DEUTSCHE
GESELLSCHAFT FÜR
ZERSTÖRUNGSFREIE
PRÜFUNG E.V.

ZfP-Sonderpreis der DGZfP beim Landeswettbewerb Jugend forscht

HAMBURG

Verbesserung der Bildqualität von Rastergrafiken

Lasse Schuirmann

Schule:

Albert-Schweitzer-Gymnasium
22337 Hamburg

Jugend forscht 2009

jugend forscht

44. Landeswettbewerb 2009

Verbesserung der Bildqualität von Rastergrafiken

Arbeit von Lasse Schuirmann
vorgelegt im Rahmen des Wettbewerbs
„Jugend forscht“

Albert-Schweitzer-Gymnasium
22337 Hamburg
März 2009

Datenblatt

Projekt:

Titel:	Verbesserung der Bildqualität von Rastergrafiken
Sparte:	Jugend forscht
Bundesland:	Hamburg
Patenunternehmen:	Airbus
Fachgebiet:	Mathematik/Informatik

Teilnehmer:

Geschlecht:	Männlich
Nachname:	Schuirmann
Vorname:	Lasse
Straße:	Struckholt 4
PLZ, Ort:	22337 Hamburg
Telefon:	040/79306891
E-Mail:	lasse.schuirmann@gmx.de
Geburtsdatum:	22.08.1994
Klassenstufe:	9

Schule:

Name:	Albert-Schweitzer-Gymnasium
Schultyp:	Gymnasium
Straße:	Struckholt 27
PLZ, Ort:	22337 Hamburg
E-Mail:	Albert-Schweitzer-Gymnasium@bsb.hamburg.de
Internet:	http://asg-hh.de/

Projektbetreuer:

Nachname, Vorname:	Boy-Will, Ursula
Schule/Betrieb:	Albert-Schweitzer-Gymnasium
Telefon:	040/5317254
E-Mail:	u.boywill@web.de

Folgende Personen haben mich bei meiner Arbeit unterstützt:
Ursula Boy-Will, Amelie Kallmann, Prof. Dr. Friedrich Mayer-Lindenberg,
Prof. Dr. Wolfgang Mackens, Prof. Dr. rer.-nat. habil. Ralf Möller,
Prof. Dr.-Ing. Rolf-Rainer Grigat, Juliane Becker, Dennis Bludau

Inhaltsverzeichnis

1. Kurzfassung

Ich möchte herausfinden, ob man bei digitalen Bildern mit schlechter Qualität per Software die Qualität verbessern kann. Dadurch könnte man mit schlechteren Kameras bessere Bilder oder Videos produzieren, als mit geringfügig besseren, deren Bilder bzw. Videos nicht aufgewertet werden. Sogar der Treppeneffekt von Fernsehbildern könnte verhindert werden.

Vorgehensweise:
1. Bild (bei Videos erst in Einzelbilder zerlegen) in Vektorformen zerlegen und anschließend wieder in eine qualitativ höhere Rastergrafik konvertieren oder
2. das Bild in Regionen einteilen und dann auf alle Regionen einzeln eine Kantenglättung anwenden.
Diese Vorgehensweisen werden analysiert, erprobt bzw. errechnet und dann im Blick auf ihre Umsetzbarkeit reflektiert.

2. Aufgabenstellung

2.1. Einführung

Die folgende schriftliche Ausarbeitung erläutert Vorgehensweisen zur nachträglichen Aufwertung von Rastergrafiken durch Software.
Gegenwärtig müssen Nachrichtensender Beträge in Höhe mehrerer Millionen Euro in neue HD-fähige Kameras investieren, da die Umstellung von der heutigen Standard Fernsehauflösung (PAL; 768 x 576 Pixel) auf den High Definition Standard (HD; 1920 x 1080 Pixel) bis 2010 erfolgen soll [1]. Ließe sich die Bildqualität per Software nachträglich von PAL auf den High Definition Standard aufwerten, wäre keine neue Hardware nötig, so dass ohne größeren Kostenaufwand allgemein eine höhere Bildqualität erreicht werden kann, da statt Millionen Kameras nur eine Software sowie ein leistungsfähiges System beschaffen werden muss.

2.2. Aktuelle Verfahren

Heutzutage werden zur Verbesserung von Bildqualität verschiedene Verfahren angewendet:

2.2.1. Das Antialiasing (engl.: Kantenglättung): [2] [3]

Bei dem Antialiasing wird jeder Pixel neu errechnet: Die Farbwerte der umliegenden acht Pixel werden zu dem Farbwert des eigentlichen Pixels hinzu addiert. Das Ergebnis wird durch neun (also der Anzahl aller Pixel, aus denen der entsprechende Pixel errechnet wird) geteilt. Dieses Ergebnis ist der neue Farbwert des Pixels, also der Durchschnittsfarbwert aller umliegenden und des zu bearbeitenden Pixels. Der Vorteil dieser Methode ist, dass sie in der Programmierung sehr einfach und in der Ausführung relativ schnell ist. Man sieht gut, dass die Linien bei dem bearbeiteten Bild keine „Treppen“ mehr enthalten. Sie hat aber auch große Nachteile:

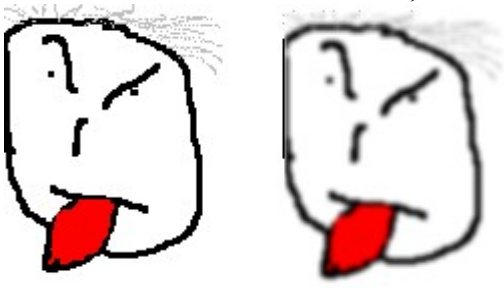


Abbildung 1: Handgemaltes Bild vor (links) und nach (rechts) Verwendung der Antialiasing Methode

1. Das gesamte Bild wird unscharf.
2. Die Farben verlieren an Intensität.

2.2.2. Entrauschen (durch z. B. Median Filter): [4]

Der Median Filter sortiert die Grauwerte der umliegenden Pixel und weist dem zu berechnenden Pixel den mittleren Wert zu.

Beispiel:
Der Grauwert der neun betreffenden Pixel (der zu bearbeitende Pixel und die acht umliegenden) wird in einen „Array“ geschrieben. Ein Array (engl.: Bereich) ist ein Datenfeld, das mehrere Variablen enthält. In diesem Fall ist ein Array mit neun Variablen nötig, da die Grauwerte je eines Pixels in je eine Variable geschrieben werden:

Variable	**Pixel 1**	**Pixel 2**	**Pixel 3**	**Pixel 4**	**Pixel 5**	**Pixel 6**	**Pixel 7**	**Pixel 8**	**Pixel 9**
Grauwert	119	184	143	91	218	157	184	167	205

Nun wird der Array in aufsteigender Reihenfolge sortiert:

Variable	**Pixel 4**	**Pixel 1**	**Pixel 3**	**Pixel 6**	**Pixel 8**	**Pixel 2**	**Pixel 7**	**Pixel 9**	**Pixel 5**
Grauwert	91	119	143	157	167	184	184	205	218

Der aktuelle Pixel bekommt jetzt den mittleren Farbwert. - Der Farbwert des Pixels, der nach der Sortierung nach Grauwerten in der Mitte lag, also in diesem Fall Pixel 8.

Abbildung 2: Verrauschtes Bild vor (links) und nach (rechts) Verwendung des Medianfilters
http://upload.wikimedia.org/wikipedia/en/4/41/Median_filter_example.jpg

Ein Entrauschen ruft häufig ebenfalls Unschärfe hervor und bezweckt, dass störende Unreinheiten, häufig hervorgerufen durch fehlerhaft arbeitende Aufnahmesensoren, aus dem Bild verschwinden.

2.2.3. Die Kontrastverstärkung: [5]
Durch eine Kontrastverstärkung werden alle Farbunterschiede verstärkt.

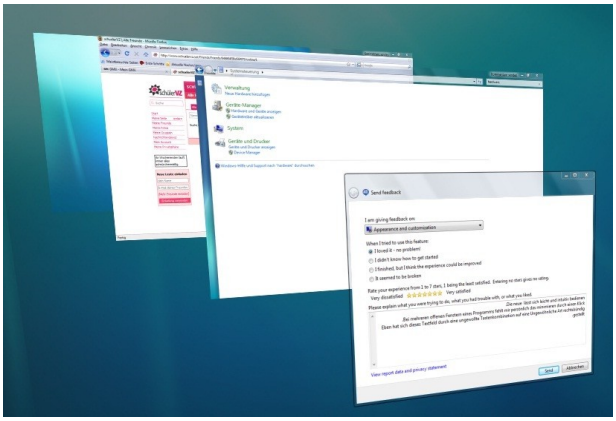


Abbildung 3: Screenshot von Windows 7 vor Kontrastverstärkung

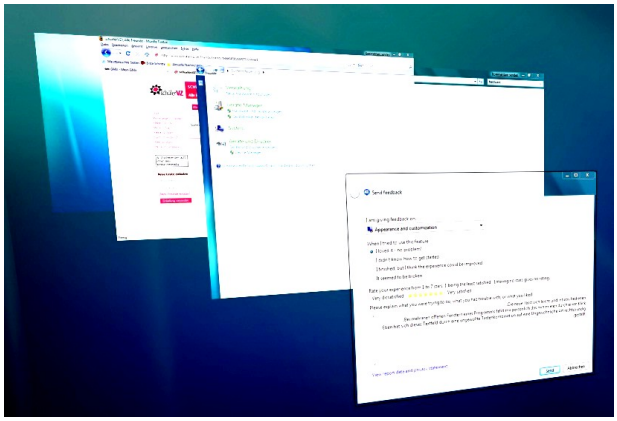


Abbildung 4: Screenshot von Windows 7 nach Kontrastverstärkung

Der Nachteil der Methode ist ein unnatürliches Aussehen des Bildes, sowie (je nach Methode) ein Detailverlust nach der Bearbeitung.

(Das Menschliche Auge arbeitet ähnlich: [6] [7]

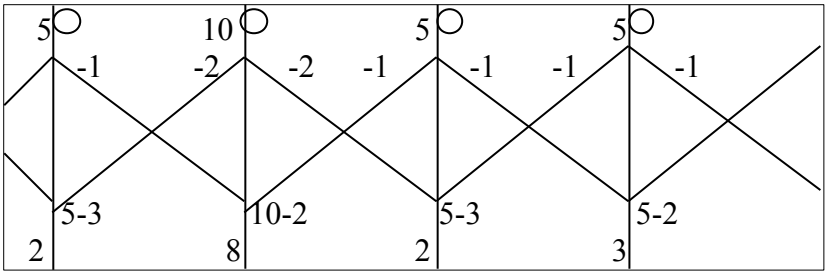
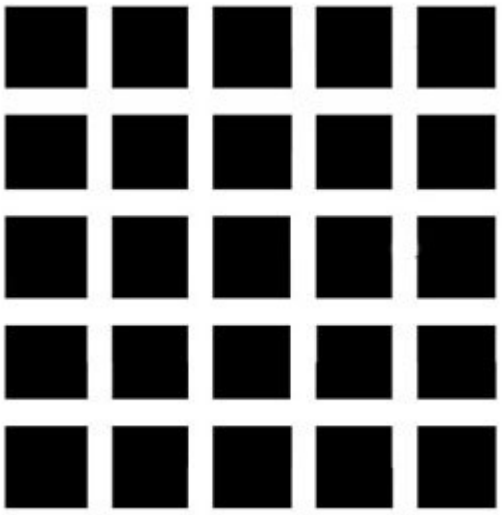


Abbildung 5: Grafik zur Veranschaulichung der lateralen Inhibition



Jede Sinneszelle gibt einen Teil, im Beispiel ein Fünftel, ihres Wertes an die Nachbarzelle weiter.
Dieser Wert wird von dem der Nachbarzelle subtrahiert. So entstehen stärkere Kontraste.
In dem nebenstehenden Bild zeigt sich der Effekt:
Dort wo sich die weißen Linien kreuzen, glaubt man, schwarze Punkte zu sehen.
Dies ist eine Folge des eben beschriebenen Vorgangs, der „Laterale Inhibition“ oder „Laterale Hemmung“ genannt wird.

Abbildung 6:Optische Täuschung
http://www.hsg-kl.de/faecher/bio/auge/bilder/Image11.gif)

2.2.4. Andere Optimierungsverfahren:
Es existieren noch andere Bildoptimierungsverfahren wie zum Beispiel die Gamma Korrektur [8].

3. Durchführung

3.1. Eigener Lösungsweg
Als Rastergrafik wird eine Grafik bezeichnet, bei der jeder Punkt der Grafik als Farbwert gespeichert wird. Diese Punkte nennt man „Pixel“. Bei einer Vektorgrafik sind, anders als bei einer Rastergrafik, nicht die einzelnen Pixel definiert, sondern die Formen, Positionen und Farben [9].

(Beispiel: „Kreis(PosX=50; PosY=50; Rad=r; LinBr=10; Col=“Black“)“.
„PosX“ (für „Position X“) wäre die X-Koordinate des Mittelpunktes des Kreises, „PosY“ (für „Position Y“) wäre die Y-Koordinate. Der Radius würde in dem Beispiel als „Rad“ (für „Radius“) bezeichnet, während „LinBr“ (für „Linienbreite“) die Breite der Linie des Kreises darstellt. Die Farbe wird hier mit „Col“ (für „Color“) mit einer englischen Farbbezeichnung definiert.)

Weil eine Vektorform immer generiert wird, kann man bei Vektorgrafiken ohne Qualitätsverlust zoomen (=vergrößern). Daher habe ich mir gedacht, dass man vielleicht eine Rastergrafik vektorisieren (=in eine Vektorgrafik umwandeln) kann, um diese Vektorgrafik dann in höherer Auflösung zu speichern.

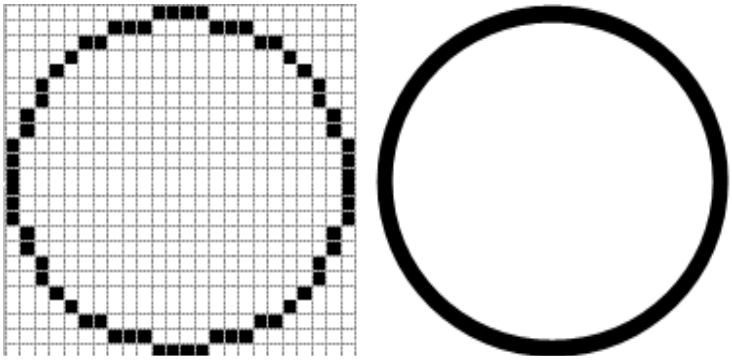


Abbildung 7: Ein vergrößerter Kreis als Rastergrafik (links) und als Vektorgrafik (rechts)

1. Die Grafik ist in einem beliebigen Bildformat (häufig JPEG, BMP oder PNG) als Rastergrafik gespeichert.
2. Die Grafik wird von einem Programm vektorisiert.
   Das Ergebnis wäre dann in diesem Fall ungefähr dies:
   Kreis(PosX = 6; PosY = 6; Rad = 12; LinBr = 1; Col = “Black”).
3. Die erkannten Vektorgrafiken werden als höher auflösende Rastergrafik gezeichnet bzw. gespeichert.

Die Schritte 1 und 3 sind relativ leicht in einem Programm zu realisieren, da es in vielen Programmiersprachen schon vordefinierte Komponenten gibt, mit denen man Bilder einfach laden, zeichnen und speichern kann.
Bei Schritt 2 („Erkennung der Vektorformen“) treten jedoch massive Probleme auf:

1. Aufgrund der Tatsache, dass Digitalkameras oft bis zu 16 Millionen (und mehr) verschiedene Farben speichern, ist es immens unwahrscheinlich, dass sich eine Vektorform findet, die in allen Punkten mit der restlichen Form übereinstimmt. Kleinere Abweichungen sollten toleriert werden, da man sonst fast jedes einzelne Pixel als eigene Vektorform vektorisieren müsste.
Dies hätte zur Folge, dass im Endeffekt das neue, vergrößerte Bild, genauso wie vorher aussieht.

2. Durch diese Toleranz verliert das Bild an Details.

3. In einem Foto wird nur in sehr wenigen Fällen eine Vektorform auftreten, welche nicht in irgendeiner Weise verzerrt ist. Minimale Verzerrungen sollte ein derartiges Programm tolerieren, größer verzerrte Vektorformen würden in einzelne Linien und/oder Bögen aufgeteilt.
Beide Methoden rufen einen Detailverlust hervor.

4. Verschiedene Farben sind innerhalb einer Vektorform nicht möglich.

4.1. Das führt wegen der Toleranz zu Detailverlust.

4.2. Es muss in dem Programm eine komplexe Funktion zu jeder beliebigen Vektorform geben, die eine Art Linien- bzw. Flächenfüllung zur Verfügung stellt. Diese würde beispielsweise in einer Linie jedem Pixel, anstelle von Schwarz oder einer anderen konstanten Farbe, individuell genau die Farbe des entsprechenden Pixels einer anderen Linie zuweisen, nämlich die Linie in der Rastergrafik, die als Vektorform erkannt wurde. Dabei ist zu beachten, dass bei einer Vergrößerung eine Art Antialiasing mit der entsprechenden Linie der Rastergrafik durchgeführt wird, man die neu hinzugekommenen Pixel also aus den Nachbarpixeln errechnet (genaueres siehe unter „Aktuelle Verfahren“; Seite 5). Die dabei entstehende Unschärfe sollte in diesem Fall kein Problem sein, da es

immer noch eine Linie ist, die keine harten Farbübergänge hat. (Sie ist ja mit nur ein wenig Toleranz als Linie erkannt worden.)

5. Systemanforderungen spielen ebenfalls eine große Rolle, da die Methode, um bei Echtzeitvideos angewendet werden zu können, schnell genug sein muss, um flüssig Bilder verbessern und übertragen zu können.
Bereits existierende Vektorisierungs-Programme brauchten über 10 Sekunden für ein Bild.
Eine Bildfrequenz von 1/10 Bildern pro Sekunde wäre kaum effektiv.

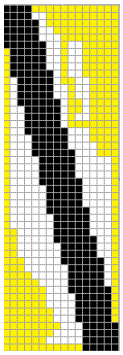
Es ist also eine komplexe Programmierung notwendig um eine gute Vektorisierung zu realisieren.
Man könnte versuchen, mit der so genannten „Region Growing“ Methode Vektorformen zu erkennen.

Dabei werden von einem Pixel aus die Nachbarpixel auf Gleichheit bzw. hohe Ähnlichkeit überprüft [10] [11].
Die erkannten „Regionen“ könnte man dann verschiedenen Vektorformen zuordnen.
Tests mit schon vorhandenen Vektorisierungs-Programmen haben keine passablen Ergebnisse hervorgebracht:

Abbildung 8: *Abbildung 9:*
Originalbild (Ausschnitt aus einem Screenshot) → Vektorisiertes Bild

Daher habe ich ein Modell für eine weiterführende Verfahrensweise aufgestellt, eine Kombination aus Region Growing, Antialiasing und Vektorisierung:

1. Die Bildauflösung der Rastergrafik wird erhöht.
2. Regionen erkennen mit Region Growing:
   2.1. Man vergleicht einen Pixel jeweils mit den umliegenden Pixeln, die noch keiner Region angehören.
   2.2. Wenn einer oder mehrere der verglichenen Pixel ungefähr die gleichen Farbwerte besitzen (es würde ein Toleranzwert definiert werden), wird in einem Array (Erklärung: Siehe „Aktuelle Verfahren“; Seite 6) gespeichert, dass der oder die Pixel der aktuell gesuchten Region angehören. Anschließend wird 2.1. mit den als ähnlich identifizierten Pixeln durchgeführt. Falls es keine derartigen Pixel gibt, wird nach der nächsten Region gesucht.
   2.3. Das beschriebene Verfahren wird wiederholt, bis alle Pixel einer Region angehören.
3. Der Rand von jeder Region wird so tolerant vektorisiert, dass
   3.1. keine oder möglichst wenige Details verloren gehen sollten;
   3.2. die Vektorisierung unter möglichst genauer Beachtung der Farbwerte verläuft, um 3.1. zu entsprechen, aber dennoch ein wenig tolerant ist;
   3.3. die vektorisierten Formen zusammen ein Polygon ergeben;
   3.4. die Anforderungen in der Aufzählung „Erkennung der Vektorformen“ auf Seite 8 f. erfüllt werden.
4. Auf die Fläche der Region wird Antialiasing angewendet. (Es gibt dort nur minimal scharfe Übergänge; es entsteht also auch nur minimale generelle Unschärfe.)
5. Die Ränder der Regionen werden durch die in 3. erkannten Vektorformen ersetzt.
6. Sonstige Verbesserungsmaßnahmen (siehe „Aktuelle Verfahren“; Seite 5 f.).



Bei der nebenstehenden Grafik wird die schwarze Linie als Region erkannt.
Auf die Fläche der Linie wird Antialiasing angewendet und die Ränder werden durch die vorher erkannten Vektorformen ersetzt.

Abbildung 10: Hand gezeichnete Grafik zur Veranschaulichung einer möglichen Funktionsweise

3.2. Umsetzung
Ein derartiges Programm ist in folgende Module einzuteilen:
1. Region Growing
2. Kantenglättung/Antialiasing
3. Vektorisierung
4. Sonstige Verbesserungsmaßnahmen

Folgende Module wurden während meines Projektes implementiert (=programmiert):

1. Region Growing:
Erklärung: Siehe „Weiterführende Theorie“ auf Seite 9; Punkt 2 und Unterpunkte.

2. Kantenglättung/Antialiasing:
Funktionsweise siehe „Aktuelle Verfahren“; Seite 5.
Um das Verhältnis zwischen Unschärfe und Verminderung des Treppeneffekts regulieren zu können, wurde eine Gewichtung implementiert:
Einige Pixel werden also mehr in das Ergebnis eingerechnet als andere.
Dies kann dadurch erreicht werden, dass der Farbwert einiger (oder aller) Pixel mehrere Male zu dem vorherigen Zwischenergebnis dazu addiert werden. Um den Durchschnittsfarbwert zu erhalten, muss folglich nicht mehr durch neun geteilt werden, sondern durch die Summe der unterschiedlich gewichteten Pixel.

4. Sonstige Verbesserungsmaßnahmen:
Genauere Erläuterungen: Siehe „Aktuelle Verfahren“, Seite 5 f.
Implementiert wurden:
- Antialiasing [2][3]
- Median-Filter [4]
- Kontrastverstärkung [5]
- Gamma-Korrektur [8]
- Lineare Spreizung des Grauwertebereichs [12]
- Nichtlineare Spreizung des Grauwertebereichs [12]

Um Systemressourcen voll auszunutzen und den Vorgang zu beschleunigen, ist das Programm als Multithreading-Anwendung aufgebaut [13]. (Es führt also mehrere Vorgänge gleichzeitig aus.) Dabei lastet das Programm bis zu vier Prozessorkerne voll aus, bis der Vorgang abgearbeitet ist. Des Weiteren benötigt das Programm ungefähr 10MB Arbeitsspeicher im laufenden Betrieb. Um das Programm auch mit größeren Bildern verwenden zu können, wurden als Speichertyp für die Regionen 64Bit Variablen vom Typ Integer benutzt. Dadurch können in einem Bild theoretisch bis zu 9223372036854775808 (2^{63}) Regionen erkannt werden.

Als Lösungsansatz für die Vektorisierung habe ich zunächst einen Hough Algorithmus programmiert. Dieser dient zur Geradenerkennung.
Das Prinzip des Hough Algorithmus ist im Grunde einfach: [14]
Der Algorithmus probiert für jeden Pixel jede mögliche Gerade aus und trägt in einem Array deren Position (im Bogenmaß) ein. Wenn man nun eine Gerade sucht, kann man diese ausmachen, indem man sich einfach die Position heraussucht, die am häufigsten verzeichnet ist. Das Problem dabei ist, dass erstens keine Geraden, sondern Strecken gesucht sind und zweitens das Programm nicht weiß, wie viele Strecken gefunden werden sollen.

Als Alternative zu der Vektorisierung habe ich überlegt, mit Splines zu arbeiten, also die Seitenränder durch eine Kurve zu verbinden [15] [16] [17]. Splines sind allerdings nicht optimal, da diese nicht senkrecht verlaufen können, was bei einer Form innerhalb eines Bildes mit sehr hoher Wahrscheinlichkeit mindestens ein mal vorkommt. Eine Anpassung der Methode, um auch senkrechte Abschnitte zu begradigen, würde neue Probleme aufwerfen, was die Komplexität in dem Maße steigern würde, dass es in dem gegebenen Zeitrahmen für mich nicht umsetzbar ist.

Die als komplett bezeichneten Funktionen wurden auf mehrere Bilder angewendet:

Ausschnitt eines Beispielbildes von Microsoft © Windows ® Vista™

Abbildung 11, 12 und 13: Links ist das Originalbild zu sehen, in der Mitte die daraus resultierende Vergrößerung und schließlich rechts die vergrößerte und optimierte Version

4. Ergebnis

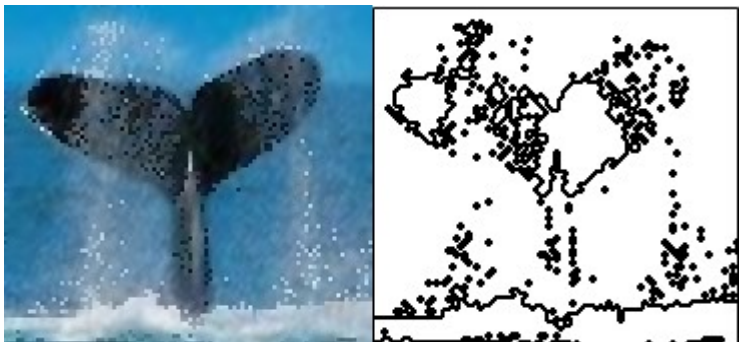


Abbildung 13: Automatisch optimiertes Bild (links) und Ränder der in diesem Bild erkannten Regionen (rechts)

Der Optimierungsvorgang dauerte auf einem Intel® Core™2 Duo Prozessor, welcher auf 2.0 Ghz getaktet war, ca. 5 Sekunden. Erst wurden die Regionen erkannt. Nach der Vergrößerung wurde mit den Regionen die Antialiasing Prozedur durchgeführt.
Sonstige Verbesserungsmaßnahmen wurden nicht angewendet, da diese keine neuen Verfahren darstellen.

Das optimierte Bild ist leicht unscharf, wird aber von 9 von 10 Testpersonen dem normal vergrößerten Bild vorgezogen.
Auf dem optimierten Bild ist der Treppeneffekt nur noch an wenigen Stellen zu erkennen.
Die Kanten der einzelnen Regionen sind jedoch, aufgrund der fehlenden Vektorisierung, noch nicht gut behandelt worden.
Bei zu großen Bildern gibt das Programm häufig eine Fehlermeldung aus, welche besagt, dass irgendein Wert zu groß für irgendeine Variable ist. Die Umstellung auf 64 Bit Variablen geschah erst nach der Programmierung des Region Growing Algorithmus. Es ist wahrscheinlich, dass bei der Umstellung eine oder mehrere relevante 32 Bit Variablen übersehen worden sind.

5. Diskussion

Bei der Konfiguration der Toleranzwerte meines Region Growing Algorithmus fiel mir auf, dass verschiedene Bilder unterschiedliche Toleranzwerte brauchten.
Wählt man einen zu hohen Toleranzwert, wird das Bild zu unscharf. Bei einem zu niedrigen Toleranzwert zeigt die Optimierung nur wenig Wirkung. Daher habe ich überlegt, dass das Programm eine Intelligenz entwickeln könnte und bin auf „Künstliche neuronale Netze“ gestoßen [18] [19]. Ein künstliches neuronales Netz könnte den Toleranzwert anhand der Durchschnittshelligkeit ermitteln. Hier sehe ich weitere Forschungsmöglichkeiten.

Der Median Filter meines Programms zeigt keine Wirkung. Es wäre sinnvoll, die Implementierung ein weiteres Mal genauer zu überprüfen und auszubessern, um seine Funktionalität zu gewährleisten.
Da mir zur Zeit noch keine weiteren Informationen über einen möglichen Fehler vorliegen, ist es mir momentan weder möglich, den Umfang der nötigen Korrekturen zu bestimmen, noch zu sagen, in welcher Weise er ausgebessert werden muss.
Um das Programm gut nutzen zu können, wäre eine weitere Geschwindigkeitsoptimierung notwendig. Des Weiteren wäre es angebracht, mein Programm durch die Vektorisierung zu vervollständigen.

Es wäre von Vorteil, wenn das Programm für jedes Bild automatisch erkennen würde, welche Verbesserungsmaßnahmen sinnvoll sind, um die entsprechenden Maßnahmen dann einzusetzen.

Im Grunde geht es darum, das Bild für die Wahrnehmung des Menschen zu optimieren.
Das Originalbild ist immer am aussagekräftigsten, wenn ausschließlich das zu optimierende Bild als Ausgangspunkt für das neue Bild benutzt wird/werden kann, da das optimierte Bild in diesem Fall ausschließlich aus dem Originalbild resultiert.
Eine wirkliche Verbesserung könnte man durch eine Art „Objekterkennung“ bewirken:
Hierbei würde beispielsweise der Schrank auf Abbildung 14 als derselbe Schrank wie in Abbildung 15 erkannt werden. Der Schrank in Abbildung 14 könnte unter Zuhilfenahme des Schrankes auf Abbildung 15 und anderen Bildern, die denselben Schrank abbilden, detailreicher abgebildet werden.

Abgesehen von den Systemanforderungen, die dieses Programm stellen würde, ist die Programmierung außerordentlich schwierig:
Es gibt viele Abweichungen, die bei der Erkennung toleriert und bei der Optimierung korrigiert werden müssten, wie zum Beispiel die Belichtung, die Position sowie die Entfernung.
Es ist sehr unwahrscheinlich, dass sich zwei Bilder finden, in denen der selbe Schrank in exakt der gleichen Position und Belichtung zu finden ist.

Abbildung 14: Fotografie eines Schrankes (1)

Abbildung 15: Fotografie eines Schrankes (2)

Für ein Programm wäre es schwierig, diese beiden Schränke als die selben zu identifizieren.
Dies könnte beispielsweise durch neuronale Netze realisiert werden.
Damit Abbildung 15 nun zur Optimierung von Abbildung 14 verwendet werden kann, muss ein derartiges Programm:

1. die Belichtung von Abbildung 15 anpassen.
2. den Schrank aus Abbildung 15 extrahieren (= herausschneiden), wobei beachtet werden muss, dass störende Bildelemente (beispielsweise ein Ball, der vor dem Schrank in einem der beiden Bilder herunterfällt) entfernt werden müssen.
3. den extrahierten Schrank so rotieren und vergrößern bzw. verkleinern, dass sich die Schränke aus Abbildung 14 und Abbildung 15 übereinander legen lassen.
4. überprüfen, welche Details in Abbildung 15 besser dargestellt werden als in Abbildung 14, oder in Abbildung 14 nicht vorhanden sind.
5. die in 4. festgestellten Details in Abbildung 14 entsprechend ergänzen/verbessern.

Dieser Vorgang ist so kompliziert, dass ich nicht glaube, dass ein derartiges Programm heutzutage schon in ausgereifter Form existiert.

6. Ausblick

Ein Bildoptimierungs-Programm könnte eine Texterkennung beinhalten, die auf einem künstlichen neuronalen Netz basiert: [18] [19]
Ein paar Standardinformationen werden der Texterkennung gegeben. (Man speist beispielsweise ein „A“ in das Neuronale Netz ein und gibt das Ergebnis („A“) vor. Es sollten alle Zeichen unseres Alphabets in mehreren Varianten eingespeist werden.)
Wenn der Text nicht mit x-prozentiger Sicherheit (oder mehr) erkannt wird, meldet das Programm dies dem Benutzer und der Benutzer korrigiert die Fehler. Dadurch wird das Ergebnis beim nächsten Mal wahrscheinlich besser, wenn das Programm das Gelernte abspeichert.
Die gelernten Informationen könnten, mit Zustimmung des Benutzers, im Internet auf einen Server gelegt werden und auch andere Benutzer würden davon profitieren.
Ein weiterer Vorteil eines neuronales Netzes gegenüber der traditionellen künstlichen Intelligenz ist, dass es auch ohne Benutzer lernen kann, indem es die falsch liegenden künstlichen Neuronen den richtigen anpasst. (Die richtigen Neuronen werden als die definiert, die in der Überzahl sind. Wenn auch bei der Überzahl eine zu große Unsicherheit existiert, wird keine Anpassung vorgenommen.)

7. Danksagung

Mein besonderer Dank gilt meiner Lehrerin Ursula Boy-Will, die mich zu dieser Arbeit angeregt und mit sehr viel Engagement unterstützt hat.
Des Weiteren bedanke ich mich bei meinen Eltern, die mich bei meiner Arbeit begleitet haben.

8. Quellen

[1] http://www.paradiso-design.net/videostandards.html

[2] http://alt.3dcenter.org/artikel/anti-aliasing/index02.php

[3] http://www2.biochemtech.uni-halle.de/lernwww/tif.htm

[4] http://www.informatik.uni-osnabrueck.de/sstiene/masterthesis/node20.html

[5] http://www.vision-academy.org/mv_wbuch/np/F8DCA9BBA7E777B9C1256C1C0023AD76.htm

[6] http://www.hsg-kl.de/faecher/bio/auge/inhibition.htm

[7] http://lexikon.meyers.de/wissen/laterale+Inhibition

[8] www.poeschek.at/files/publications/gamma.pdf

[9] http://www.gif-bilder.de/begriffe/vektorgrafik.html

[10] http://www.geoinformatik.uni-rostock.de/einzel.asp?ID=1012414638

[11] http://www.cs.cf.ac.uk/Dave/Vision_lecture/node35.html

[12] https://www.mi.fh-wiesbaden.de/~schwan/Vorlesungen/RIP/Skripte/RIPUS3.pdf

[13] http://www.delphi-treff.de/tutorials/objectpascal/threads/

[14] http://page.mi.fu-berlin.de/alt/vorlesungen/sem04/10_6Die%20Hough.doc

[15] http://lexikon.meyers.de/wissen/Spline

[16] http://www.itwissen.info/definition/lexikon/Spline-spline.html

[17] http://www.geoinformatik.uni-rostock.de/einzel.asp?ID=1595

[18] http://fbim.fh-regensburg.de/~saj39122/NN/index.html

[19] www.u-helmich.de/inf/NeuronaleNetze.pdf