstähnlichkeit des Bildes ausnutzen.[10] Die FIASCO-Datei, mit der Dateiendung .fco, die nach dem Encoden gespeichert wird, enthält den mit Entropie-Coding-Methoden komprimierten WFA.

Beim Decoden wird der in der codierten Datei beschriebene WFA aufgebaut und für alle möglichen Partitionen ausgeführt, bis ein komplettes Bild entstanden ist.[11]

---

3 18.
4 19.
5 9, README.
6 10, S. 106 (beschreibt frühere Version).
7 9, README.
8 10, S. 106 (beschreibt frühere Version).
9 9, README.
10 10, S. 23 ff.
11 10, S. 93 ff.

4.2.3 *Videocodierung*

FIASCO bietet zudem die Möglichkeit, Videos zu encoden. Diese werden als Folge von einzelnen PNM-Dateien nach einem Template eingelesen und der Reihe nach bearbeitet.[12]

Die Motion Compensation wird nach dem Vorbild der I-, P- und B-Frames aus 2.2.2 umgesetzt. Es werden also zwischen mehreren Frames einzelne Bildausschnitte referenziert, allerdings handelt es sich dabei in der Regel nicht um dieselben Blöcke, die in den WFA eingefügt werden.[13]

Die codierte Einzelbildfolge wird, ebenso wie ein codiertes Einzelbild, in eine FIASCO-Datei geschrieben, wobei die einzelnen Frames in ihrer ursprünglichen Reihenfolge angeordnet werden.

Beim Decoden eines FIASCO-Videos werden die WFA für die einzelnen Frames nacheinander aufgebaut und für alle Partitionen ausgeführt. WFA werden zwischen den Frames nicht persistiert, sondern stets neu aufgebaut. Beim Einsatz von P- und B-Frames werden Pointer auf das vorherige und gegebenenfalls das nachfolgende Frame vom Decoding-Kontext gespeichert, um Domain-Blöcke zu referenzieren.

4.3 VORGEHEN

Die Referenz derselben fraktalen Partitionen in mehreren Einzelbildern wird durch das gleichzeitige Encoden mehrerer Einzelbilder eines Videos umgesetzt. Dadurch teilen diese sich denselben WFA und sparen potenziell redundante Zustände in aufeinanderfolgenden Frames ein.

4.3.1 *Encoding*

Das zu encodende Video wird in GoPs eingeteilt, die einer beim Encoden vorgegebenen Länge entsprechen. Die GoPs werden jeweils als PNM-Datei gespeichert, in der die Einzelbilder nach einem vorgegebenen Layout angeordnet werden. Diese Datei wird mit cfiasco encodet und als FIASCO-Datei gespeichert.

4.3.2 *Decoding*

Nach dem Decoden der FIASCO-Datei durch dfiasco werden aus den GoPs wieder Videodateien erstellt, die die Einzelbilder in der korrekten Reihenfolge abspielen. Diese Videodateien werden verkettet und die verkettete Videodatei wird gespeichert.

12 10, S. 106.
13 10, S. 61 ff.

5 ENTWICKLUNG EINES FRAKTALEN VIDEOCODECS

5.1 ANSATZ

Die Umsetzung des geplanten Vorgehens erfolgt durch die Entwicklung eines En- und Decoders in der Programmiersprache Golang. Die Partitionierung der Bilder wird mit den Tile- und Untile-Video-Filtern von FFmpeg umgesetzt, die das Aufteilen der Frames eines Videos auf mehrere getilete Bilder, beziehungsweise das Aneinanderfügen der Frames von mehreren getileten Bildern zu einem Video ermöglichen. Beim Encoden wird cfiasco mit den generierten Partitionen aufgerufen, beim Decoden dfiasco, wodurch die getileten Partitionen wieder aufgebaut werden. Die erstellten Dateien werden abschließend gespeichert.

5.1.1 *FFmpeg*

FFmpeg ist ein Multimedia-Framework, das das En- und Decoden, Bearbeiten und Abspielen verschiedener Bild-, Video- und Audioformate erlaubt. Im hier entwickelten Programm wird es, ebenso wie cfiasco und dfiasco, als externes Programm über das von Golang bereitgestellte *exec*-Paket aufgerufen. FFmpeg wird zum einen für die Partitionierung der GoPs genutzt, zum anderen bietet es Unterstützung zum Einlesen und Ausgeben vieler verschiedener Videoformate. So ist es beispielsweise möglich, ein MP4-Video in FIASCO zu encoden und danach als WebM-Video zu decoden.[1]

5.1.2 *Quellcode*

5.1.2.1 *Main*

Das *main*-Paket beinhaltet den kompletten Programmablauf und ruft je nach übergebenen Parametern die Encode- und Decode-Funktionen der *ffmpeg*- und *fiasco*-Pakete auf.

In den Zeilen 14 bis 22 werden einige Konstanten deklariert, die zum einen die verfügbaren Aktionen des Codecs beschreiben und zum anderen in späteren Teilen des Quellcodes eine konsistente Benennung der temporär erzeugten Dateien gewährleisten.

Von Zeile 24 bis 78 werden die Programmargumente mit dem auf Github für Golang verfügbaren *argparse*-Paket[2] eingelesen und gegebenenfalls mit Standardwerten initialisiert. Für die korrekte Ausführung des Programms sind mindestens die Argumente *action*, das entscheidet, ob en- oder decodet

1 3.
2 11.

werden soll, *input* und *output* erforderlich. Mit *layout* lässt sich das Tiling-Layout, in dem die GoPs ausgelegt werden, anpassen, während *fps* die Wiederholungsrate, mit der ein Video decodet wird, verändern kann. *ffmpegPath*, *cfiascoPath* und *dfiascoPath* können genutzt werden, wenn sich die jeweiligen Binaries nicht über die PATH-Variable finden lassen. Mit *ffmpegArgs* und *fiascoArgs* können benutzerdefinierte Parameter an die jeweiligen Programme bei deren Aufruf durchgereicht werden.

Von Zeile 80 bis 102 wird anhand des *action*-Arguments entschieden, ob ein Encoding- oder ein Decoding-Prozess durchgeführt wird. In beiden Fällen werden die entsprechenden Funktionen des *ffmpeg*- und des *fiasco*-Pakets ausgeführt. Die Input- und Output-Dateinamen werden unter Zuhilfenahme der zuvor definierten Konstanten erzeugt und in die Aufrufe übergeben. Am Ende beider Vorgänge werden alle temporären Dateien entfernt.

Die Funktion zum Entfernen dieser Dateien ist in Zeile 105 bis 117 definiert. Hier werden zuerst alle Dateien, die während des Coding-Prozesses temporär erzeugt werden, durch die Ausführung eines Glob-Kommandos nach dem definierten Bezeichnungsschema ausgewählt. Die gefundenen Dateien werden im Anschluss gelöscht.

Zeile 120 bis 139 beinhaltet eine Funktion zum Validieren des optionalen *layout*-Arguments. Dieses muss aus zwei ganzen Zahlen, getrennt durch ein „x“, bestehen. Folgt der eingegebene Wert nicht diesem Format, wird ein Fehler an den Benutzer ausgegeben.

main.go

```
1  package main
2
3  import (
4   "FiascoExtension/ffmpeg"
5   "FiascoExtension/fiasco"
6   "errors"
7   "fmt"
8   "github.com/akamensky/argparse"
9   "os"
10  "path/filepath"
11  "regexp"
12 )
13
14 const (
15  ActionEncode = "encode"
16  ActionDecode = "decode"
17
18  EncodingTempFilename         = "out/frame"
19  EncodingTempExtension        = "ppm"
20  EncodingTempFiascoWildcard = "[001-%03d+1]"
21  EncodingTempFFmpegWildcard = "%03d"
22 )
23
24 func main() {
25  // Read program arguments
26  parser := argparse.NewParser("FiascoExtension", "Extends the
        functionality of Fiasco")
```

```
27  action := parser.Selector("a", "action", []string{ActionEncode,
        ActionDecode}, &argparse.Options{
28   Required: true,
29   Help:     "Action to run the coder with. One of: [" + ActionEncode
         + ", " + ActionDecode + "].",
30  })
31  input := parser.String("i", "input", &argparse.Options{
32   Required: true,
33   Help:     "Input file to encode/decode from.",
34  })
35  output := parser.String("o", "output", &argparse.Options{
36   Required: true,
37   Help:     "Output file to encode/decode to.",
38  })
39  layout := parser.String("l", "layout", &argparse.Options{
40   Required: false,
41   Validate: validateLayout,
42   Help: "Layout to tile the picture groups in. Specified in the
         format '4x1', where the first number is the " +
43    "width and the second number is the height of the tiling.",
44   Default: "1x8",
45  })
46  fps := parser.Int("f", "fps", &argparse.Options{
47   Required: false,
48   Help:     "Target fps for the decoded video.",
49   Default:  25,
50  })
51  ffmpegPath := parser.String("", "ffmpegPath", &argparse.Options{
52   Required: false,
53   Help:     "Override the path to the ffmpeg binary.",
54   Default:  "ffmpeg",
55  })
56  cfiascoPath := parser.String("", "cfiascoPath", &argparse.Options{
57   Required: false,
58   Help:     "Override the path to the cfiasco binary.",
59   Default:  "cfiasco",
60  })
61  dfiascoPath := parser.String("", "dfiascoPath", &argparse.Options{
62   Required: false,
63   Help:     "Override the path to the dfiasco binary.",
64   Default:  "dfiasco",
65  })
66  ffmpegArgs := parser.String("", "ffmpegArgs", &argparse.Options{
67   Required: false,
68   Help:     "Additional arguments to append to the ffmpeg command.",
69  })
70  fiascoArgs := parser.String("", "fiascoArgs", &argparse.Options{
71   Required: false,
72   Help:     "Additional arguments to append to the fiasco command.",
73  })
74
75  err := parser.Parse(os.Args)
76  if err != nil {
77   fmt.Print(parser.Usage(err))
78  }
79
80  switch *action {
```

```
81  case ActionEncode:
82   // Tile a given videos frames into .ppm files and store the number
         of produced files
83   matches, err := ffmpeg.Encode(*input, EncodingTempFilename+
         EncodingTempFFmpegWildcard+"."+EncodingTempExtension,
84    *ffmpegPath, *layout, *ffmpegArgs)
85   if err != nil {
86    panic(err)
87   }
88
89   // Encode tiled files into one .fco file that is named according
         to the naming constants
90   err = fiasco.Encode(fmt.Sprintf(EncodingTempFilename+
         EncodingTempFiascoWildcard+"."+EncodingTempExtension, matches)
         ,
91    *output, *cfiascoPath, *fiascoArgs)
92   cleanupCodingFiles()
93  case ActionDecode:
94   // Decode .fco compressed file into tiled .ppm files
95   err = fiasco.Decode(*input, EncodingTempFilename+"."+
         EncodingTempExtension, *dfiascoPath, *fiascoArgs)
96
97   // Fiasco puts out files in the format of '[filename without
         extension].[sequence number].[extension]'
98   err = ffmpeg.Decode(fmt.Sprintf("%s.%%*.%s", EncodingTempFilename,
         EncodingTempExtension),
99    *output, *ffmpegPath, *layout, *fps, *ffmpegArgs)
100  cleanupCodingFiles()
101 }
102}
103
104// cleanupCodingFiles Removes all temporary files that are produced
     during encoding/decoding
105func cleanupCodingFiles() {
106 files, err := filepath.Glob(EncodingTempFilename + "*." +
       EncodingTempExtension)
107 if err != nil {
108  fmt.Println(err)
109 }
110
111 for _, f := range files {
112  err := os.Remove(f)
113  if err != nil {
114   fmt.Println(err)
115  }
116 }
117}
118
119// validateLayout Checks if a given tiling layout has the correct
     format
120func validateLayout(args []string) error {
121 if len(args) <= 0 {
122  return errors.New("no layout parameter specified")
123 }
124 if len(args) > 1 {
125  return errors.New("too many layout parameters specified")
126 }
```

```
127
128  matched, err := regexp.Match('^\d+x\d+$', []byte(args[0]))
129  if err != nil {
130   return err
131  }
132
133  if !matched {
134   return errors.New("invalid layout specified. Use the format '4x1',
          where the first number is the width " +
135    "and the second number is the height of the tiling")
136  }
137
138  return nil
139 }
```

5.1.2.2 *FFmpeg*

Das *ffmpeg*-Paket ermöglicht das Tiling und Untiling der GoPs sowie die Verarbeitung unterschiedlicher Videoformate.

In Zeile 11 bis 39 findet sich die Funktion, die beim Encoden einer Videodatei die GoPs auf einzelne partitionierte Bilddateien verteilt. Zu Beginn werden die Parameter für den FFmpeg-Aufruf in ein Array geschrieben. Das *input*-Argument verweist auf die vom Benutzer angegebene zu verarbeitende Videodatei. Der *y*-Marker erlaubt FFmpeg das Überschreiben bestehender Dateien. Das Video-Filter Argument, *vf*, konfiguriert das Tiling der Einzelbilder nach einem bestimmten Layout unter Verwendung des Tile-Filters. *output* ist der Pfad, unter dem die entstandenen Bilddateien abgelegt werden.

Sofern benutzerdefinierte FFmpeg-Parameter übergeben wurden, werden diese im Folgenden an den Anfang des Arrays gesetzt, da FFmpeg nach dem Output-Pfad keine Parameter mehr erwartet. Der Aufruf von FFmpeg erfolgt über das *exec*-Paket, das die Möglichkeit bereitstellt, ein Kommandozeilenprogramm mit optionalen Parametern zu starten. Die Ausgabe dieses Programms wird auf den Standard Output weitergeleitet. Nach dem Start des FFmpeg-Prozesses wird bis zu dessen Halt abgewartet. Sofern bei der Ausführung kein Fehler auftritt, wird die Anzahl der erzeugten GoPs ermittelt. Dies geschieht durch ein Glob-Kommando das nach dem Template, in dem die erzeugten Dateien vorliegen, sucht. FFmpeg selbst gibt diese Zahl nicht zurück. Die Anzahl der gefundenen Dateien wird abschließend aus der Funktion zurückgegeben.

In Zeile 42 bis 61 befindet sich die Funktion, die beim Decoden einer Videodatei die partitionierten GoPs wieder zu einem Video zusammensetzt. Der grundlegende Ablauf ähnelt der Encoding-Funktion. Die Unterschiede bestehen zum einen in der Verwendung des Untile-Filters und dem Setzen einer Bildwiederholrate im FFmpeg-Aufruf, um das Video korrekt zusammenzusetzen, und zum anderen darin, dass die Funktion keinen Wert zurückgibt.

ffmpeg/ffmpeg.go

```
1 package ffmpeg
```

```
2
3  import (
4   "os"
5   "os/exec"
6   "path/filepath"
7   "strconv"
8  )
9
10 // Encode Splits a video file into a number of tiled .ppm image
       files containing the frames
11 func Encode(input string, output string, path string, layout string,
       customArgs string) (matches int, err error) {
12  args := []string{"-i", input, "-y", "-vf", "tile=layout=" + layout,
       output}
13  if customArgs != "" {
14   args = append([]string{customArgs}, args...)
15  }
16
17  cmd := exec.Command(path, args...)
18
19  // Verbosity
20  cmd.Stderr = os.Stdout
21
22  err = cmd.Start()
23  if err != nil {
24   return 0, err
25  }
26
27  err = cmd.Wait()
28  if err != nil {
29   return 0, err
30  }
31
32  // Count the number of files produced
33  matchesSlice, err := filepath.Glob("out/frame*.ppm")
34  if err != nil {
35   return 0, err
36  }
37
38  return len(matchesSlice), err
39 }
40
41 // Decode Combines a number of tiled .ppm images containing frames
       into a video file
42 func Decode(input string, output string, path string, layout string,
       fps int, customArgs string) error {
43  // if '-f image2' isn't specified before the input file, ffmpeg
       fails to use wildcards correctly
44  // start_number is set to 0, because Fiasco starts its output files
       at 0
45  args := []string{"-f", "image2", "-i", input, "-y", "-vf", "untile=
       " + layout + ",setpts=N/(" + strconv.Itoa(fps) + "*TB)", "-
       start_number", "0", output}
46  if customArgs != "" {
47   args = append([]string{customArgs}, args...)
48  }
49
```

```
50  cmd := exec.Command(path, args...)
51
52  // Verbosity
53  cmd.Stderr = os.Stdout
54
55  err := cmd.Start()
56  if err != nil {
57   return err
58  }
59
60  return cmd.Wait()
61 }
```

5.1.2.3 *FIASCO*

Das *fiasco*-Paket stellt die Funktionen zum Encoden und Decoden von getileten PNM-Dateien zu und von FIASCO-Dateien bereit.

Zeile 9 bis 27 beinhaltet die Funktion zum Encoden einer Anzahl getileter PNM-Dateien. Auch hier ist der Aufbau den Coding-Methoden des *ffmpeg*-Pakets ähnlich. Der Aufruf der cfiasco-Binary erhält als Parameter das Verbosity-Argument, *-V*, mit dem Wert zwei, wodurch eine Fortschrittsanzeige beim Encoden aktiviert wird, ein *input*-Argument, das ein Template, das anhand der Anzahl erzeugter Dateien des FFmpeg-Encode in der *main*-Funktion erstellt wird, beinhaltet, einen Output-Dateinamen, der vom Benutzer vorgegeben werden kann und ein GoP-Pattern, das dazu führt, dass FIASCO die eingegebenen Bilder ausschließlich als I-Frames encodet.

Von Zeile 30 bis 47 findet sich die Funktion zum Decoden einer FIASCO-Datei zu einer Menge getileter PNM-Dateien. Hierbei werden in den Argumenten von dfiasco eine Output-Datei, die vom Benutzer vorgegeben werden kann, und eine Input-Datei, bei der es sich um das zu decodende FIASCO-Video handelt, angegeben. Der übrige Ablauf gleicht den anderen En- und Decoding-Funktionen.

fiasco/fiasco.go

```
1  package fiasco
2
3  import (
4   "os"
5   "os/exec"
6  )
7
8  // Encode Encodes several files matching with the input pattern in
       fiasco
9  func Encode(input string, output string, path string, customArgs
       string) error {
10  // Only encode to I-Frames for now, as the default pattern causes
        crashes while decoding
11  args := []string{"-V", "2", "-q", "100", "-i", input, "-o", output,
        "--pattern=I"}
12  if customArgs != "" {
13   args = append([]string{customArgs}, args...)
14  }
```

```
15
16  cmd := exec.Command(path, args...)
17
18  // Verbose
19  cmd.Stderr = os.Stdout
20
21  err := cmd.Start()
22  if err != nil {
23   return err
24  }
25
26  return cmd.Wait()
27 }
28
29 // Decode Decodes a fiasco file
30 func Decode(input string, output string, path string, customArgs
       string) error {
31  args := []string{"-o", output, input}
32  if customArgs != "" {
33   args = append([]string{customArgs}, args...)
34  }
35
36  cmd := exec.Command(path, args...)
37
38  // Verbose
39  cmd.Stderr = os.Stdout
40
41  err := cmd.Start()
42  if err != nil {
43   return err
44  }
45
46  return cmd.Wait()
47 }
```

5.1.3 *Entwicklungsprobleme*

Bei der Entwicklung des Programms sind, vor allem mit den FIASCO-Binaries, einige Probleme aufgetreten, die im Folgenden erläutert werden.

Beim Encoden von großen Bildern, für die die Standardkonfiguration des FIASCO-Encoders nicht ausgelegt ist, kam nach etwa der Hälfte der Laufzeit regelmäßig der Fehler „Error: Maximum number of states reached“ auf. Dieser ließ sich, aufgrund der aussagekräftigen Fehlermeldung, schnell durch das Erhöhen der MAXSTATES-Konstante in der Header-Datei codec/wfa.h beheben. Die Konstante begrenzt die Anzahl an Zuständen, die der dem Bild zugrundeliegende WFA beinhalten darf, und steht standardmäßig auf 6000. Beim Encoden von Standard HD Videos mit einer GoP-Größe von 4 hat sich ein Wert von 30000 Zuständen als zweckmäßig herausgestellt. Bei noch höheren Auflösungen sollte die Variable entsprechend angepasst werden.

Nach dem Beheben dieses Fehlers kam es beim Encoden von großen Einzelbildern weiterhin zu Abstürzen, die diesmal durch einen Segmentation Fault, also den Zugriff des Programms auf einen ungültigen Speicherbe-

reich, ausgelöst wurden. Der Grund hierfür liegt in der begrenzten Partitionstiefe, die durch die MAXLEVEL-Konstante in codec/wfa.h gesteuert wird. Beim Überschreiten dieser Grenze greift der Encoder während des Partitionierens auf Offsets der Arrays des Baummodells zu, die sich außerhalb deren initialer Grenzen befinden. Zusätzlich zum Erhöhen der MAXLEVEL-Konstante muss auch die Initialisierung der Arrays counts_0 und counts_1 in der Datei codec/bintree.h, die die Konstante als Längenwert verwenden, angepasst werden. Zu diesem Zweck wurde vorerst der Wert des letzten Array-Elements mehrfach dupliziert. Der standardmäßige Wert der Konstante beträgt 22, als zweckmäßiger neuer Wert hat sich 27 herausgestellt.

Beim Encoden von Videos mit hoher Auflösung kam es immer noch zu Segmentation Faults, die behoben werden konnten, indem das Encoding-Pattern der Eingabebilder auf durchgehende I-Frames gesetzt wurde. Da sowieso jedes eingegebene Bild mit einem eigenen WFA codiert werden sollte, ist dies aber kein Problem für die Funktionalität des erweiterten Codecs.

Zudem wurden oft „Out of memory“-Fehlermeldungen ausgelöst, die beim Encoden von hoch aufgelösten Videos auftraten, die länger als ein bis zwei Sekunden auf einer virtuellen Maschine mit 15GB RAM andauerten. Diese Fehler werden hauptsächlich durch die Encoding-Context-Struktur verursacht, die während des Encoding-Prozesses von Bildern stetig anwächst und nicht komplett geleert wird. Der einfachste Weg, mit diesem Problem während der in 6 folgenden Messungen umzugehen, ist das Encoden von niedriger aufgelösten Videos. Ein weiterer Ansatz wäre eine Optimierung des Encoding-Context.

Durch die meisten der vorgestellten Fehler wird eine unvollständige und fehlerhafte FIASCO-Datei erzeugt, die beim Decoden durch dfiasco die Fehlermeldung „Error: Can’t read next bit from bitfile“ hervorbringt. Diese tritt auf, weil es bei der Eingabe des Wertes, der mit einem Adjusted-Binary-Code encodet sein sollte, in die Funktion bits_bin_code in der Datei codec/domain-pool.c:788 zu einem Integer Underflow kommen kann, wenn der zuletzt encodete Wert größer als der momentane ist. Dieser Fehler tritt naturgemäß nicht auf, wenn fehlerfreie Dateien encodet werden.

6 BETRACHTUNG DER ERGEBNISSE

Um das Potenzial des erweiterten FIASCO-Codecs einschätzen und vergleichen zu können, wird es im Folgenden mit geläufigen Videocodecs auf seine Einsatzmöglichkeiten hin verglichen.

6.1 TESTVIDEO

Ein Video mit geringen Unterschieden zwischen den einzelnen Frames wird untersucht, um einen Vergleich der Encoding-Vorgänge aller untersuchten Codecs zu erzielen. Das untersuchte Video läuft mit 25 Bildern pro Sekunde bei einer Länge von 30 Sekunden und hat eine Auflösung von 480 * 270 Pixeln, um die Länge des Encoding-Prozesses in FIASCO in Grenzen zu halten. Es zeigt den Anblick der Erde aus dem All, wobei sich die Erdkugel kontinuierlich dreht. Abgesehen von der Bewegung der Kontinente hat das Video keine Bereiche, die sich zwischen den Frames verändern. Motion Compensation sollte hier also starke Kompression zwischen den Einzelbildern erreichen können. Auch die Zustände des WFA sollten sich gegenseitig innerhalb einer GoP oft referenzieren können.[1]

Abbildung 6.1: Einzelbild aus dem originalen Testvideo

6.2 METRIKEN

Um die Codecs zu vergleichen, werden die Videos in den untersuchten Codecs encodet und zurück in das Ursprungsformat decodet. Die decodeten

1 5.

Dateien werden daraufhin in Bezug auf mehrere Metriken miteinander verglichen.

6.2.1 *Geschwindigkeit*

Die Geschwindigkeit beim Encoden wird mithilfe des Time-Kommandos in Linux festgestellt. Die Erwartung ist, dass das erweiterte FIASCO-Codec aufgrund des hohen Alters und der rechnerischen Komplexität von Fraktalen hier schlechter als etablierte Codecs abschneiden wird.

6.2.2 *Dateigröße*

Die Größe der Dateien im codierten Format in Byte wird verglichen. Die Erwartung hierbei ist, dass das erweiterte FIASCO-Codec ähnliche Ergebnisse wie etablierte Codecs erzielen kann und im Vergleich zum originalen FIASCO-Codec besser abschneidet.

6.2.3 *Bildqualität*

Bildqualität ist eine subjektive Metrik und wird in diesem Kontext vor allem anhand der wahrgenommenen Auflösung und Wiederholungsrate, sowie auftretender Encoding-Artefakten festgemacht.

6.3 MESSUNGEN

Die Messergebnisse der betrachteten Codecs wurden auf einem Linux-System mit 16GB RAM und einem 16-kernigen Prozessor mit einer Taktung von 3,20GHz erbracht. Bei allen Codecs wurden die Software-Encoder und Software-Decoder genutzt.

6.3.1 *Ergebnisse*

Codec	Encoding-Zeit	Dateigröße	Kompressionsrate
FIASCO (I- & P-Frames)	6:27,96 Min.	584 Byte	38,0%
FIASCO (I-Frames)	4:59,98 Min.	1388 Byte	90,4%
Erweitertes FIASCO (8x1)	27:30,63 Min.	1232 Byte	80,2%
Erweitertes FIASCO (1x4)	15:47,56 Min.	1148 Byte	74,7%
Erweitertes FIASCO (1x8)	30:38,15 Min.	1128 Byte	73,4%
Erweitertes FIASCO (2x2)	15:44,16 Min.	1220 Byte	79,4%
Erweitertes FIASCO (2x3)	23:39,28 Min.	1180 Byte	76,8%
Erweitertes FIASCO (3x2)	22:05,09 Min.	1180 Byte	76,8%
Erweitertes FIASCO (1x16)	60:48,71 Min.	1108 Byte	72,1%
Erweitertes FIASCO (4x4)	55:57,53 Min.	1008 Byte	65,6%
H.264 (CRF 0)	00:01,75 Min.	3800 Byte	234,4%
H.264 (CRF 18)	00:01,54 Min.	544 Byte	35,4%
H.264 (CRF 30)	00:01,05 Min.	160 Byte	10,4%
H.264 (CRF 51)	00:00,705 Min.	60 Byte	3,9%
VP9 (Best Quality)	01:00,79 Min.	2704 Byte	176,0%
VP9 (Constant Quality)	00:07,01 Min.	192 Byte	12,5%
VP9 (Constrained Quality)	00:08,02 Min.	512 Byte	33,3%

6.3.2 *FIASCO*

Der erste Ansatz zum Encoden eines Videos mit FIASCO besteht darin, das Testvideo mit FFmpeg in Einzelbilder aufzuteilen und diese anhand eines Template-Dateinamens in die cfiasco-Binary einzugeben. Grundsätzlich wird hier also ähnlich vorgegangen wie beim erweiterten FIASCO-Codec, in diesem Fall mit einem ein 1x1-Tiling-Layout. Aus diesem Grund kann die FIASCO-Erweiterung auch zum Evaluieren dieses Falls eingesetzt werden, da das von ihr verwendete Tiling-Layout konfigurierbar ist.

Beim Encoden der Einzelbilder mit FIASCO wurde zuerst das Standard-Frame-Type-Muster genutzt, das aus einem I-Frame und neun nachfolgenden P-Frames besteht, die sich jeweils auf das vorherige Frame beziehen. Der Encoding-Prozess läuft in knapp sechseinhalb Minuten durch. Die erzeugte Datei hat mit einer Größe von 584 Byte im Vergleich zu den 1536 Byte des Originals eine Kompressionsrate von 38%. Die codierte Datei lässt sich allerdings nicht decoden, da dfiasco beim ersten P-Frame, auf das es stößt, den Fehler "Wrong Huffman Code!"wirft. Der Grund für diesen Fehler konnte nicht gefunden werden und tritt auch auf, wenn eine Binary ohne die Anpassungen aus dieser Arbeit verwendet wird.

Sobald das Frame-Type-Muster von cfiasco auf ausschließlich I-Frames gesetzt wird, ist auch ein fehlerfreies Decoding möglich. Mit diesem Muster wird allerdings keinerlei Motion-Compensation benutzt, sondern jedes Einzelbild wird mit einem eigenen WFA encodet. Dementsprechend ist die Kom-

pressionsrate im Vergleich zum Original mit einem Wert von 90,4% deutlich schlechter. Die Encoding-Zeit beschleunigt sich dafür mit knapp 5 Minuten etwas. Das decodete Video hat deutliche Qualitätseinbußen. Die wahrgenommene Auflösung ist schlechter und es ist eine starke Blockbildung erkennbar, die vermutlich die Partitionierung der Range-Blöcke widerspiegelt. Zudem sind am unteren Bildschirmrand und an der linken Seite der Erdkugel am Äquator helle Artefakte sichtbar.

Abbildung 6.2: Einzelbild aus dem mit FIASCO encodeten Video mit Artefakten am Äquator und am unteren Bildschirmrand

6.3.3 *Erweitertes FIASCO*

Das Encoden des Testvideos mit der FIASCO-Erweiterung lässt sich ohne Probleme durchführen. Als initiales Tiling-Layout wird 8x1 gewählt, das heißt jede GoP wird als horizontale Folge von 8 Einzelbildern in eine PNM-Datei geschrieben.

Das Encoden dauert mit diesem Layout 27:30 Minuten, wobei das Tiling, das von FFmpeg übernommen wird, nur ein paar Sekunden in Anspruch nimmt. Während der restlichen Zeit wird der cfiasco-Prozess ausgeführt. Das encodete Video hat eine Größe von 1236 Byte, was 80,5% der Größe des Originalvideos entspricht. Beim Decoding fällt auf, dass das Ausführen der WFA durch dfiasco sehr schnell von statten geht, während der größere Teil der Ausführungszeit, die etwa dreizehn Sekunden beträgt, diesmal auf FFmpegs Untiling-Filter entfällt. Die Qualität des decodeten Videos ist der des reinen FIASCO-Videos ähnlich. Auch hier wird die wahrgenommene Auflösung stark reduziert und helle Artefakte tauchen auf dem Erdball und am unteren Bildrand auf. Zudem schwanken viele der Range-Blöcke deutlich in ihrem Helligkeits-Level.

Abbildung 6.3: Einzelbild aus einem 8x1-getileten Video

6.3.3.1 *Weitere Tiling-Layouts*

Zusätzlich zum initial getesteten 8x1-Tiling werden einige andere Tiling-Layouts auf ihre Auswirkungen auf den Encoding-Prozess hin untersucht.

Mit einem vertikal ausgerichteten 1x4-Layout, wird eine Encoding-Zeit von knapp 16 Minuten, sowie eine Kompressionsrate von 74,7% erreicht. Diese Rate lässt sich durch ein Verdoppeln der vertikalen Einzelbildanzahl auf 73,4% senken, wobei die Encoding-Zeit auf 30 Minuten, dafür aber auch die Bildqualität steigt, vor allem am Rand der Erde und in den Wolken.

Abbildung 6.4: Einzelbild aus einem 1x4-getileten Video

Die Nutzung eines quadratischen Layouts mit der Seitenlänge zwei resultiert in einem Kompressionslevel von 79,4% und erneut einer Encoding-Zeit von knapp 16 Minuten. Wird das Quadrat in X- beziehungsweise in Y-Richtung um eine Einheit erweitert, steigt die Encoding-Zeit auf 22, be-

Abbildung 6.5: Einzelbild aus einem 1x8-getileten Video

ziehungsweise 23,5 Minuten an, während die Kompressionsrate bei beiden Varianten 76,8% beträgt. Die Bildqualität ist bei all diesen Variationen der des mit dem 1x4-Layout getileten Videos ähnlich.

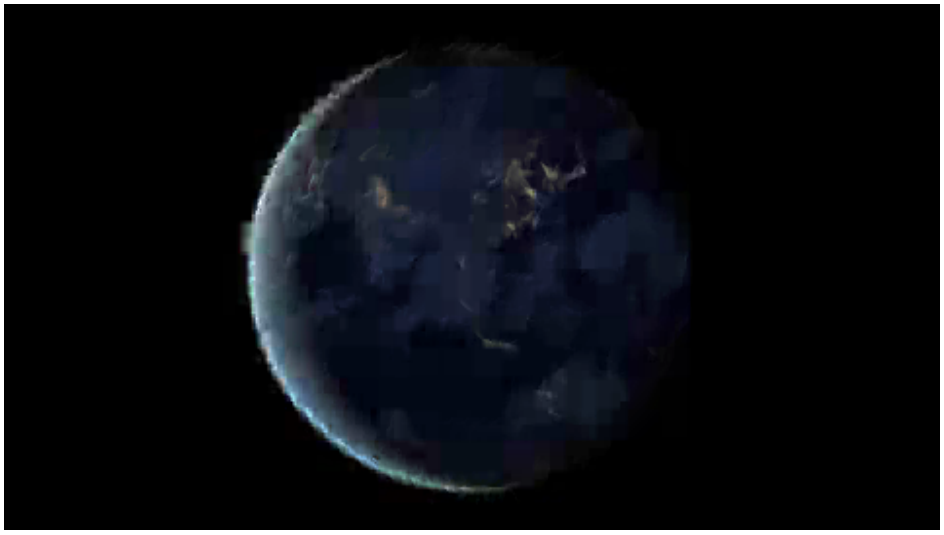

Abbildung 6.6: Einzelbild aus einem 2x2-getileten Video

Das Encoden eines vertikal ausgerichteten 1x16-Layouts nimmt eine ganze Stunde in Anspruch und resultiert in einer Kompressionsrate von 72,1%. Die beste Rate lässt sich allerdings mit einem Quadrat mit einer Seitenlänge von 4 Einzelbildern erreichen. Das Encoden dieser Variante nimmt knapp 56 Minuten in Anspruch und ermöglicht eine Kompression von 65,6% im Vergleich zum Originalvideo. Beide Layouts verfügen über eine bessere Auflösung und weniger Artefakte als die vorherigen Varianten.

Abbildung 6.7: Einzelbild aus einem 3x2-getileten Video

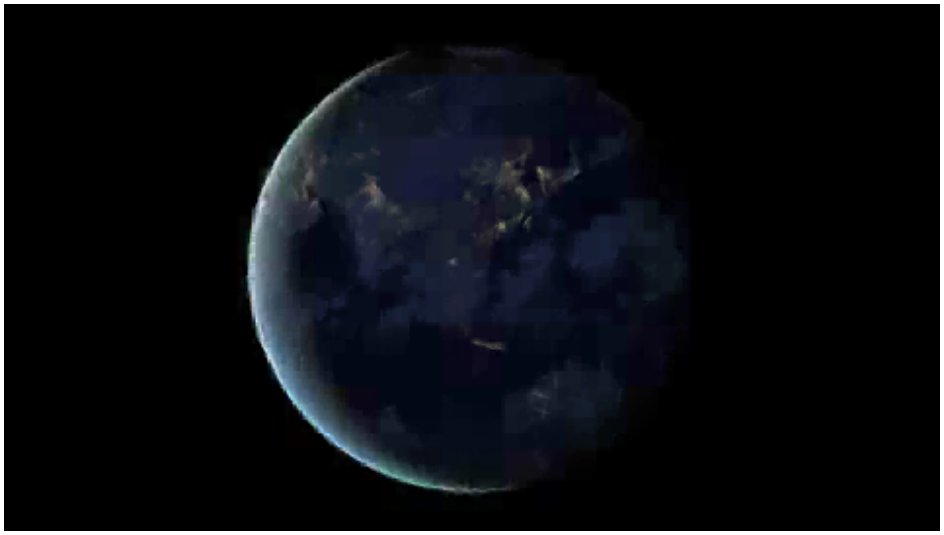

Abbildung 6.8: Einzelbild aus einem 1x16-getileten Video

6.3.4 *H.264*

Die Untersuchung des H.264-Codecs gestaltet sich etwas anders als die der restlichen Codecs, da das Testvideo bereits in H.264 encodet vorliegt. Aus diesem Grund wird das Testvideo mit FFmpeg in unterschiedlichen Qualitätseinstellungen zu H.264 transcodet. Da hierbei kein Zwischenformat erzeugt wird, von dem sich die Dateigröße messen lassen würde, beinhaltet die gemessene Encoding-Zeit auch die Decoding-Zeit.

Die Qualität eines mit H.264 encodeten Videos lässt sich über den CRF beeinflussen. Dabei handelt es sich um eine Variable, die Werte von 0 bis 51 annehmen kann, wobei 0 in verlustfreiem Encoding und 51 in maximal verlustbehaftetem Encoding resultiert. Die Messungen werden mit den CRF-Einstellungen 0, 18, 30 und 51 durchgeführt.

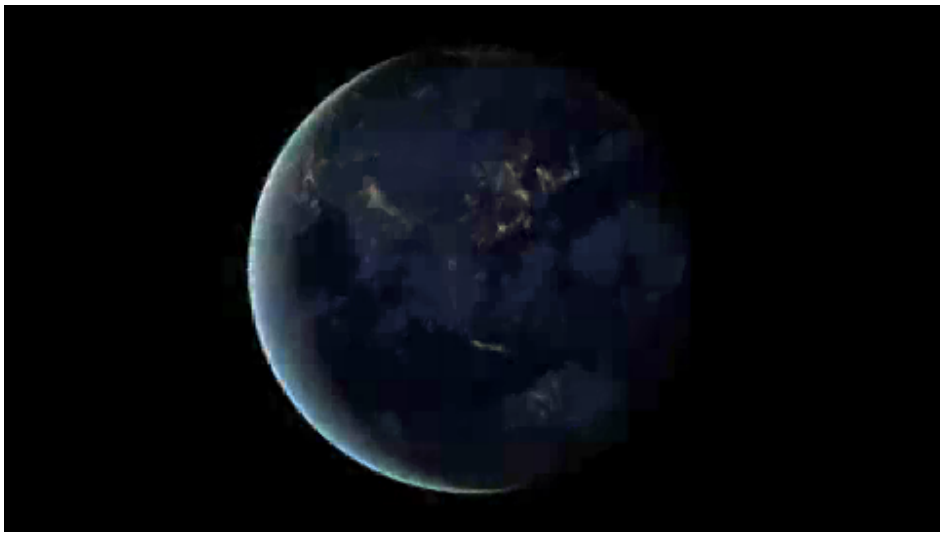

Abbildung 6.9: Einzelbild aus einem 4x4-getileten Video

Die Decoding-Zeit wird bei den mit H.264 und VP9 encodeten Videos nicht berücksichtigt, da diese Formate sich im encodeten Zustand von FFplay abspielen lassen. Daran lässt sich erkennen, dass mit diesen Codecs ein Decoding in Echtzeit auf jeden Fall möglich ist.

Bei einem CRF-Wert von 0 hat die resultierende Datei eine Größe von 3800 Byte, ist also sogar größer als das ursprünglich eingegebene Video. Das Transcoding ist in 1,7 Sekunden durchlaufen und die erzeugte Datei hat dieselbe Qualität wie das Original.

Abbildung 6.10: Einzelbild aus einem mit CRF=0 encodeten Video

Eine CRF-Einstellung von 18 resultiert in einem verlustbehafteten Video, das allerdings eine wahrgenommene Qualität hat, die dem des Originals sehr nahe ist. Das Transcoding ist in 1,5 Sekunden abgeschlossen und die erzeugte Datei erzielt eine Kompressionsrate von 35,4%.

Abbildung 6.11: Einzelbild aus einem mit CRF=18 encodeten Video

Das Setzen des CRF auf 30 erzeugt innerhalb von einer Sekunde ein Video mit einer Kompressionsrate von 10,4%, dem man diese Kompression ansieht. Vor allem an Kanten und Übergängen im Bild tauchen Artefakte der Motion-Compensation auf und auch die Bildqualität ist im Vergleich zu den vorherigen CRF-Werten verschwommener.

Abbildung 6.12: Einzelbild aus einem mit CRF=30 encodeten Video

Steht der CRF auf 51, kann eine Kompressionsrate von 3,9% erreicht werden. Das Encoding dauert in diesem Fall 0,7 Sekunden. Die Bildqualität ist dadurch allerdings extrem schlecht. Im erzeugten Video sind kaum noch Details erkennbar. Der Umriss der Erdkugel hebt sich zwar noch vom Hintergrund ab, die Kontinente sind allerdings schon nicht mehr von den Weltmeeren zu unterscheiden.

Abbildung 6.13: Einzelbild aus einem mit CRF=51 encodeten Video

6.3.5 *VP9*

Beim Erheben der Messdaten zu VP9 werden die empfohlenen Einstellungen zum Encoden von WebM-DASH-Dateien aus dem Wiki des webmproject verwendet, um unterschiedliche Qualitätsstufen zu erreichen.[2]

Mit der besten Qualitätseinstellung wird in einem Zeitraum von einer Minute ein WebM-Video mit einer Größe von 2704 Byte erstellt. Analog zu dem verlustfrei encodeten H.264-Video wird also auch hier eine Datei erzeugt, die deutlich größer als das Original ist. Die Qualität gleicht dabei auch weiterhin der des Originalvideos.

Abbildung 6.14: Einzelbild aus einem mit der besten Qualitätseinstellung encodeten Video

2 21.

Mit den empfohlenen Einstellungen für ein konstantes Qualitätslevel ist das Encoding in 7 Sekunden abgeschlossen und resultiert in einem 192 Byte großen Video, das also nur 12,5% der Größe des Originals hat. Die Qualität ist, besonders für diese starke Datenkompression, sehr gut. Es sind, ähnlich wie bei dem mit einem CRF-Wert von 18 encodeten H.264-Video, nur minimale Verschleierungen an den Rändern der Wolken zu sehen.

Abbildung 6.15: Einzelbild aus einem mit konstanter Qualitätseinstellung encodeten Video

Die Einstellungen für ein unter einer Bitrate von 64k liegendes Qualitätslevel resultieren in einem Video, in dem sich die Verschleierung und verschwommene Kanten aus dem vorherigen Test verstärken, das mit 33,3% der Größe des Originals allerdings deutlich mehr Speicher verbraucht. Die Encoding-Zeit unterscheidet sich mit knapp 8 Sekunden kaum von der des konstant qualitativen Videos.

Abbildung 6.16: Einzelbild aus einem mit limitierter Qualitätseinstellung encodeten Video

7 FAZIT

Grundsätzlich lassen sich sowohl mit FIASCO selbst als auch mit der in dieser Arbeit implementierten Erweiterung Kompressionsraten im Bereich von 38% bis 90% erreichen. Die nur mit I-Frames codierten Videos zeichnen sich vor allem durch eine verminderte Auflösung aus, die sich mehr oder weniger der Verteilung der Range-Blöcke anpasst. Zudem tauchen an charakteristischen Positionen, am Äquator und am unteren Bildrand, vermehrt helle Artefakte auf. Die Regelmäßigkeit, mit der diese Artefakte sichtbar werden, wird von der GoP-Größe beeinflusst. Je höher diese ist, desto seltener erscheinen sie. Auch im übrigen Bildbereich gibt es vermehrt flackernde Blöcke, die ihre Helligkeit zwischen den einzelnen Frames verändern. Diese werden vom Tiling-Layout allerdings nicht beeinflusst. Insgesamt haben sich in den durchgeführten Tests vertikale Tiling-Layouts als den horizontalen überlegen herausgestellt. Bei einer GoP-Größe von acht Einzelbildern hat das vertikale Layout eine um 7% bessere Kompressionsrate als das horizontale. Zudem tauchen hier keine Artefakte am unteren Bildrand auf. Die beste Kompression lässt sich allerdings mit quadratischen Layouts erzielen. Das 2x2-Layout wird doppelt so schnell wie das 1x8-Layout encodet, hat aber trotzdem eine Kompressionsrate, die nur um knapp 100 Byte darüber liegt (zugegebenermaßen mit einer niedrigeren Bildqualität) während das 4x4-Layout dem 1x16-Layout um ganze 100 Byte voraus ist. Zudem zeigt sich, dass eine Erweiterung des Quadrats in eine Achsenrichtung die gleichen Auswirkungen wie eine Erweiterung in die andere Richtung hat.

Die Encoding-Zeit steigt proportional zur gewählten GoP-Größe an. Die benötigte Zeit lässt sich in Abhängigkeit von der gewählten Größe annähernd durch die Funktion $f(x) = \frac{19x}{5}$ beschreiben. Die Dateigröße der encodeten Datei fällt derweil proportional zur Anzahl der Einzelbilder in einer GoP. Hierfür ließ sich keine genaue Funktion feststellen, da die Dateigröße stark vom gewählten Tiling-Layout abhängt.

Im Vergleich mit den anderen untersuchten Codecs schneidet die FIASCO-Erweiterung schlecht ab. H.264 schließt auf jeder Qualitätsstufe den Encoding-Prozess in unter zwei Sekunden ab und auch VP9 benötigt maximal eine Minute zum Encoden. Dagegen steht die minimale Encoding-Zeit von FIASCO beim Einsatz von einzelnen I-Frames bei 5 Minuten. Die Dateigröße der anderen Codecs liegt zwar beim verlustfreien Encoden deutlich über der der FIASCO-Dateien, sobald aber Verlustbehaftung erlaubt wird, erzielen die etablierten Codecs deutlich bessere Ergebnisse. Auch ist bei ihnen die Qualität des codierten Videos deutlich besser als bei FIASCO. Lediglich das mit einem CRF-Wert von 51 codierte H.264-Video hat eine schlechtere Qualität als die fraktal codierten Videos. FIASCO selbst kann durch den Einsatz von P-Frames Ergebnisse erzielen, die mit den moderaten Qualitätseinstel-

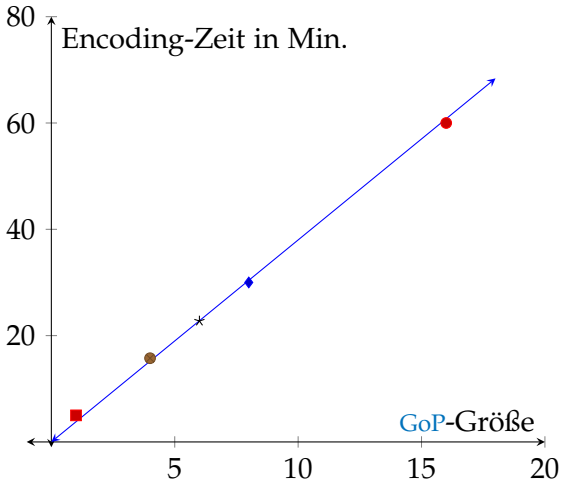

Abbildung 7.1: Wachstum der Funktion $f(x) = \frac{19x}{5}$ mit Messdaten zum Vergleich

lungen der anderen Codecs mithalten können, allerdings war es aufgrund der fehlerhaften Huffman-Codes nicht möglich, die so codierten Dateien zu decoden und ihre Qualität zu untersuchen.

Es hat sich bestätigt, dass der Einsatz von Fraktalen zum Codieren von Videodateien durchaus niedrige Kompressionsraten zur Folge haben kann. In den Messungen, die in dieser Arbeit durchgeführt wurden, hat sich aber gezeigt, dass die Idee, mehrere Einzelbilder mit denselben Fraktalen zu Encoden, um so die Redundanz mehrfach referenzierter Bildausschnitte zu verringern, im Vergleich zu den etablierten Videocodecs nicht erfolgreich war. Der hybride Ansatz der Videocodierung von FIASCO hat sich als deutlich mächtiger herausgestellt. Wahrscheinlich trägt auch das hohe Alter von FIASCO dazu bei, dass die Kompressionsraten in den hier angestellten Versuchsreihen unwirtschaftlich ausfallen. Dennoch hat sich die Möglichkeit, für die implementierte Erweiterung auf FIASCO aufzubauen, als sehr positiv für den Entwicklungsprozess herausgestellt. Ich sehe auch weiter Potenzial für sinnvolle zukünftige Anwendungen hybrider Codecs. Daher halte ich die weitere Erforschung des Anwendungsbereichs von Fraktalen für den Einsatz in der Videocodierung auch in Zukunft lohnend, möglicherweise auf Grundlage eines vollständig neu entwickelten, erneut auf einem hybriden Ansatz basierenden Codecs.

LITERATUR

[1] Adobe. *Unterstützte Dateiformate: Videodateiformate - Export*. 2021. URL: https://web.archive.org/web/20210129105807/https://helpx.adobe.com/de/premiere-elements/kb/supported-file-formats.html.

[2] Estefania Alcocer, Roberto Gutierrez, Otoniel Lopez-Granado und Manuel P. Malumbres. "Design and implementation of an efficient hardware integer motion estimator for an HEVC video encoder". In: *Journal of Real-Time Image Processing* 16.2 (2019), S. 547–557. ISSN: 1861-8200. DOI: 10.1007/s11554-016-0572-4.

[3] FFmpeg. *About FFmpeg*. URL: https://web.archive.org/web/20210209103611/https://ffmpeg.org/about.html.

[4] FFmpeg. *Muxing*. 2021. URL: https://web.archive.org/web/20210129155928/https://ffmpeg.org/doxygen/trunk/group__lavf__encoding.html.

[5] File-Examples. *file_example_MP4_480_1_5MG.mp4*. URL: https://file-examples-com.github.io/uploads/2017/04/file_example_MP4_480_1_5MG.mp4.

[6] Yuval Fisher. *Fractal Image Compression*. New York, NY: Springer New York, 1995. ISBN: 978-1-4612-7552-7. DOI: 10.1007/978-1-4612-2472-3. URL: https://books.google.de/books?id=bG_jBwAAQBAJ.

[7] Florian Freistetter. *Was sind fraktale Dimensionen?* 2009. URL: https://web.archive.org/web/20210205112456/https://scienceblogs.de/astrodicticum-simplex/2009/10/06/was-sind-fraktale-dimensionen/?all=1.

[8] "Discrete Cosine Transform (DCT)". In: *Encyclopedia of Multimedia*. Hrsg. von Borko Furht. New York: Springer-Verlag, 2006, S. 203–205. ISBN: 0-387-24395-X. DOI: 10.1007/0-387-30038-4_61.

[9] Ullrich Hafner. *FIASCO*. URL: https://github.com/l-tamas/Fiasco.

[10] Ullrich Hafner. *Low bit-rate image and video coding with weighted finite automata: Zugl.: Würzburg, Univ., Diss., 1999*. 1. Aufl. Berlin: Mensch & Buch Verl., 1999. ISBN: 3-89820-002-7.

[11] Alexey Kamenskiy. *argparse*. 2020. URL: https://github.com/akamensky/argparse.

[12] NVIDIA. *NVIDIA Video Codec SDK*. 2021. URL: https://web.archive.org/web/20210129125104/https://developer.nvidia.com/nvidia-video-codec-SDK.

[13] Robert Stocker. *Jpeg Compression: Jpeg Quantization*. URL: https://web.archive.org/web/20200716174356/http://www.robertstocker.co.uk/jpeg/jpeg_new_10.htm.

[14] Thomas Wiegand. “Source Coding: Part I of Fundamentals of Source and Video Coding”. In: *Foundations and Trends® in Signal Processing* 4.1-2 (2010), S. 1–222. ISSN: 1932-8346. DOI: 10.1561/2000000010.

[15] Thomas Wiegand. *Image and Video Coding I: Introduction*. 2021. URL: https://www.ic.tu-berlin.de/fileadmin/fg121/WS19-20_IVC-I/handouts/00-Introduction.pdf.

[16] Thomas Wiegand und Heiko Schwarz. “Video Coding: Part II of Fundamentals of Source and Video Coding”. In: *Foundations and Trends® in Signal Processing* 10.1-3 (2016), S. 1–346. ISSN: 1932-8346. DOI: 10.1561/2000000078.

[17] Bruno Zatt, Leandro M. de L. Silva, Arnaldo Azevedo, Luciano Agostini, Altamiro Susin und Sergio Bampi. “A reduced memory bandwidth and high throughput HDTV motion compensation decoder for H.264/AVC High 4:2:2 profile”. In: *Journal of Real-Time Image Processing* 8.1 (2013), S. 127–140. ISSN: 1861-8200. DOI: 10.1007/s11554-011-0216-7.

[18] linux.die.net. *pnm(5) - Linux man page*. URL: https://web.archive.org/web/20210208123136/http://linux.die.net/man/5/pnm.

[19] linux.die.net. *ppm(5) - Linux man page*. URL: https://web.archive.org/web/20210208124203/https://linux.die.net/man/5/ppm.

[20] speedtest.net. *Speedtest Global Index: Ranking mobile and fixed broadband speeds from around the world on a monthly basis*. Hrsg. von speedtest.net. 2021. URL: https://web.archive.org/web/20210129093529/https://www.speedtest.net/global-index.

[21] webmproject. *VP9 Encoding Guide*. 2021. URL: https://web.archive.org/web/20210218093928/http://wiki.webmproject.org/ffmpeg/vp9-encoding-guide.