Nicole Möstel

Bestimmung des Ähnlichkeitsgrades zweier Bilder

eingereicht als

BACHELORARBEIT

an der

HOCHSCHULE MITTWEIDA

UNIVERSITY OF APPLIED SCIENCES

Fachbereich Informatik

Mittweida, 16. September 2011

Hochschulbetreuer: Prof. Dr.-Ing. M. Geißler
Firmenbetreuer: Dipl.-Ing. Torsten Haß, Typo3-Beratung

Bibliographische Beschreibung:

Möstel, Nicole:
Bestimmung des Ähnlichkeitsgrades zweier Bilder. - 2011. - 88 S. Mittweida, Hochschule Mittweida, Fachbereich Informatik, Bachelorarbeit, 2011

Referat:

Das Thema der Bachelorarbeit besteht in der Auswahl, Konzeption und prototypischen Implementierung eines geeigneten Verfahrens zur Bestimmung des Ähnlichkeitsgrades zweier Bilder. Im ersten Schritt wird eine Analyse aller denkbaren Ähnlichkeitsvariationen durchgeführt. Anhand der ermittelten Ergebnisse werden Ähnlichkeitsgrade definiert. Des Weiteren werden bereits existierende Programme, die dieses Problem lösen, aufgezeigt und miteinander verglichen. Im Anschluss werden Algorithmen vorgestellt, die für die Implementierung des Prototyps verwendet werden. Als Ergebnis der Bachelorarbeit entsteht ein Prototyp, welcher Bilder miteinander vergleicht und ihren Ähnlichkeitsgrad ausgibt.

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Motivation 1
1.2 Szenario 2
1.3 Aufgaben und Ziel 2
1.4 Gliederung der Arbeit 2

2 Grundlagen **3**

2.1 Bildverarbeitung, Bildbearbeitung, Computergrafik 3
2.2 Bildanalyse im Gegensatz zu Mustererkennung und Computer Vision 3
2.3 Menschliches und maschinelles Sehen 4
2.4 Grundlagen digitaler Bilder 5
2.4.1 Arten von digitalen Bildern 5
2.4.2 Eigenschaften von Bildern 6
2.4.3 Dateiformate 9
2.5 Histogramme 10
2.5.1 Definition eines Histogramms 10
2.5.2 Interpretation eines Histogramms 11

3 Analyse bereits existierender Bildvergleich-Programme **13**

3.1 TinEye 13
3.1.1 Handhabung 13
3.1.2 Analyse der Suchergebnisse 14
3.1.3 Funktionsweise 15
3.1.4 Fazit 16
3.2 GazoPa 16
3.2.1 Handhabung 16
3.2.2 Analyse der Suchergebnisse 17
3.2.3 Funktionsweise 18
3.2.4 Fazit 19
3.3 Anti-Twin 19
3.3.1 Handhabung 20
3.3.2 Analyse der Suchergebnisse 21
3.3.3 Funktionsweise 24
3.3.4 Fazit 25

3.4 PictureRelate . . . 25
3.4.1 Handhabung . . . 26
3.4.2 Analyse der Suchergebnisse . . . 28
3.4.3 Funktionsweise . . . 28
3.4.4 Fazit . . . 28
3.5 Zusammenfassung . . . 28

4 Problem- und Anforderungsanalyse 30
4.1 Problemstellung: Wann sind Bilder ähnlich? . . . 30
4.2 Konzeption geeigneter Algorithmen . . . 32
4.3 Anforderungen an den Prototyp . . . 33
4.4 Analyse verschiedener möglicher Programmiersprachen . . . 34

5 Lösungskonzeption 35
5.1 Definition von Ähnlichkeitsgraden . . . 35
5.1.1 Ähnlichkeitsgrad 1 . . . 35
5.1.2 Ähnlichkeitsgrad 2 . . . 36
5.1.3 Ähnlichkeitsgrad 3 . . . 39
5.1.4 Ähnlichkeitsgrad 4 . . . 42
5.2 Steigerung der Verarbeitungsgeschwindigkeit durch Verwendung von Threads 42
5.3 Auswahl der Programmiersprache und Frameworks . . . 42

6 Implementierung 44
6.1 Darstellung des statischen Aufbaus anhand von Klassendiagrammen . . . 44
6.2 Dynamischer Ablauf – Anwendungsfall-, Aktivitäts- und Sequenzdiagramme 47
6.3 Implementierungsdetails . . . 51
6.3.1 Start des Programms . . . 51
6.3.2 Das Hauptfenster . . . 52
6.3.3 Auswahl eines `Comparers` durch den `ImageComparer` . . . 53
6.3.4 Der `CompareThread` . . . 56
6.3.5 Die Klasse `AbstractComparer` . . . 59
6.3.6 Die Klasse `IdenticComparer` . . . 60
6.3.7 Die Klasse `DCTComparer` . . . 61
6.3.8 Die Klasse `TemplateMatchingComparer` . . . 66
6.3.9 Die Klasse `ColorComparer` . . . 69

7 Zusammenfassung und Ausblick **73**
7.1 Was wurde erreicht? 73
7.2 Mögliche Verbesserungen und Erweiterungen 74

Abbildungsverzeichnis

2.1 Grundlagen: Optische Täuschungen 4
2.2 Grundlagen: Beispiele für maschinelle Bildverarbeitung 5
2.3 Grundlagen: Arten von Bildern 6
2.4 Grundlagen: Auflösung und Bildgröße 7
2.5 Bildkoordinaten im Koordinatensystem 8
2.6 Grundlagen: Histogramm 10
2.7 Grundlagen: Erkennen von Belichtungsfehlern im Histogramm 12
3.1 TinEye: Suchergebnisse 14
3.2 TinEye: Vergleich der Suchergebnisse 15
3.3 GazoPa: Suchergebnisse 17
3.4 GazoPa: Fehlerhafte Treffer 18
3.5 Anti-Twin: Benutzeroberfläche 20
3.6 Anti-Twin: Suchergebnisse 21
3.7 Anti-Twin: Fehlerhafte Ergebnisse 22
3.8 Anti-Twin: Korrektes Ergebnis 23
3.9 Anti-Twin: Versagen des Algorithmus 24
3.10 PictureRelate: Benutzeroberfläche 26
3.11 PictureRelate: Einstellung der Ähnlichkeitskriterien 27
4.1 Beispiel 1 zum Begriff der Ähnlichkeit 31
4.2 Beispiel 2 zum Begriff der Ähnlichkeit 31
4.3 Beispiel 3 zum Begriff der Ähnlichkeit 32
5.1 Verschiedene Algorithmen zum Bildvergleich: Originalbild 35
5.2 Beispielbilder für Ähnlichkeitsgrad 2 37
5.3 Beispielbilder für Ähnlichkeitsgrad 2 nach Erweiterung durch DCT 38
5.4 Beispielbilder für Ähnlichkeitsgrad 3 40
5.5 Geometrie des Template Matching 40
5.6 Beispielbilder für Ähnlichkeitsgrad 4 42
6.1 Klassendiagramm des `ImageComparers` 44
6.2 Klassendiagramm mit den wichtigsten Klassen der GUI 45
6.3 Klassendiagramm mit den `Comparer` Klassen 46
6.4 Anwendungsfalldiagramm 47
6.5 Aktivitätsdiagramm 48
6.6 Sequenzdiagramm zum Szenario „Bild mit Bild Vergleich“ 49

6.7 Sequenzdiagramm zum Szenario „Bild mit Bild Vergleich“ 50
6.8 Das Hauptfenster 52
6.9 Das Fortschrittsfenster 55
6.10 Das Ergebnisfenster 58
7.1 Ergebnisse des Vergleichs 74

Abkürzungsverzeichnis

AWT	Abstract Windowing Toolkit
DCT	*Discrete Cosine Transformation* (Diskrete Kosinustransformation)
dpi	dots per inch
GIF	Graphics Interchange Format
GUI	Graphical User Interface
JAI	Java Advanced Imaging
JFIF	JPEG File Interchange Format
JPEG	Joint Photographic Experts Group
LZW	Lempel-Ziv-Welch Algorithmus
PKZIP	Phil Katz' ZIP Programm
PNG	Portable Network Graphics
RGB	Rot Grün Blau
TIFF	Tagged Image File Format
URL	Uniform Resource Locator

1 Einleitung

Im Rahmen dieser Bachelorarbeit stehen ausgewählte Verfahren, um Bilder hinsichtlich ihres Inhaltes miteinander zu vergleichen, im Mittelpunkt.

1.1 Motivation

Die Bedeutung der digitalen Fotografie nahm im Laufe der letzten Jahre einen immer höheren Stellenwert ein. Mit der Möglichkeit, Fotos zu digitalisieren, sind viele Vorteile verbunden. Bilder können sofort betrachtet, bewertet und korrigiert sowie zentral auf der Festplatte des Computers gespeichert werden. Damit ist die Aufbewahrung einer Vielzahl von Fotos im Dateisystem des Rechners verbunden. Dies zieht die Gefahr von Redundanz nach sich, insbesondere, wenn die Fotos auf verschiedenen Rechnern verwaltet werden. So kann es vorkommen, dass sich ein und dasselbe Bild in unterschiedlichen Ausführungen auf verschiedenen Festplatten befindet. In solchen Fällen ist ein Programm nützlich, das die einander ähnlichen Bilder im Dateisystem anhand der Originalvorlage ermittelt, so dass die entsprechenden Duplikate gelöscht werden können.

Ein zweiter Anwendungsfall für einen Ähnlichkeitsvergleich von Bildern findet sich im Internet. Mit der Anzahl der digitalen Fotografen nimmt auch die Verbreitung von Fotos im Internet zu. Die Vorteile dessen sind offensichtlich: Ein Künstler kann seine Fotos im Internet veröffentlichen und spricht somit ein breites Publikum von potentiellen Käufern an. Die meisten Bilder, die sich einmal im Internet befinden, sind allerdings von anderen Internetnutzern leicht zu kopieren. Für den Fotografen ist es nicht nachzuvollziehen, wer seine Fotos für welchen Zweck verwendet. Auch in diesem Fall ist ein Programm, das im Web nach Duplikaten von Fotos sucht, eine große Hilfe, um mögliche Copyrightverletzungen aufzudecken.

Dieser Art von Programmen liegen komplexe Algorithmen zum Bildvergleich zugrunde, von deren Leistungsfähigkeit die Trefferquote und damit der Nutzen derartiger Software abhängen. Da Computer keine kontinuierlichen Bilder verarbeiten können, sondern nur digitale Zahlenfelder, müssen zu vergleichende Bilder mit Verfahren der Bildverarbeitung analysiert werden. Derartige Vorgehensweisen, die in der Lage sind, zuverlässig Fotos miteinander zu vergleichen, sind bis heute ein Bestandteil der Forschung.

1.2 Szenario

Es liegt eine Menge von Bildern, zumeist Fotos, in unterschiedlichen Formaten im Dateisystem vor. Einige der Bilder können einander ähnlich sein, in dem sie z.B. das gleiche Motiv in einer anderen Farbe oder auch einen anderen Ausschnitt zeigen. Ein vorgegebenes Originalbild ist mit der Menge aller Bilder zu vergleichen.

1.3 Aufgaben und Ziel

Zunächst sind alle möglichen Ähnlichkeitsvariationen zweier Bilder zu analysieren. Anhand dieser Ergebnisse ist ein Ähnlichkeitsgrad zu definieren. Im Anschluss ist eine Recherche durchzuführen, um bereits existierende Programme zu finden, die einen Bildvergleich implementieren. Die gefundenen Softwarelösungen werden miteinander hinsichtlich ihrer Funktionalität zueinander in Bezug gesetzt. Anschließend werden Algorithmen, die zur Lösung des Problems verwendet werden können, miteinander verglichen. Das Ziel der vorliegenden Arbeit ist die Entwicklung eines Prototyps, bei der die in der Analyse gewonnenen Erkenntnisse angewendet werden. Der Prototyp führt einen Ähnlichkeitsvergleich durch, indem ein vorgegebenes Original mit jedem anderen Foto aus dem Dateisystem verglichen und der Ähnlichkeitsgrad ausgegeben wird.

1.4 Gliederung der Arbeit

Kapitel 2 gibt einen Überblick über die Grundlagen, die zum Verständnis der vorliegenden Arbeit von Bedeutung sind. Im 3. Kapitel werden vier bereits existierende Programme, die einen Bildvergleich realisieren, analysiert. Dabei wird auf ihre Handhabung, die gelieferten Ergebnisse und ihre Funktionsweise eingegangen. Die Problemstellungen, die ein Vergleich der Ähnlichkeit von Bildern aufwirft, werden im 4. Kapitel behandelt. Dort stellt sich u.a. die Frage, wie die Ähnlichkeit von Bildern definiert ist. Die Lösungsmöglichkeiten der aufgezeigten Probleme werden im Kapitel 5 besprochen. Dazu werden Ähnlichkeitsgrade definiert sowie auf verschiedene Algorithmen und Verfahren, die für die Implementierung des Prototyps verwendet werden können, eingegangen. Hinweise zur eigentlichen Implementierung des Prototyps sind im 6. Kapitel zu finden. Das Augenmerk liegt hier auf der Struktur und dem Ablauf des Programmes. Dies wird mithilfe von verschiedenen Diagrammtypen und anhand der wichtigsten Quellcodeausschnitte dargestellt. In Kapitel 7 folgt die Zusammenfassung der Ergebnisse dieser Arbeit. Der entstandene Prototyp wird hinsichtlich der gestellten Anforderungen bewertet. Außerdem werden Anregungen zur Erweiterung der Software gegeben.

2 Grundlagen

In diesem Abschnitt werden die theoretischen Grundlagen gelegt, die sowohl zur Einleitung in das Thema, als auch zum besseren Verständnis der späteren Abschnitte dienen.

2.1 Bildverarbeitung, Bildbearbeitung, Computergrafik

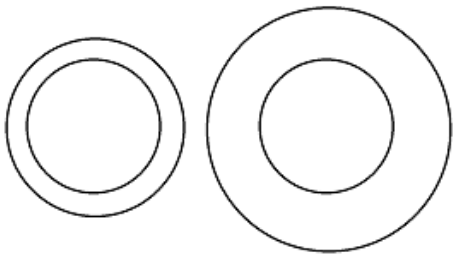
Die Bildverarbeitung wird in der Umgangssprache häufig mit der Bildbearbeitung gleichgesetzt oder verwechselt. Letzteres beschreibt aber die Manipulation von Bildern mit fertigen Programmen wie *Adobe Photoshop* oder *Corel Photo-Paint*. Im Gegensatz dazu umfasst die Bildverarbeitung die Konzeption und Erstellung solcher Software. Die Computergrafik beschäftigt sich hingegen mit der Synthese von Bildern aus geometrischen Beschreibungen. Doch auch wenn jeder dieser Themenbereiche eigene Ziele und Anwendungsfälle hat, gibt es trotzdem viele Berührungspunkte zwischen den Methoden, die dabei zum Einsatz kommen. Im Rahmen dieser Bachelorarbeit werden vor allem die Mittel der Bildverarbeitung im Vordergrund stehen.

2.2 Bildanalyse im Gegensatz zu Mustererkennung und Computer Vision

Viele Problemstellungen der Bildverarbeitung scheinen für das Auge des Menschen auf den ersten Blick einfach lösbar. Doch bei genauerer Betrachtung fällt auf, dass insbesondere Aufgaben aus dem Bereich der Bildanalyse, welche dazu dient, sinnvolle Informationen aus Bildern zu extrahieren, nicht so trivial sind, wie sie scheinen. Die Methoden der Bildanalyse kommen bei der Segmentierung von Bildregionen, Auffinden von einfachen Kurven sowie auch bei dem Vergleich von Bildern zum Einsatz. Dabei wird ausschließlich auf Basis von Pixeldaten gearbeitet, sprich „bottom-up“. Zusätzliches Wissen über das Bild steht nicht zur Verfügung. Im Gegensatz dazu stehen die Musterkennung und Computer Vision. Diese Vorgehensweisen greifen zwar häufig auf die Methoden der Bildverarbeitung zurück, gehen hinsichtlich ihrer Anforderungen jedoch weit über eine einfache Bildanalyse hinaus. Die Mustererkennung beschäftigt sich, wie der Name vermuten lässt, mit der Erkennung von Mustern in Daten. Damit ist es möglich, Texturen zu unterscheiden oder auch eine optische Zeichenerkennung durchzuführen. Im Mittelpunkt des maschinellen Sehens (Computer Vision), einem Teilgebiet der künstlichen Intelligenz, steht die Aufgabe, Sehvorgänge, wie sie in der realen Welt vorkommen, maschinell umzusetzen. Das schließt die räumliche Erfassung von Gegenständen, das Erkennen von Objekten und die Interpretation von Bewegungen mit ein.

2.3 Menschliches und maschinelles Sehen

Die Grundlage für die maschinelle Bildverarbeitung ist das menschliche Sehsystem. Im Gegensatz zu einem Computer ist der Mensch in der Lage, mithilfe seines visuellen Systems ein Bild intuitiv zu beurteilen. Die maschinelle Bildverarbeitung muss sich daher an den Eigenschaften des menschlichen Sehsystems orientieren. Mithilfe des Auges kann ein Mensch zwischen verschiedenen Intensitäten und Farbtönen unterscheiden, die Längen und Flächen verschiedener Objekte miteinander vergleichen, sowie ein Objekt innerhalb eines Bildes anhand von verschiedenen Eigenschaften erkennen. Dem menschlichen Sehsystem fehlt aber die Fähigkeit, genaue mathematische Aussagen über die Lage eines Objektes in einem Bild, den Längenunterschied zweier Strecken oder auch die konkrete Farbwertdifferenz zweier Farbtöne zu treffen. Diese Tatsache begründet sich damit, dass der Mensch mit dem Auge ein Bild „versteht“, ohne es mathematisch zu analysieren. Dadurch unterliegt das menschliche Auge auch häufig optischen Täuschungen, wie Abbildung 2.1 zeigt.



(a) Die eigentlich identischen Innenkreise wirken unterschiedlich groß.

(b) Trotz gleicher Länge scheinen die parallelen Linien unterschiedlich lang zu sein.

Abbildung 2.1: Zwei Beispiele für optische Täuschungen[1]

Ein Computer hingegen ist (bisher) noch nicht in der Lage, Bilder in der Form zu „verstehen“, wie Menschen es können. Stattdessen muss auf fest definierte Bildverarbeitungsalgorithmen zurückgegriffen werden, mit denen z.B. wichtige Bildabschnitte („Points of interest“, s. Abbildung 2.2a) oder auch Kanten und Ecken (Abbildung 2.2b) in einem Bild erkannt werden können.

[1] Nach [Jäh02], S.19

(a) Erkennung von „Points of interest“[2]

(b) Kantendetektion

Abbildung 2.2: Zwei Beispiele für maschinelle Bildverarbeitung

2.4 Grundlagen digitaler Bilder

Digitale Bilder kommen u.a. in fast allen Bereichen der Informatik zur Anwendung. Im Folgenden werden grundlegende Informationen zu digitalen Bildern erläutert.

2.4.1 Arten von digitalen Bildern

In der Praxis kommen viele verschiedene Arten von Rasterbildern vor. Diese bestehen aus regelmäßig angeordneten Elementen. Beispiele für Rastergrafiken sind Fotos (Abbildung 2.3a), Farb- und Grautonbilder (Abbildung 2.3b), gescannte Druckvorlagen, Baupläne, Fax-Dokumente, Bildschirmfotos, Ultraschallbilder, Magnetresonanz-Tomographie Aufnahmen (Abbildung 2.3c) usw.

[2]Quelle: [Sch10], S.11

(a) Foto (b) Grautonbild (c) Magnetresonanz-Tomographie[3]

Abbildung 2.3: Beispiele für verschiedene Arten von Bildern

Die meisten Arten von Bildern bestehen aus rechteckig angeordneten Bildelementen. Die verschiedenen Arten von Bildern unterscheiden sich hauptsächlich durch die in den Bildelementen abgelegten Werte. Ein digitales Bild kann als zweidimensionale Funktion von ganzzahligen Koordinaten $\mathbb{N} \times \mathbb{N}$ auf eine Menge von Bildwerten $\mathbb{P}$, also $I(u,v) \in \mathbb{P}$ und $u,v \in \mathbb{N}$ aufgefasst werden.

2.4.2 Eigenschaften von Bildern

Bildgröße und Auflösung

Die Größe eines Bildes wird durch die Breite M und die Höhe N der zugehörigen Bildmatrix I bestimmt. Dabei spezifiziert M die Anzahl der Spalten und N die Anzahl der Zeilen. Die Auflösung eines Bildes bezeichnet seine räumliche Ausdehnung in der realen Welt. Sie wird in der Anzahl der Bildelemente pro Längeneinheit angegeben, z.B. in „dots per inch“ (dpi).

[3] Quelle: [Jäh02], S.7

(a) 5x4 Bildpunkte

(b) 16x12 Bildpunkte

(c) 60x45 Bildpunkte

(d) 300x225 Bildpunkte

Abbildung 2.4: Darstellung eines Bildes mit verschiedener Anzahl von Bildpunkten[4]

Die Abbildung 2.4 zeigt, wie sich die Anzahl der Bildpunkte auf die Darstellung des Bildes auswirkt. Erst wenn ein Bild ausreichend viele Bildpunkte enthält, nimmt das menschliche Auge es als kontinuierliches Bild wahr.

Bildkoordinaten

Mithilfe eines Koordinatensystems kann angegeben werden, welche Bildposition zu welchem Bildelement gehört. Dabei verläuft die y-Achse, wie in der Programmierung üblich, von oben nach unten, so dass sich der Koordinatenursprung links oben befindet.

[4]Nach [Jäh02], S.31

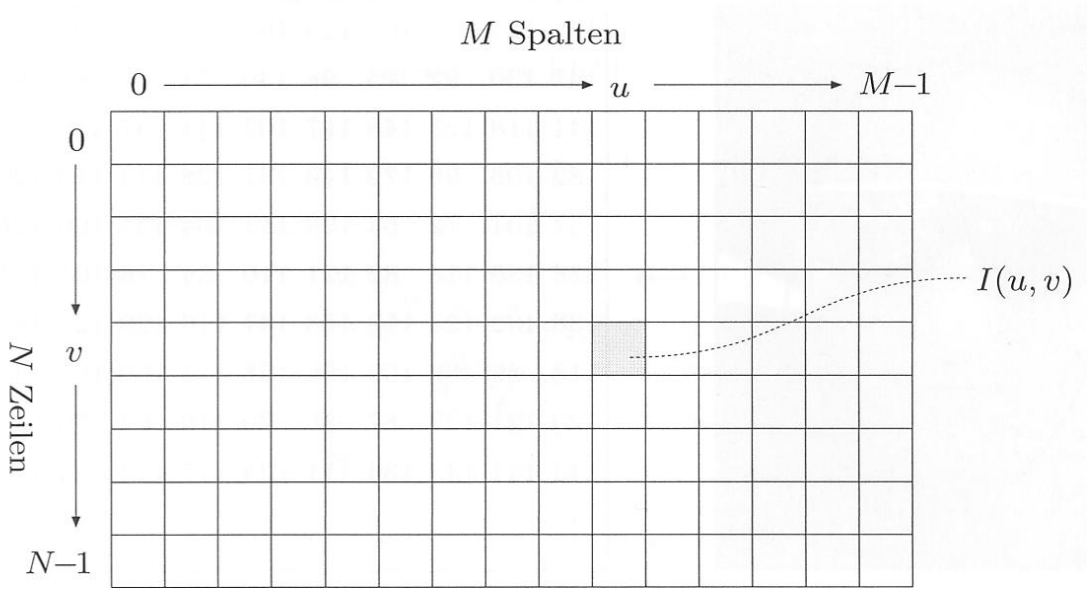


Abbildung 2.5: Bildkoordinaten. Für ein Bild der Größe $M \times N$ beträgt der maximale Spaltenindex $u_{max} = M - 1$ und der maximale Zeilenindex $v_{max} = N - 1$. [5]

Pixelwerte

Pixelwerte sind binäre Wörter der Länge k. Ein Pixel kann grundsätzlich 2^k Werte annehmen. Dabei entspricht k der Bit-Tiefe des Bildes. Der Typ des Bildes (Grauwert-, Binär- oder RGB Bild) bestimmt, wie die genaue Kodierung der einzelnen Pixelwerte in zugehörige Bitmuster erfolgt. Bei Grauwertbildern bestehen Bilddaten aus einem Kanal, der die Intensität beschreibt. Hierzu werden üblicherweise positive ganze Zahlen im Bereich $[0...2^k - 1]$ zur Darstellung verwendet. Ein typischer Wert ist $k = 8$ Bit pro Pixel, wodurch sich Intensitätswerte von 0...255 ergeben. Binärbilder sind Intensitätsbilder, die die Besonderheit aufweisen, dass sie nur zwei Pixelwerte haben - schwarz und weiß. Sie werden z.B. für Strichgrafiken und die Kodierung von Faxdokumenten verwendet. Ein Farbbild weist drei Komponenten für die Primärfarben Rot, Grün und Blau auf. Typischerweise gibt es dabei 8 Bit pro Komponente, so dass jedes Pixel aus $3 \cdot 8 = 24$ Bit besteht.

[5] Quelle: [BB06], S.12

2.4.3 Dateiformate

Die Auswahl des Dateiformates ist abhängig von verschiedenen Aspekten. Die Art des Bildes ist entscheidend, da sich für Schwarzweißbilder, Grauwertbilder und Scans andere Dateiformate anbieten als beispielsweise für Farbfotos. Zudem sind Fragen über Speicherbedarf und Kompression zu klären: Darf das Bild eine hohe Dateigröße aufweisen? Ist eine verlustbehaftete Kompression erlaubt? Auch die geforderte Kompatibilität ist zu beachten. Je nachdem, wie wichtig der Austausch von Bilddaten oder die Archivierung ist, sollten entsprechend passende Dateiformate zur Speicherung eines Bildes gewählt werden. Zuletzt ist auch der Anwendungsbereich von Bedeutung. Soll das Bild im Druck, Web, in der Computergrafik oder der Medizin zum Einsatz kommen? Die gängigsten Bildformate, ihre Eigenschaften und Anwendungsgebiete werden daher im Folgenden kurz dargestellt.

TIFF - Tagged Image File Format

Das Tagged Image File Format (TIFF) ist ein Dateiformat für professionelle Ansprüche, wie z.B. zum Austausch von Daten in der Druckvorstufe. Es deckt zahlreiche Anwendungsbereiche ab und unterstützt Grauwert-, Index[6]- und Vollfarbenbilder[7]. Für dieses Format werden unterschiedliche Kompressionsverfahren und Farbräume unterstützt. TIFF hat den Vorteil, dass das Bild in einzelnen Blöcken gespeichert wird, die separat gelesen und dargestellt werden können. So können auch große Bilder ohne hohen Speicherbedarf bearbeitet werden. Zur Darstellung im Browser sollte dieses Format nicht verwendet werden.

GIF - Graphics Interchange Format

Das Graphics Interchange Format (GIF) wird ausschließlich für Indexbilder verwendet. Es gibt die Möglichkeit für transparente Farbwerte sowie animierte GIF-Dateien, die mehrere Bilder gleicher Größe enthalten können. Als Kompressionsverfahren wird der verlustfreie „Lempel-Ziv-Welch-Algorithmus“ (LZW) verwendet. Dieses Format eignet sich vor allem für Farbgrafiken, die mit wenigen Farbwerten auskommen, wie beispielsweise Firmenlogos.

PNG - Portable Network Graphics

Das Portable Network Graphics (PNG) Format ist der Nachfolger von GIF. Es dient als universelles Bildformat insbesondere für Internetanwendungen. Das Format unterstützt Vollfarben-, Grauwert- und Indexbilder. Es existiert ein Alphakanal für transparente Werte mit maximal 16 Bit. Zur Kompression wird das „Phil Katz’ ZIP Programm“ (PKZIP) eingesetzt.

[6] Farb- und Grauwertbilder mit maximal 8 Bit Farbtiefe
[7] Bilder mit bis zu 24 Bit Farbtiefe

JFIF - JPEG File Interchange Format

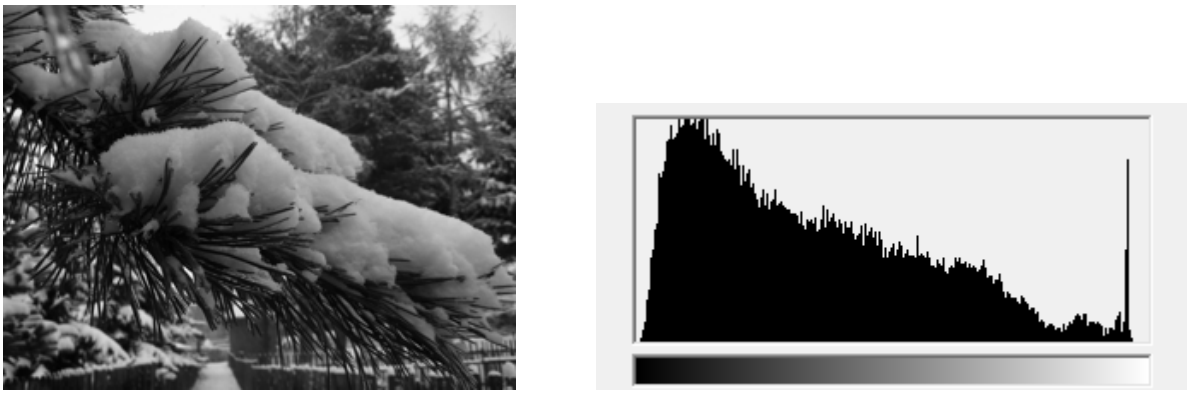
Das Joint Photographic Experts Group (JPEG) File Interchange Format (JFIF) ist das Format, das üblicherweise als JPEG Datei bezeichnet wird. Der eigentliche JPEG Standard definiert allerdings nur den JPEG Kompressor und Dekompressor, alle übrigen Elemente werden durch JFIF spezifiziert. Das JPEG Verfahren kombiniert mehrere verschiedene, sich ergänzende und zum Teil verlustbehaftete Kompressionsmethoden. Es sind Bilder mit bis zu 256 Farbkomponenten vorgesehen. Das Format wird in zahlreichen Bereichen verwendet, wie z.B. zur Darstellung im Internet und in der Fotografie.

2.5 Histogramme

Das Histogramm eines Bildes beschreibt die Häufigkeit der einzelnen Helligkeitswerte. Es wird verwendet, um Bildfehler aufzudecken und die Bildintensität auszuwerten.

2.5.1 Definition eines Histogramms

Die Definition eines Histogramms lässt sich am besten für ein Grauwertbild I verdeutlichen (s. Abbildung 2.6).



(a) Grauwertbild (b) Zugehöriges Histogramm

Abbildung 2.6: Ein Grauwertbild und das zugehörige Histogramm

Das Grauwertbild I hat mögliche Intensitätswerte im Bereich $I(u,v) \in [0, K-1]$. Dabei gilt für ein typisches 8-Bit-Grauwertbild $K = 2^8 = 256$. Das zugehörige Histogramm H enthält damit genau K Einträge. Jeder Eintrag $H(i)$ im Histogramm ist wie folgt definiert:

Definition 2.1 (Histogramm[8])**.**

$h(i) =$ die Anzahl der Pixel von I mit dem Intensitätswert i für alle $0 \leq i < K$.

Formaler ausgedrückt:

$$h(i) = \text{card}\{(u,v)|I(u,v) = i\}.$$

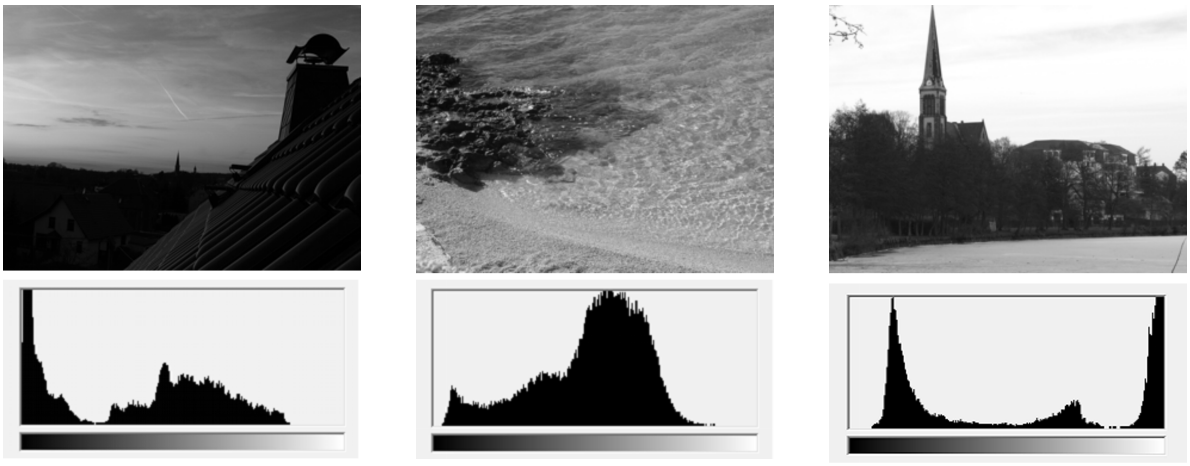
Dabei bezeichnet card die Kardinalität, sprich die Anzahl der Elemente einer Menge.

Nach dieser Definition bestimmt $h(0)$ die Anzahl der Pixel mit dem Wert 0, $h(1)$ die Anzahl der Pixel mit dem Wert 1 usw. Schließlich gibt $h(255)$ die Anzahl der Pixel mit dem maximalen Intensitätswert – also die Anzahl der weißen Pixel – an. Alle räumlichen Informationen über das zugehörige Bild, wie z.B. Angaben darüber, aus welchem Bildbereich die einzelnen Einträge stammen, gehen in einem Histogramm verloren. Daher ist es nicht möglich, anhand eines Histogramms das Originalbild zu konstruieren.

2.5.2 Interpretation eines Histogramms

Ein Histogramm zeigt wesentliche Eigenschaften eines Bildes, wie Kontrast, Dynamik, Belichtung und Bildfehler. Das Augenmerk bei der Interpretation eines Histogramms sollte auf der Gleichmäßigkeit der Häufigkeitsverteilung liegen. Belichtungsfehler liegen vor, wenn in größeren Intensitätsbereichen an einem Ende der Intensitätsskala eine Häufung von Pixelwerten auftritt, während auf der gegenüberliegenden Seite große Bereiche ungenutzt bleiben. Ein Bild ist unterbelichtet, wenn sich Pixelwerte im minimalen Intensitätsbereich häufen (Abbildung 2.7a) und überbelichtet, wenn das Histogramm eine Ansammlung von Pixelwerten im maximalen Intensitätsbereich aufweist (Abbildung 2.7c).

[8] Nach [BB06], S.40



(a) Unterbelichtung (b) Korrekte Belichtung (c) Überbelichtung

Abbildung 2.7: Erkennen von Belichtungsfehlern im Histogramm [9]

Der Kontrast lässt sich aus dem Histogramm ablesen, indem überprüft wird, inwiefern ein Bild den gesamten Bereich von Intensitätswerten ausnutzt. Bei Bildern mit wenig Kontrast bleiben viele Intensitätswerte ungenutzt, während ein Bild mit maximalem Kontrast alle Werte von schwarz bis weiß verwendet. Auch verschiedene Arten von Bildfehlern können mit einem Histogramm aufgedeckt werden, wie z.B. ein zu starker JPEG Kompressionsgrad.

[9] Nach: [BB06], S.42

3 Analyse bereits existierender Bildvergleich-Programme

Es existieren bereits verschiedene Programme, die einen Bildvergleich durchführen. Dabei gibt es verschiedene Softwarelösungen, welche sich durch den Anwendungsbereich, die implementierten Algorithmen und ihre Funktionsweise unterscheiden. In diesem Kapitel werden vier Programme hinsichtlich ihrer Funktionalität, den verwendeten Algorithmen und der gelieferten Ergebnisse analysiert. Dabei werden jeweils zwei Programme aus dem Desktop- und Webbereich gewählt.

3.1 TinEye

TinEye ist eine von Idée Incorporated entwickelte Websuche, die das Internet nach ähnlichen Bildern durchsucht. Ein beliebiges Foto, das per URL oder auch per Upload von der Festplatte an die Suchmaschine übergeben wird, dient dabei als Vorlage. Zusätzlich zu der Website des Dienstes, die über die URL `http://www.tineye.com/` aufrufbar ist, existieren auch Plug-Ins für alle gängigen Browser. Neben der Websuchmaschine gibt es von Idée Incorporated auch die Desktoplösung PixMatch zum Auffinden ähnlicher Bilder im Dateisystem.

3.1.1 Handhabung

Das Foto, nach dem das Internet auf Duplikate durchsucht werden soll, wird entweder direkt von der Festplatte des Rechners hochgeladen oder über die URL an die Suchmaschine übermittelt. Nach Installation des Browser Plug-Ins ist es möglich, ein beliebiges Bild mit Rechtsklick über den Menüpunkt „Search Image on TinEye“ direkt suchen zu lassen. In diesem Fall öffnet sich in einem neuen Tab die Website mit dem TinEye Dienst, wo die Suchergebnisse angezeigt werden.

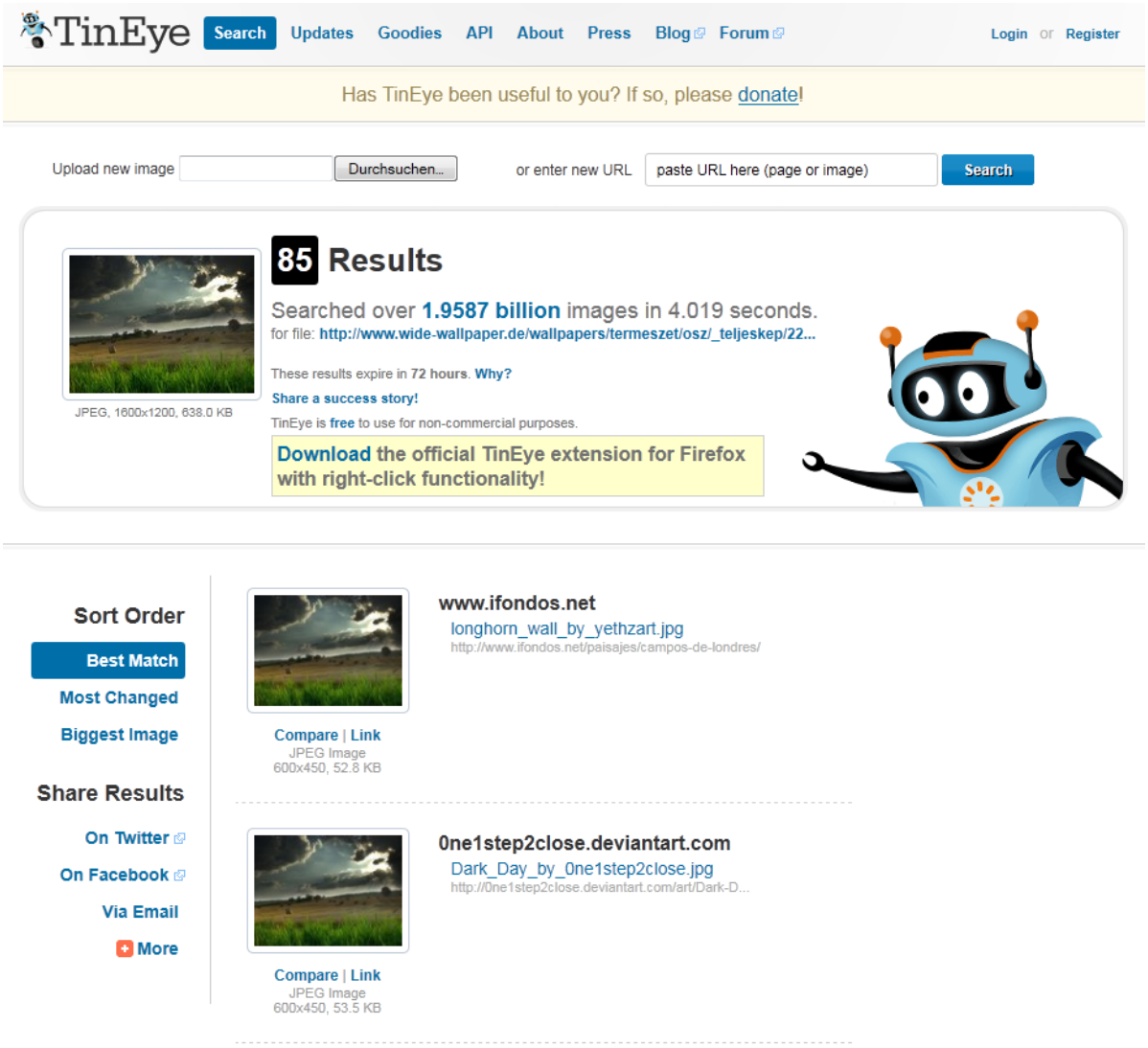
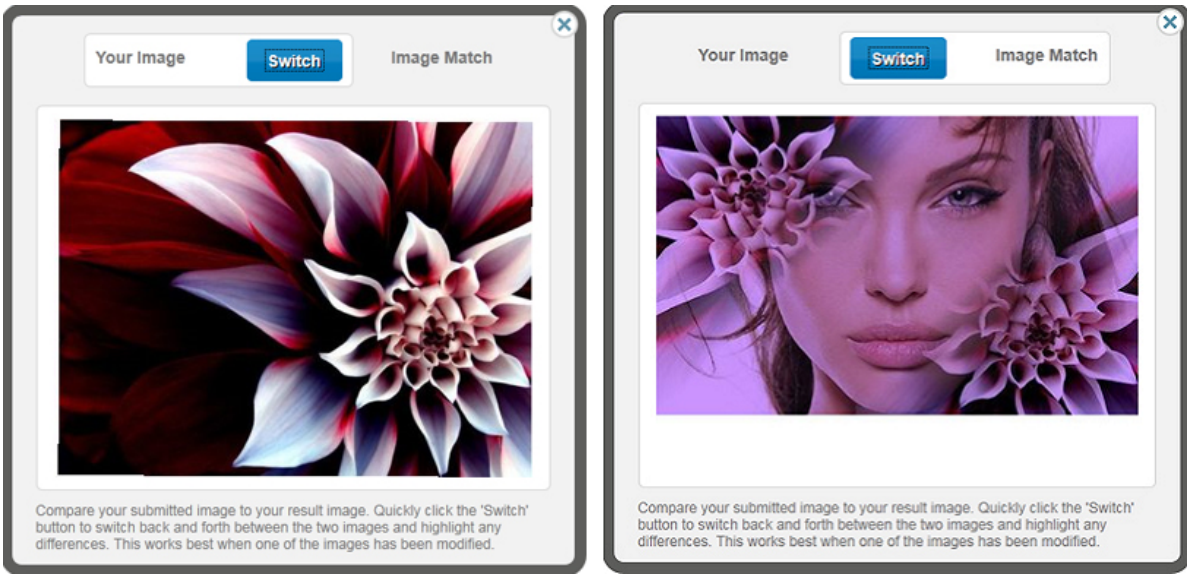


Abbildung 3.1: Die Suchergebnisse einer TinEye Suche

Die Suchergebnisse werden der Reihe nach aufgelistet. Angezeigt wird zusätzlich zu einer Vorschau auch die Bildgröße und die Adresse, wo das Duplikat gefunden wurde. Es ist möglich, ein Ergebnis über den Link „Compare“ direkt mit dem Originalbild zu vergleichen. Dabei wird automatisch angezeigt, welche Veränderungen bezüglich des ursprünglichen Fotos vorgenommen wurden, wie z.B. Farbänderungen, Ausschnitte oder Hinzufügen von Schriften und Logos.

3.1.2 Analyse der Suchergebnisse

In ungefähr 0,5 Sekunden durchsucht die Suchmaschine ca. 2 Milliarden indizierte Bilder. Dabei deckt TinEye nicht nur Duplikate auf, sondern auch einander ähnliche Bilder.



(a) Originalbild (b) Ermittelte Kopie

Abbildung 3.2: Vergleich von Originalbild mit dem gefundenen Duplikat

Abbildung 3.2 zeigt, dass TinEye auch stark verfremdete Bilder bei der Suche berücksichtigt. Das Originalbild (a) wurde ausgeschnitten, farblich verändert, gedreht und teilweise transparent in das andere Bild (b) eingefügt. Dennoch wird auch dieses Bild als Treffer erkannt.
Es ist auch möglich, der Suchmaschine ein bereits verändertes Bild zu übergeben und das Originalbild suchen zu lassen. Dabei sind allerdings einige Einschränkungen vorzunehmen. Wenn das übergegebene Foto um mehr als ein paar Grad gedreht ist, findet TinEye das Originalbild nicht mehr. Ebenso verhält es sich mit gespiegelten Bildern: Treten durch eine Spiegelung keine erheblichen Änderungen hinsichtlich des Originalbildes auf, liefert die Suche weiterhin die entsprechenden Ergebnisse. Das funktioniert z.B., wenn es sich um ein Gesichtsporträt handelt. Wird hingegen eine Landschaft gespiegelt, ist es nicht mehr möglich, das Originalbild zu finden. Auch mittels eines Bildausschnittes kann versucht werden, das zugehörige Ursprungsbild zu finden. Die dabei auftretenden Ergebnisse sind abhängig von der Größe des gewählten Ausschnittes: Je markanter der Ausschnitt gewählt wird, desto höher ist die Wahrscheinlichkeit einer guten Trefferquote.

3.1.3 Funktionsweise

Bei einer Suche mit TinEye werden überwiegend Duplikate eines Bildes gefunden. Dazu gehören auch Duplikate, die durch Bildmanipulation modifiziert wurden. Ziel von TinEye ist es hingegen nicht, Bilder zu finden, die sich hinsichtlich ihres Motivs oder ihrer Farbe ähneln.

Wird als Ausgangsbild z.B. ein Bild von einem Sonnenuntergang vorgegeben, wird nur nach diesem Bild gesucht, nicht jedoch nach anderen Fotos mit dem Motiv „Sonnenuntergang“.
Bei einer Suchanfrage wird von dem übergebenen Bild ein einzigartiger, digitaler Fingerabdruck erstellt. Dieser wird mit allen Fingerabdrücken der von TinEye bereits indizierten Bilder hinsichtlich ihrer Übereinstimmungen verglichen. Auch teilweise übereinstimmende Signaturen werden als Treffer gewertet, damit auch Ausschnitte, Bilder verschiedener Farbräume oder Skalierungen gefunden werden können. Der Algorithmus zum Erstellen des Fingerprints wurde von Idée Inc. entworfen.

3.1.4 Fazit

Die TinEye Suche ist nützlich, um nach Duplikaten im Web zu suchen. Dabei ist sowohl die Website als auch das Plug-In einfach zu handhaben und intuitiv zu verstehen. Durch die Erweiterung für den Browser ist ein schneller Zugriff auf den Dienst möglich. Allerdings kann TinEye längst noch nicht in allen Bildern des Internets suchen, da auch nach Jahren noch nicht alle Bilder indiziert wurden. Dennoch wächst die Anzahl der Fotos, die in die TinEye Datenbank übernommen werden, stetig. Somit wird die Suche auch für professionelle Anwender, die z.B. Copyrightverletzungen aufdecken möchten, immer attraktiver.

3.2 GazoPa

GazoPa wurde von Hitachi entwickelt und ist eine Websuche, um ähnliche Bilder im Internet zu finden. Dabei konzentriert sich GazoPa nicht wie TinEye auf gleiche oder abgeänderte Kopien eines Bildes, sondern auf verschiedene Bilder mit ähnlichen Motiven. Auch diese Suchmaschine ist als Browser Plug-In sowie zusätzlich als iPhone Anwendung verfügbar. Der Webdienst kann über die URL `http://www.gazopa.com/` verwendet werden[10].

3.2.1 Handhabung

Der Suchmaschine kann eine Vorgabe auf verschiedene Weise übergeben werden. Es ist möglich, ein Bild direkt von der Festplatte hochzuladen oder über eine URL anzugeben. Außerdem bietet GazoPa dem Anwender die Möglichkeit, selbst etwas zu zeichnen, wonach anschließend gesucht werden soll. Zuletzt ist es möglich, mithilfe eines Suchbegriffs zu suchen.

[10] Anmerkung: Der Dienst wurde im Mai 2011 in Anspruch genommen und getestet. Bei einem erneuten Aufruf im September 2011 wurde festgestellt, dass der Service seinen Dienst eingestellt hat.

3.2.2 Analyse der Suchergebnisse

GazoPa liefert zuverlässig Bilder, die dem Inhalt des per URL oder Upload vorgegebenen Bildes hinsichtlich Aufbau, Form und Farben ähneln.

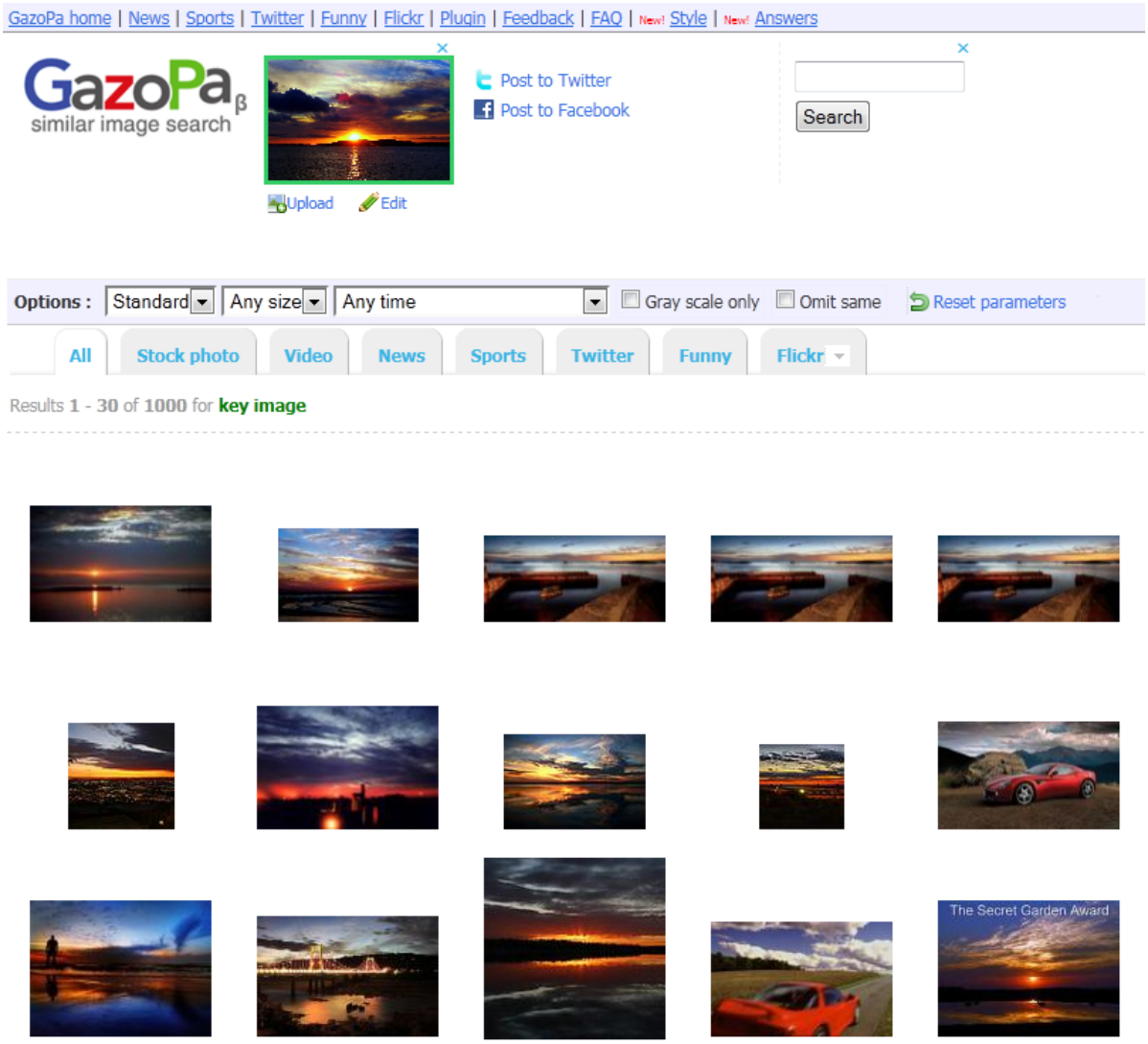


Abbildung 3.3: Anzeige ähnlicher Motive durch GazoPa

Die Erfolgsrate hängt allerdings vom Ausgangsbild ab. Übergibt man der Suchmaschine ein gedrehtes Bild oder einen Ausschnitt, liefert die Suche „false positives“, wie die Abbildung 3.4 veranschaulicht.

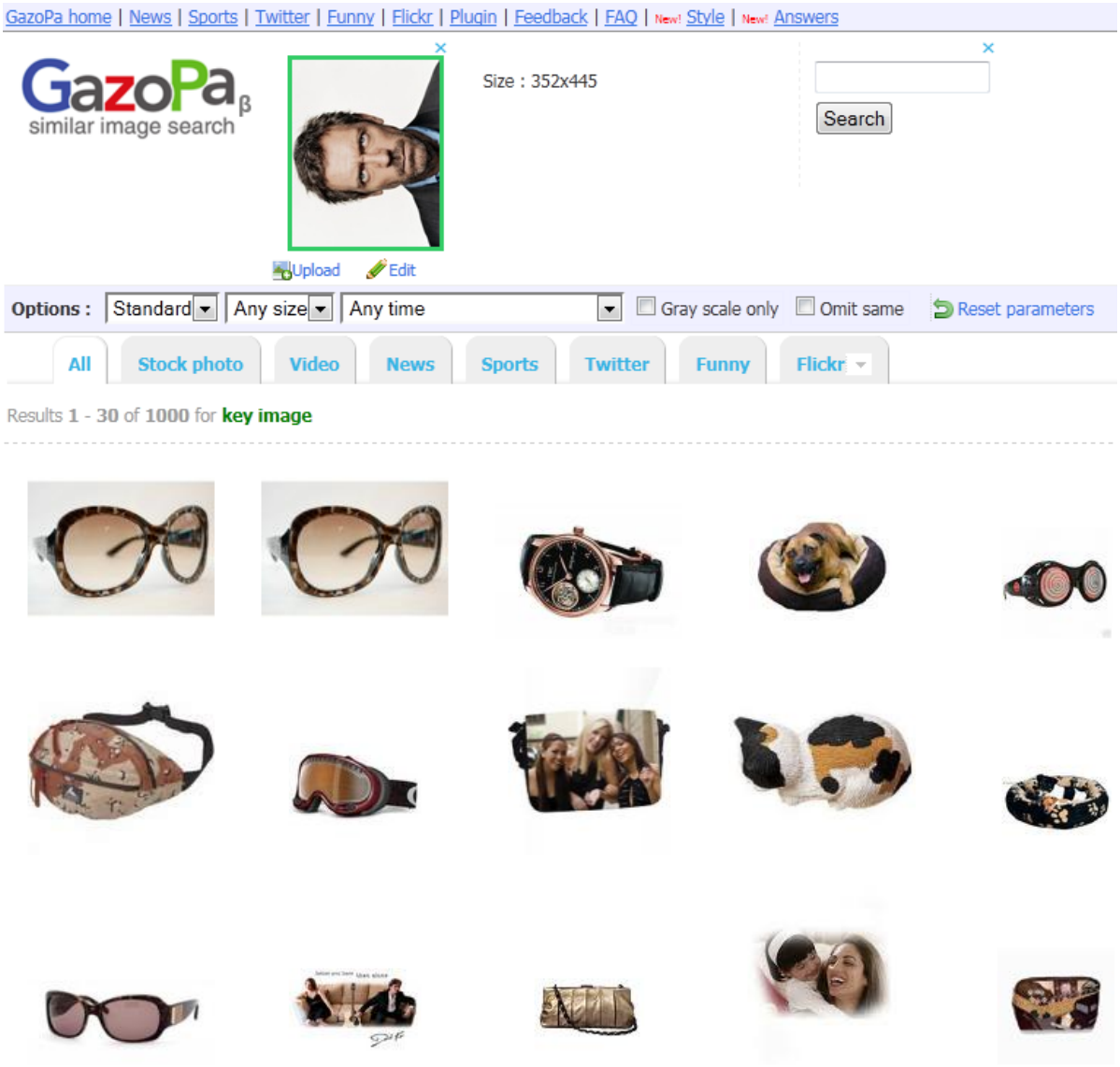


Abbildung 3.4: False Positives

Wenn der Anwender ein gezeichnetes Bild oder ein Schlagwort übergibt, ist es selten, dass wirklich brauchbare Ergebnisse geliefert werden.

3.2.3 Funktionsweise

Ähnlich wie bei TinEye werden Bilder aus dem Internet indiziert, um mit dem übergebenen Bild hinsichtlich ihres Inhalts verglichen zu werden. Dabei wird hier ein Algorithmus verwendet, der insbesondere die Form und die Farbe des Fotoinhaltes analysiert, um Bilder zu finden, die sich bezüglich ihrer Motive und Farbverläufe ähneln.

3.2.4 Fazit

GazoPa findet zwar auch einige Duplikate von Bildern, ist aber im Zusammenhang mit dem Aufdecken von Copyrightverletzungen eher unbrauchbar. Die Ähnlichkeit steht vor der Gleichheit, weswegen das Auffinden derselben Bilder nur eine Nebenbedeutung hat. Dennoch gibt es auch für diese Suchmaschine nützliche Anwendungsfälle, die dem Internetnutzer helfen. Nicht immer ist es einfach, den Inhalt eines Bildes in kurzen, präzisen Worten auszudrücken, um mit herkömmlichen Suchmaschinen weitere derartige Fotos zu finden. In diesem Fall kann eine Suche mit GazoPa Abhilfe schaffen. Die Ergebnisse bei gezeichneten Vorlagen und Schlagwörtern sind allerdings in den meisten Fällen nicht brauchbar.

3.3 Anti-Twin

Anti-Twin ist eine von Jörg Rosenthal entwickelte Desktoplösung, die doppelte oder einander ähnliche Dateien auf der Festplatte des Anwenders findet. Für den privaten Gebrauch kann die Freeware von der Website `http://www.aidex.de/software/antitwin/` heruntergeladen werden. Das Programm bietet die Möglichkeit, Dateien sowohl hinsichtlich ihres Dateinamens als auch hinsichtlich ihres Inhalts zu vergleichen. Neben einem Bildvergleich kann die Software auch Dokumente und Musikdateien auf ihre Ähnlichkeit überprüfen. Diese Eigenschaften werden an dieser Stelle aber nicht weiter untersucht.

3.3.1 Handhabung

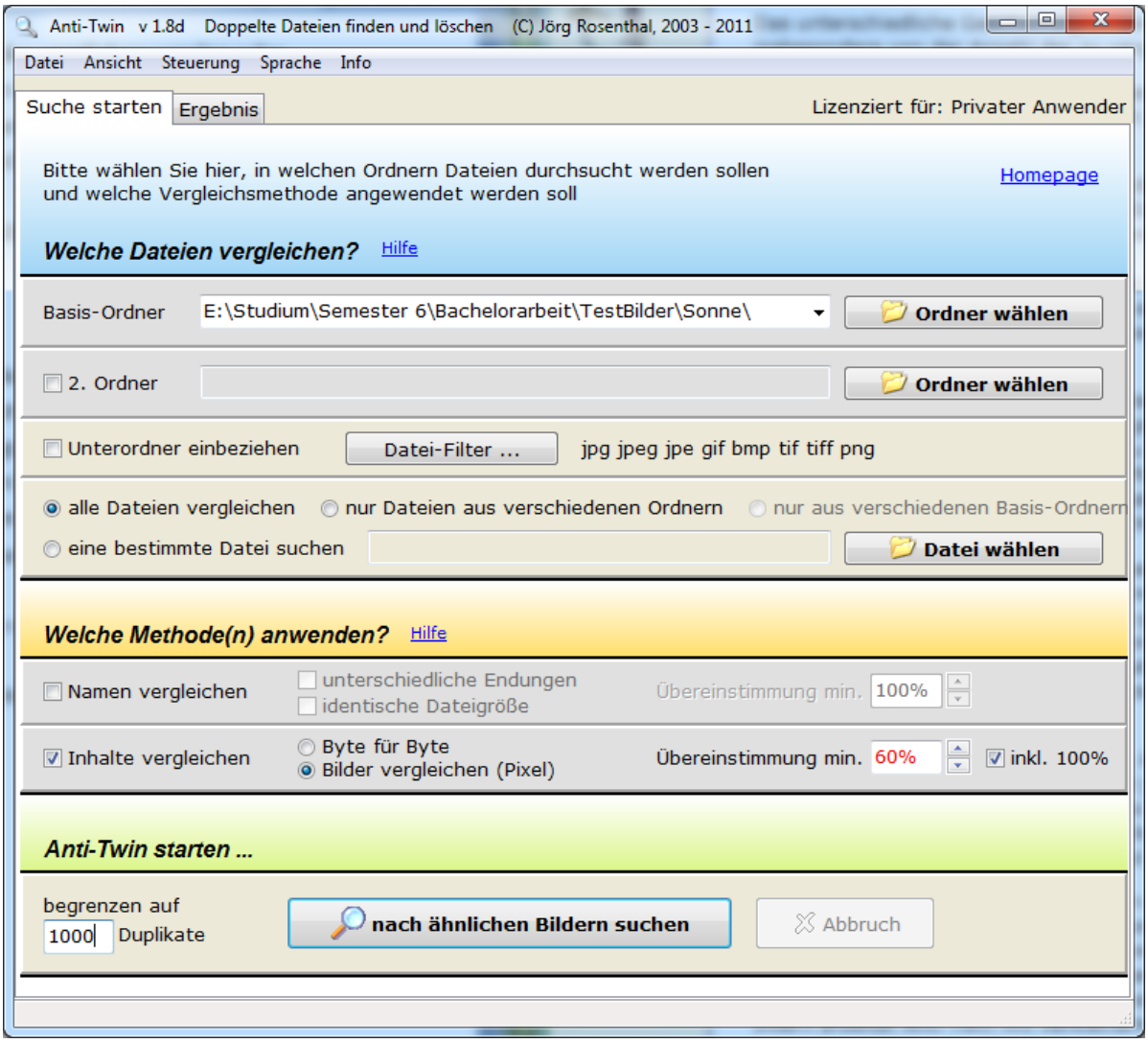


Abbildung 3.5: Oberfläche des Tools Anti-Twin

Die Benutzeroberfläche ist unterteilt in die zwei Tabs „Suche starten“ und „Ergebnis“. In ersterem wird der Ordner, der auf ähnliche Bilder untersucht werden soll, angegeben. Es gibt dann zwei Möglichkeiten zum Vergleich der Bilder: Entweder werden in dem gewünschten Ordner alle Bilder miteinander verglichen, oder es wird zusätzlich ein Bild angegeben, nach dem gesucht wird. Um eine Ähnlichkeitssuche nach Bildern zu starten, muss der Punkt „Inhalte vergleichen“ und anschließend „Bilder vergleichen (Pixel)“ ausgewählt werden. Bei einer Übereinstimmungsquote von 100% sollen nur identische Bilder als Treffer gewertet werden. Diese Prozentzahl kann auf bis zu 60% herabgesetzt werden, so dass auch Bilder mit Abweichungen noch als ähnlich erkannt werden.

3.3.2 Analyse der Suchergebnisse

Die Ergebnisse können unter dem Tab „Ergebnis“ angezeigt werden. Dort werden die Treffer aufgelistet.

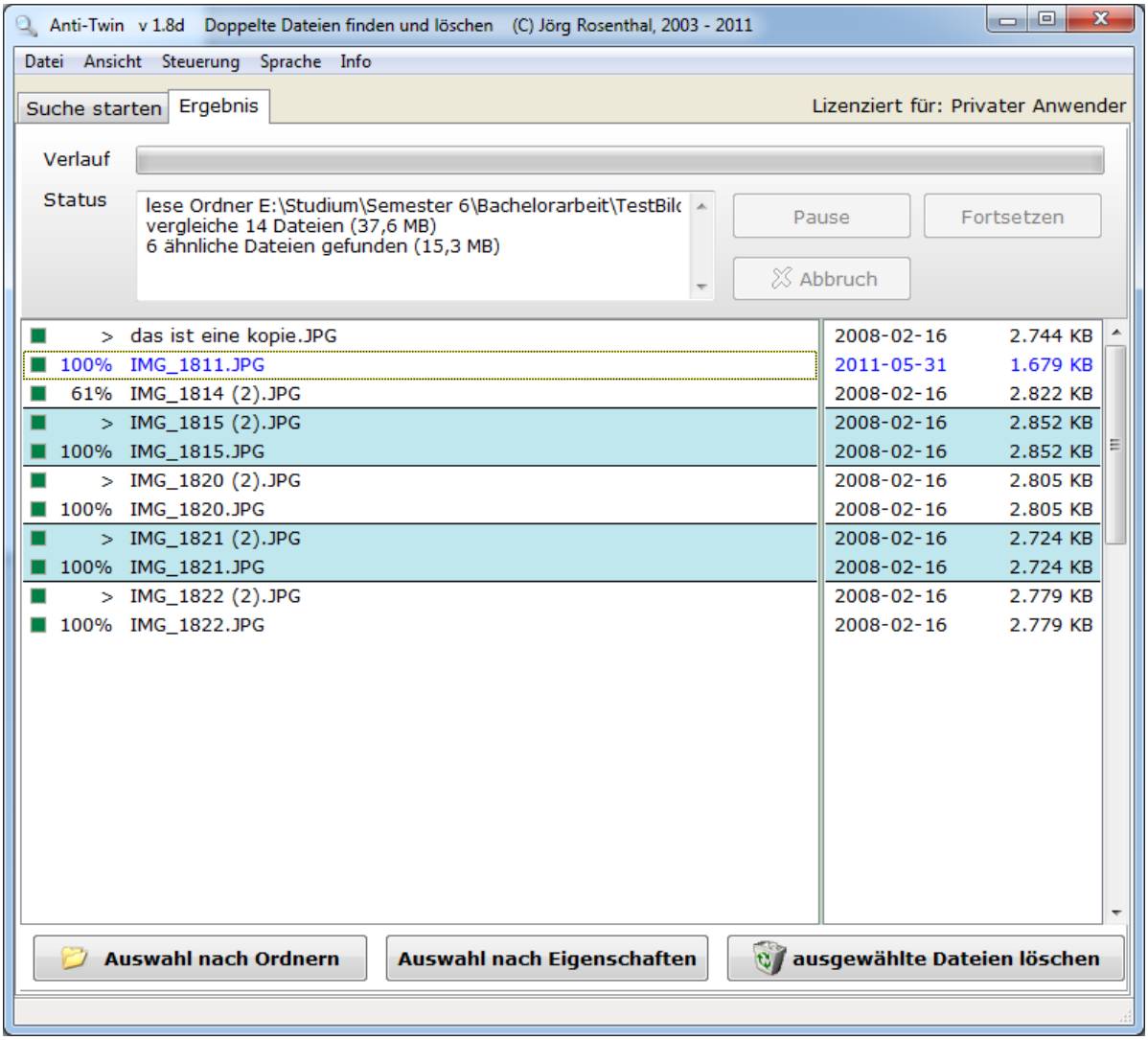


Abbildung 3.6: Anzeige der ermittelten Ergebnisse

Die Software findet überwiegend unveränderte Duplikate, auch dann, wenn die Genauigkeit auf 60% herabgesetzt wird. Zudem fallen fehlerhafte Angaben der Übereinstimmungsangaben auf. In Abbildung 3.7 wird deutlich, dass sich Bild (b) von Bild (a) durch die Umrandung der Wolke unterscheidet. Dennoch beurteilt Anti-Twin diese beiden Bilder als identisch und gibt eine Übereinstimmung von 100% an.

(a) Originalbild (b) Veränderte Kopie

Abbildung 3.7: Fehlerhafte Wertung als 100% Übereinstimmung

Trotz der Tatsache, dass sich im Testordner mehrere einander sehr ähnliche Bilder befinden, erkennt das Programm nur in einem Fall die Ähnlichkeit zwischen zwei verschiedenen Bildern, wie in Abbildung 3.8 deutlich wird.

(a) Originalbild (b) Ähnliches Bild

Abbildung 3.8: Zwei ähnliche Bilder mit 61% Übereinstimmung

Die beiden Fotos (a) und (b) sind nicht identisch, aber doch ähnlich. Die Software gibt hierfür eine Übereinstimmung von 61% an. Alle anderen ähnlichen Bilder werden nicht als solche gewertet. Auch identische Bilder, an denen größere Veränderungen vorgenommen wurden, werden nicht als ähnlich gewertet, wie die Abbildung 3.9 zeigt.

(a) Originalbild (b) Veränderte Kopie des Originalbildes

Abbildung 3.9: Bei größeren Veränderungen eines Bildes versagt der Algorithmus

Das Bild (b) ist eine Kopie des Originalbildes (a), bei dem lediglich die Helligkeits- und Farbwerte verändert wurden. Dennoch betrachtet Anti-Twin diese beiden Bilder nicht als ähnlich. Weiterhin fällt auf, dass Anti-Twin selbst einige zu 100% identische Bilder, die lediglich durch Anlegen einer Kopie entstanden, nicht als ähnlich behandelt.

3.3.3 Funktionsweise

Das Programm öffnet jedes Bild, das bei dem Vergleich berücksichtigt werden soll, und komprimiert es zu einem schwarz-weißen Miniaturbild. Anschließend werden alle Pixel miteinander verglichen, wobei aber nur ein unscharfer Vergleich durchgeführt wird. Das bedeutet, dass der Algorithmus zum Vergleich gewollt ungenau ist, indem er z.B. die Farben und die Größe des Bildes nicht berücksichtigt. So soll sichergestellt werden, dass auch Bilder mit geringen Abweichungen durch Kompression als ähnlich erkannt werden.

3.3.4 Fazit

Der Algorithmus des Programms ist ein gutes Beispiel dafür, dass ein reiner pixelbasierter Bildervergleich nicht geeignet ist, um Bilder hinsichtlich ihrer Ähnlichkeit zu vergleichen. Zwar wird der Vergleich unscharf durchgeführt, dennoch sind die Ergebnisse nicht zufrieden stellend. Die Tatsache, dass die Bilder in Abbildung 3.7 als 100% identisch erkannt werden, ist damit zu begründen, dass ausschließlich die Miniaturansichten der Bilder miteinander verglichen werden, so dass die Veränderung im zweiten Bild nicht mehr sichtbar ist. Dennoch bleibt ohne genauere Kenntnisse über den verwendeten Algorithmus weiterhin unklar, warum auch einige der identischen Bilder nicht als solche erkannt wurden. Durch diese Eigenschaft scheidet das Programm auch als zuverlässiges Tool zum Aufräumen der Festplatte aus, da selbst nach der Anwendung nicht sicher gestellt ist, dass wirklich alle doppelten Fotos entfernt wurden.

3.4 PictureRelate

PictureRelate ist eine Software von Dr. Axel Walthelm, die für den Desktopbereich entwickelt wurde. Mit PictureRelate können eigene Bilder nach Ähnlichkeit sortiert werden. Dabei ist es egal, in welchem Format die Bilder vorliegen, ob sie sich in ihrer Größe unterscheiden oder sie durch Nachbearbeitung verändert wurden. Alle gängigen Bildformate wie JPG, PNG und TIFF werden unterstützt. Weitere Bildformate können durch spezielle Import-Filter eingelesen werden. Die Software eignet sich besonders für die Anwendung mit einer großen Anzahl von Bildern, da sie über 50000 Bilder verarbeiten kann.
Unter `http://www.walthelm.net/picture-relate/de/download.php` steht das Programm als Download zur Verfügung.

3.4.1 Handhabung

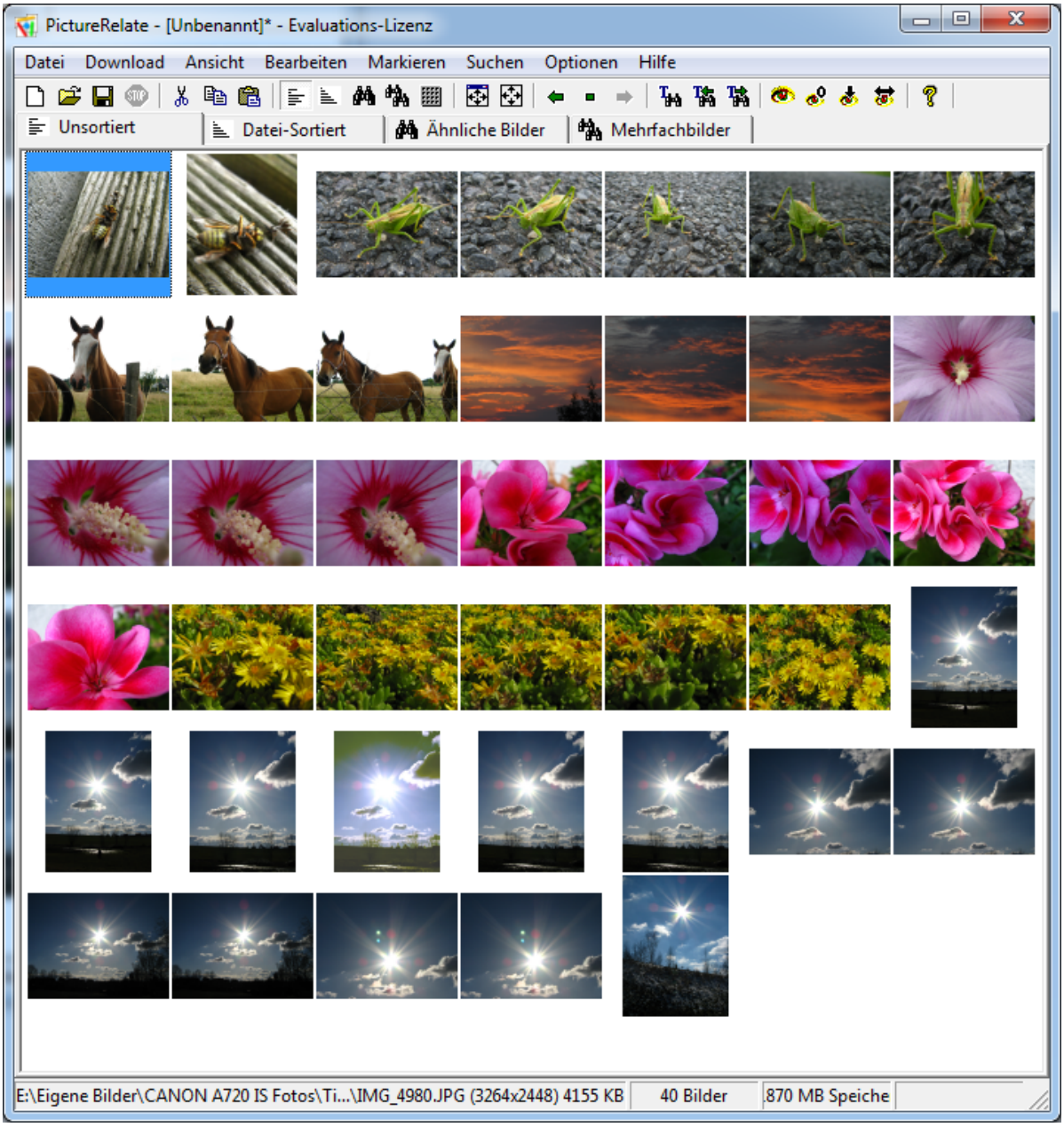


Abbildung 3.10: Die Benutzerschnittstelle von PictureRelate

Die Menge von Bildern, welche auf Ähnlichkeit und Duplikate überprüft werden soll, wird über den Menü-Befehl „Dateien importieren“ in das Programm eingebunden. Es folgt eine Miniaturansicht aller vorhandenen Bilder. Mit Doppelklick auf ein Bild wird dieses als Vorlage für den Ähnlichkeitsvergleich verwendet. Nun wird die Anzeige neu aufgebaut und die Fotos werden nach Ähnlichkeit sortiert.

Der Anwender hat die Möglichkeit, eigene Ähnlichkeitskriterien manuell festzulegen, um so bessere Ergebnisse zu erzielen.

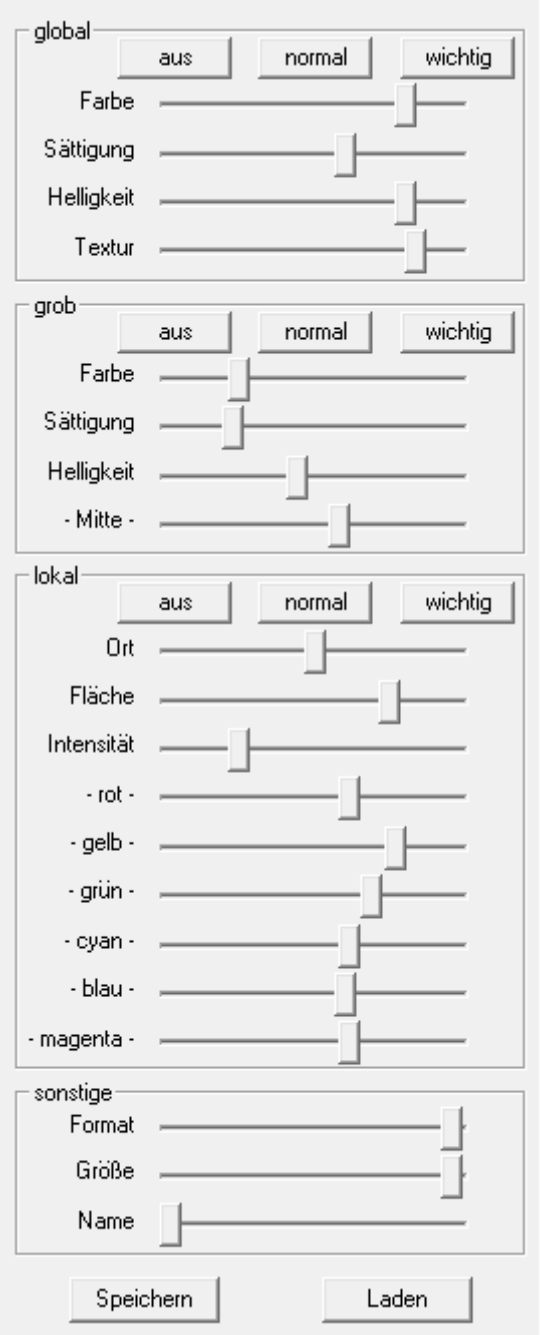


Abbildung 3.11: Einstellung der Ähnlichkeitskriterien

Unter dem Tab „Mehrfachbilder“ ist es möglich, sich Duplikate anzeigen zu lassen.

3.4.2 Analyse der Suchergebnisse

Wie bereits Abbildung 3.10 zeigt, sortiert das Programm zuverlässig nach ähnlichen Motiven. Dabei hat der Anwender selbst die Möglichkeit, die Ergebnisse über die Ähnlichkeitskriterien an seine Bedürfnisse anzupassen. Bei der Suche nach Duplikaten fällt auf, dass PictureRelate auch doppelte Bilder findet, die Anti-Twin nicht als solche erkannte. Auch die Suche nach gedrehten Bildern ist möglich. Dazu kann in den Einstellungen festgelegt werden, ob rotierte Bilder berücksichtigt werden sollen. Dies funktioniert allerdings nur für 90° Drehungen. Je nach Einstellungsmodus erkennt das Programm auch verfremdete und gespiegelte Bilder als ähnlich.

3.4.3 Funktionsweise

Die Software funktioniert auf Basis von neuronalen Netzen. Der Algorithmus verwendet die Menge von Merkmalen eines Bildes, die in der Neurologie als Pop-Out-Merkmale bezeichnet werden. So können interessante, visuelle Elemente schnell wahrgenommen werden. Daraus resultiert die hohe Geschwindigkeit des Bildvergleichs.

3.4.4 Fazit

PictureRelate führt Bildvergleiche nach Ähnlichkeit zuverlässig durch und ist auch bei einer Vielzahl an Bildern noch sehr schnell. Der Anwender braucht einige Zeit, um sich in das Programm einzuarbeiten. Erst wenn der Benutzer gelernt hat, wie die Kriterien optimal eingestellt werden, können effiziente Ähnlichkeitssuchen durchgeführt werden. Da auch Duplikate zuverlässig aufgedeckt werden, ist dieses Programm zum Aufräumen der Festplatte geeignet.

3.5 Zusammenfassung

In diesem Kapitel wurden vier Programme, die einen Bildvergleich durchführen, hinsichtlich ihrer Zuverlässigkeit der Ergebnisse analysiert. Dabei wurden bewusst je zwei Programme aus dem Web- und Desktopbereich gewählt. Ebenso unterscheiden sich die einzelnen Programme bezüglich der verwendeten Algorithmen und ihrer Anwendungsbereiche.

Während TinEye dazu geeignet ist, Copyrightverletzungen im Internet aufzudecken, da es für die Suche nach Duplikaten konzipiert wurde, ist GazoPa darauf ausgerichtet, weitere Bilder mit ähnlichen Motiven und Farben ausfindig zu machen. Dennoch ist die Rate für falsche Treffer bei GazoPa noch deutlich höher als bei TinEye. Letztere Suchmaschine liefert zudem

deutlich schneller Ergebnisse, weshalb TinEye im Webbereich mehr überzeugt.

Im Bereich der Desktopanwendungen fällt auf, dass Anti-Twin zwar intuitiv zu bedienen ist, aber große Ungenauigkeiten im Ergebnis aufweist, weil zum Teil auch identische Bilder nicht erkannt werden. Bezüglich der Ergebnisse kann PictureRelate daher eher überzeugen, auch wenn dieses Programm durch seine zahlreichen Funktionen nicht so einfach zu bedienen ist, wie Anti-Twin. PictureRelate überzeugt zudem dadurch, dass der Ähnlichkeitsvergleich auch bei einer Vielzahl von Bildern sehr schnell stattfindet.

Insgesamt ist der Algorithmus, der TinEye zugrunde liegt, am effizientesten. Unter allen getesteten Programmen fand TinEye am ehesten auch Fotos, die stark verfremdet, beschnitten, gedreht und gespiegelt waren und lieferte in keinem Testfall falsche Treffer.

4 Problem- und Anforderungsanalyse

In diesem Kapitel werden Problemstellungen, die im Zusammenhang mit einem Bildvergleich auftreten, dargelegt. Des Weiteren gilt es, Anforderungen an den zu entwickelnden Prototyp zu formulieren.

4.1 Problemstellung: Wann sind Bilder ähnlich?

Um einen Ähnlichkeitsvergleich durchführen zu können, ist zunächst folgende Problemstellung zu betrachten: Wann sind zwei Fotos ähnlich und wie kann man deren Ähnlichkeit messen? Eine Möglichkeit ist, anzunehmen, dass Bilder umso ähnlicher sind, je weniger Unterschiede sie bezüglich ihrer verschiedenen Eigenschaften, z.B. hinsichtlich ihrer Farbe, aufweisen. In Folge dessen sind zwei Bilder, die keine Unterschiede aufweisen, identisch. Dies kann mathematisch festgestellt werden, indem getestet wird, ob die Differenz der Bildwerte zwischen zwei Bildern null ergibt. In der Praxis liefert dieser Test allerdings bereits bei zwei beinahe identischen Bildern, die sich lediglich in ihrem Kompressionsgrad unterscheiden, starke Abweichungen. So werden zwei Bilder, deren Unterschiede für das menschliche Auge kaum sichtbar sind, durch diesen Algorithmus trotzdem als ungleich gewertet. Diese einfache Methode wird daher nicht ausreichen, um den Ähnlichkeitsgrad zweier Bilder zu bestimmen.

Zudem stellt sich weiterhin die Frage, welche für den Ähnlichkeitsvergleich wichtigen Eigenschaften ein Foto hat. Wird lediglich die Farbe betrachtet, können die Bilder (a) und (b) in Abbildung 4.1 als ähnlich aufgefasst werden.

(a) Zwiebel (b) Korb

Abbildung 4.1: Zwei Bilder mit ähnlichen Farben, aber verschiedenen Motiven

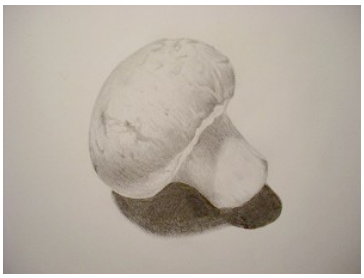
Konzentriert man sich jedoch auf das Motiv, sind diese beiden Bilder völlig ungleich. Den Gegensatz dazu verdeutlicht Abbildung 4.2. Die Bilder (a) und (b) sind auf ihr Motiv bezogen ähnlich, während sie sich in ihrer Farbe unterscheiden.

(a) Rote Rose (b) Gelbe Rose

Abbildung 4.2: Zwei Bilder mit ähnlichen Motiven, aber verschiedenen Farben

Die Entscheidung, ob zwei Bilder ähnlich sind, hängt somit immer von dem Standpunkt der Anfrage ab, wie auch das folgende Beispiel in Abbildung 4.3 verdeutlicht. Die Bilder (a) und (b) sind ähnlich hinsichtlich ihres Motivs, während die Bilder (b) und (c) bezüglich des verwendeten Stils vergleichbar sind. Bild (a) und Bild (c) sind einander aber nicht ähnlich.

(a) Foto eines Champignons



(b) Skizze eines Champignons[11]

(c) Skizze einer Blüte[12]

Abbildung 4.3: Drei Bilder, die sich hinsichtlich Motiv bzw. Stil ähneln[13]

Zusammenfassend ist festzustellen, dass die Aufgabe, die Ähnlichkeit von Bildern maschinell beurteilen zu lassen, ein schwer automatisierbarer Prozess ist. Was für den Menschen einfach ist, ist für einen Computer, dem jegliches Abstraktionsvermögen fehlt, nicht zu erfassen. Daher ist es notwendig, durch den Einsatz von Bildverarbeitungsalgorithmen den Mangel an Abstraktionsvermögen so gut wie möglich auszugleichen. Es muss eine Möglichkeit gefunden werden, die verschiedenen Variationen ähnlicher Bilder messbar zu machen. Dazu werden im Kapitel 5 Ähnlichkeitsgrade definiert.

4.2 Konzeption geeigneter Algorithmen

Um anhand verschiedener Ähnlichkeitsgrade die Gleichheit von Bildern zu messen, sind unterschiedliche Algorithmen zu implementieren.

Für einige Anwendungsfälle wird es reichen, „Pixel mit Pixel“-Vergleiche durchzuführen. Dabei ist zu beachten, dass derartige Vorgehensweisen besonders bei großen Bildern nicht sehr effizient sind. In diesem Fall ist es ratsam, die zu vergleichenden Bilder auf eine gemeinsame Größe zu skalieren. Dabei können die Bilder auch verkleinert werden, um eine höhere Rechengeschwindigkeit zu erzielen.

Sobald Bilder in unterschiedlichen Ausschnitten vorliegen, wird ein pixelbasierter Vergleich nicht mehr ausreichend sein. In diesem Fall sind andere, im Rahmen der Bildverarbeitung bereits bekannte Algorithmen zu finden und zu implementieren.

[11] Quelle: http://www.kunstnet.de/werk/119663-champignon-ii
[12] Quelle: http://www.kreativling.at/?p=209
[13] Nach [Sch10]

4.3 Anforderungen an den Prototyp

Es besteht die Aufgabe, einen plattformunabhängigen Prototyp zur Analyse ähnlicher Bilder zu entwickeln. Die dabei entstehende Software umfasst verschiedene Grundfunktionalitäten. Zum Einen ist es möglich, zwei Bilder hinsichtlich der zu definierenden Ähnlichkeitsgrade zu analysieren. Zum Anderen kann der Benutzer ein Bild vorgeben, um ein Verzeichnis des Dateisystems nach identischen und ähnlichen Bildern durchsuchen zu lassen. Dabei besteht die Möglichkeit, den Ähnlichkeitsgrad, nach dem gesucht werden soll, anzugeben.

Die Software besitzt eine grafische Oberfläche, bei der besonderer Wert auf die benutzerfreundliche und intuitive Bedienung gelegt wird. Eine hohe Bedeutung wird auch der Funktionalität der zum Einsatz kommenden Algorithmen beigemessen, um eine möglichst geringe Fehlerrate[14] zu erzielen. Im Vordergrund steht auch eine gute Zuverlässigkeit des Prototyps – unvorhergesehene Ausfälle sollen vermieden werden. Trotz der rechenaufwendigen Vergleichsalgorithmen, die zum Teil zum Einsatz kommen, ist auf eine angemessene Effizienz zu achten. Gerade weil derartige Algorithmen verwendet werden, ist die Effizienz aber nicht als wichtigste Anforderung zu verstehen. Da das Programm im Rahmen einer Studienarbeit entsteht, ist es ebenso wichtig, zu gewährleisten, dass der Programmiercode leicht modifizierbar und übertragbar ist. Programmteile müssen auf schnellem Weg zu analysieren, zu entfernen und auszutauschen sein.

Aus den zuvor getroffenen Aussagen ergeben sich folgende Qualitätsanforderungen:

	Sehr gut	**Gut**	**Normal**	**Nicht relevant**
Funktionalität	X			
Zuverlässigkeit		X		
Benutzbarkeit	X			
Effizienz			X	
Änderbarkeit		X		
Übertragbarkeit		X		

Tabelle 1: Software-Qualitätsmerkmale nach DIN ISO 9126

[14] Die Anzahl der Bilder, die bezüglich eines Ähnlichkeitsgrades als ähnlich erkannt werden, ohne tatsächlich ähnlich zu sein

4.4 Analyse verschiedener möglicher Programmiersprachen

Um eine geeignete Programmiersprache zur Implementierung des Prototyps zu finden, sind zunächst die Vor- und Nachteile verschiedener Programmiersprachen abzuwägen.
Mit einer prozeduralen Programmiersprache wie C ist aufgrund der Hardwareorientierung eine hohe Effizienz bei der Umsetzung von Algorithmen gegeben, wie sie bei der Bildverarbeitung eingesetzt werden. Für benutzerfreundliche grafische Oberflächen ist C jedoch ungeeignet. Objektorientierte Programmiersprachen wie Java oder C# hingegen erlauben das einfache Erstellen von Programmoberflächen. Zudem profitieren sie gegenüber C durch den leicht les- und erweiterbaren Quellcode. Während C# flexibler und z.T. umfangreicher[15] ist, überzeugt Java vor allem durch die Plattformunabhängigkeit, die damit verbundene Portabilität sowie die weitere Verbreitung.

[15] z.B. Möglichkeit für Structs, Referenzparameter, Attribute usw.

5 Lösungskonzeption

An dieser Stelle werden die in Kapitel 4 dargelegten Problemstellungen gelöst.

5.1 Definition von Ähnlichkeitsgraden

Für das Problem, das in Kapitel 4 unter dem Gesichtspunkt „Wann sind Bilder ähnlich?“ betrachtet wurde, ist eine Lösung zu finden. Dazu werden konkrete Ähnlichkeitsgrade definiert, die angeben, in welcher Hinsicht sich zwei Bilder ähneln. Der entstehende Prototyp vergleicht die übergebenen Bilder hinsichtlich dieser Definitionen, damit die Ähnlichkeit von Bildern messbar wird. Zu jedem Ähnlichkeitsgrad wird daher ein Verfahren beschrieben, mit dem es möglich ist, Bilder bezüglich dieser Kriterien zu vergleichen, um gleichzeitig das unter 4.2 beschriebene Problem zu lösen. Zur Veranschaulichung werden die möglichen Variationen anhand von Bildern beispielhaft demonstriert.

Abbildung 5.1: Das Originalbild

5.1.1 Ähnlichkeitsgrad 1: Identische Bilder mit 100% Übereinstimmung

Bilder, die unter diese Definition des Ähnlichkeitsgrades fallen, stimmen hinsichtlich ihrer Farbe, Größe, Motiv und Ausschnitt überein. Sie unterscheiden sich weder durch Dateiformat noch durch den verwendeten Kompressionsgrad.

Um zwei Bilder hinsichtlich dieses Ähnlichkeitsgrades zu analysieren, wird ein klassischer Bildvergleich durchgeführt. Dabei werden die zu vergleichenden Bilder pixelweise durchlaufen und für jedes Pixel überprüft, ob der Farbwert des Originalbildes mit dem Farbwert des

Vergleichsobjekts übereinstimmt. Sobald bei diesem Vergleich in mindestens einem Pixel eine Abweichung auftritt, werden die zu vergleichenden Bilder als ungleich gewertet.

5.1.2 Ähnlichkeitsgrad 2: Identische Bilder mit Abweichungen

In diese Kategorie fallen alle Bilder, die hinsichtlich ihres Ausschnitts, Motiv und Größe identisch sind, aber Abweichungen in ihren Kompressionsgraden sowie Helligkeits- und Farbwerten aufweisen.

(a) Originalbild

(b) Erhöhte Helligkeitswerte

(c) Verringerte Helligkeitswerte

(d) Erhöhter Blauanteil

(e) Anderer Kompressionsgrad

(f) Veränderte Farbverteilung

Abbildung 5.2: Beispielbilder für Ähnlichkeitsgrad 2 – Die Bilder können in ihren Farb- und Helligkeitswerten abweichen, sowie in einem anderen Kompressionsgrad vorliegen.

Analog zum klassischen Bildvergleich wird hier für jedes Pixel überprüft, ob der Farbwert des Originalbildes mit dem des Vergleichsobjektes übereinstimmt. In diesem Fall wird aber für jedes Pixel die Differenz zwischen den Farbwerten bestimmt und überprüft, ob diese Abweichung unter einem definierten Toleranzwert liegt. Ist dies der Fall, wird das Bild weiter durchlaufen. Die Pixel, deren Farbwertdifferenz den Schwellwert überschreiten, werden gezählt. Wenn die Anzahl dieser Pixel weniger als 1% aller Pixel beträgt, dann werden die Bilder als identisch gewertet.

Erweiterung des 2. Ähnlichkeitsgrades

Da der beschriebene Algorithmus Bilder mit zu starken Abweichungen nicht berücksichtigt, soll ein weiteres Verfahren mithilfe von Diskreter Kosinustransformation (DCT) implementiert werden, mit dem diese Schwäche behoben wird.

(a) Originalbild (b) Vergleichsbild

Abbildung 5.3: Beispielbilder für Ähnlichkeitsgrad 2 nach Erweiterung durch DCT – Das stark verschwommene Vergleichsbild würde ohne Einsatz von DCT nicht als Treffer gewertet werden. Erfolgt ein zusätzlicher Vergleich mit DCT, dann werden die Bilder als ähnlich erkannt.

Dazu werden Original- und Vergleichsbilder mithilfe der DCT stark komprimiert und anschließend verglichen. Die DCT ist eine reellwertige diskrete lineare orthogonale Transformation, die das Bild in Frequenzen zerlegt. Damit ist es möglich, höherwertige und niederwertige Frequenzen sowie den Gleichanteil gesondert zu betrachten. Um die Bilder mittels DCT zu komprimieren, existieren verschiedene Lösungsansätze. In diesem Fall wird die eindimensionale DCT-II einmal horizontal und anschließend vertikal auf das Bild angewendet.

Definition 5.1 (Diskrete Kosinustransformation (DCT-II)[16])**.**

$$X_k = \sum_{n=1}^{N-1} x_n \cdot \cos\left[\frac{\pi}{N} \cdot n\left(k+\frac{1}{2}\right)\right] \text{ mit } k = 0,...,N-1$$

Dabei gilt:
N ... Anzahl der Frequenzen
n ... Bildindex
k ... Ergebnisindex
x_n ... Bild
X_k ... Ergebnismatrix

Jedes Bild, für das eine DCT durchgeführt werden soll, wird auf *NxN* skaliert. Bei Betrachten der Berechnungsvorschrift fällt zudem auf, dass der Ausdruck innerhalb des *cos*- Terms lediglich von der Anzahl der Frequenzen (N) abhängt. Somit muss dieser Ausdruck nur einmal innerhalb eines Vergleichdurchlaufs berechnet werden. Das Ergebnis wird in einer Kosinustabelle gespeichert, in der dann während des Vergleichs nur noch an den entsprechenden Stellen nachgeschaut werden muss. Als Ergebnis der DCT entstehen Matrizen aus DCT-Koeffizienten, die miteinander verglichen werden können. Dafür wird zunächst die Differenz aus den Gleichanteilen und anschließend den Wechselanteilen der zu vergleichenden Fotos gebildet und mit einem Schwellwert verglichen.

5.1.3 Ähnlichkeitsgrad 3: Das Vergleichsbild ist ein Ausschnitt aus dem Originalbild

Bilder gehören dem Ähnlichkeitsgrad 3 an, wenn das eine Bild als Ausschnitt in dem anderen Bild enthalten ist. Minimale Abweichungen zwischen den Helligkeitswerten von Ausschnitt und Bild sind dabei erlaubt. Allerdings darf der Ausschnitt nicht vergrößert oder gedreht vorliegen.

[16] Nach [Wik11]

(a) Originalbild

(b) Ausschnitt aus dem Originalbild

Abbildung 5.4: Beispielbilder für Ähnlichkeitsgrad 3 – Das Vergleichsbild ist ein Ausschnitt aus dem Originalbild

Mithilfe von Template Matching ist es möglich, den Ausschnitt eines Bildes in einem anderen zu suchen. Dabei wird das gesuchte Bildmuster, also das Template, über das Bild bewegt und die Differenz zu dem darunterliegenden Originalbild berechnet. Die Stellen, an denen Template und Bild übereinstimmen, werden markiert.

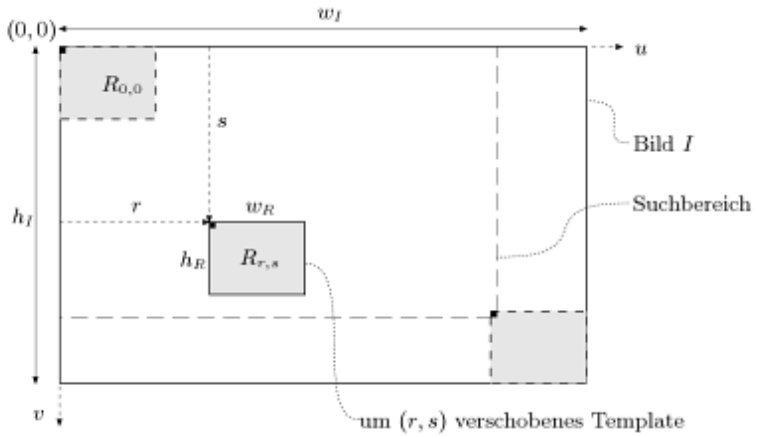


Abbildung 5.5: Geometrie des Template Matching[17]. Das Referenzbild R wird über das Bild I mit dem Offset (r,s) bewegt. Der maximale Suchbereich für den Vergleich wird durch die Größe des Originalbildes $(w_I \cdot h_I)$ und des Referenzbildes $(w_R \cdot h_R)$bestimmt.

[17] Aus: [BB06], S. 412

Um zu ermitteln, wo eine hohe Übereinstimmung zwischen Bildmustern besteht, wird der Abstand zwischen dem verschobenen Referenzbild R und dem Ausschnitt aus dem Originalbild I an jeder Position r,s berechnet. Um diesen Abstand zu berechnen, gibt es verschiedene Vorgehensweisen. Die für das Template Matching effizienteste Methode ist die Verwendung des lokalen Korrelationskoeffizienten.

Definition 5.2 (Lokaler Korrelationskoeffizient[18])**.**

$$C_L(r,s) = \frac{\sum\limits_{(i,j)\in R} \big(I(r+i,s+j)\cdot R(i,j)\big) - K\cdot \bar{I}(r,s)\cdot \bar{R}}{\left[\sum\limits_{(i,j)\in R} \big(I(r+i,s+j)\big)^2 - K\cdot(\bar{I}(r,s))^2\right]^{1/2}\cdot \sigma_R}$$

Dabei gilt:

$$K = |R| \text{ (Anzahl der Elemente im Template R)}$$

$$\bar{I}(r,s) = \frac{1}{K}\sum_{(i,j)\in R} I(r+i,s+j)\ ,\ \bar{R} = \frac{1}{K}\sum_{(i,j)\in R} R(i,j) \text{ (Durchschnittswerte)}$$

$$\sigma_R = \sqrt{\sum_{(i,j)\in R} (R(i,j))^2 - K\cdot \bar{R}^2} \text{ (Wurzel aus der Varianz der Werte im Template R)}$$

Da sowohl $\bar{R}$ als auch σ_R nur einmal berechnet werden müssen und der lokale Durchschnittswert der Bildfunktion $\bar{I}(r,s)$ bei der Ermittlung der Abweichungen zunächst vernachlässigt werden kann, kann der Ausdruck aus Definition 5.2 in einer einzigen, gemeinsamen Iteration berechnet werden. Damit ist es möglich, das Template Matching auf effizientem Weg umzusetzen. Dabei ist zu beachten, dass die benötigte Rechenzeit stark ansteigen wird, wenn die Differenz zwischen Template- und Vergleichsbildgröße zu groß wird. Der Ausschnitt aus dem Bild, nach dem gesucht werden soll, sollte daher nicht zu klein sein.

[18] Nach [BB06], S.415-416

5.1.4 Ähnlichkeitsgrad 4: Bilder mit ähnlicher Farbverteilung

In diese Kategorie werden alle Bilder eingeordnet, die sich hinsichtlich ihres Motivs unterscheiden, aber dennoch einen ähnlichen Farbverlauf aufweisen.

(a) Originalbild (b) Bild mit ähnlichen Farben

Abbildung 5.6: Beispielbilder für Ähnlichkeitsgrad 4 – Bilder mit ähnlichen Farben.

Damit die Bilder bezüglich ihrer Farben analysiert werden können, ohne zu berücksichtigen, wie die Farben konkret verteilt sind, wird eine Art Histogramm-Vergleich durchgeführt. Dazu werden verschiedene Farbkategorien angelegt und für jedes Pixel überprüft, in welche der Kategorien es aufgrund seiner Farbwerte gehört. Anhand der Häufigkeitsverteilung in den einzelnen Kategorien können die Bilder dann hinsichtlich ihrer Farben verglichen werden.

5.2 Steigerung der Verarbeitungsgeschwindigkeit durch Verwendung von Threads

Bei der Implementierung der im vorhergehenden Punkt genannten Algorithmen wird schnell auffallen, dass die Verarbeitungsgeschwindigkeit zum Teil nicht optimal ist. Aus diesem Grund wird an verschiedenen Stellen im Programm Multithreading eingesetzt. Dadurch, dass mehrere Aufgaben parallel ausgeführt werden, kann die Geschwindigkeit der Abarbeitung der einzelnen Algorithmen gesteigert werden.

5.3 Auswahl der Programmiersprache und Frameworks

Für die Implementierung des Prototyps wird Java verwendet, um die Vorteile der Objektorientierung, Plattformunabhängigkeit und Flexibilität zu nutzen. In den frühen Java Versio-

nen existierte im Zusammenhang mit Bildern ausschließlich das Abstract Windowing Toolkit (AWT), das für die Darstellung von Bilddaten verwendet wurde. Dabei fehlte jedoch jegliche Funktionalität für die Manipulation von Bildern. Des Weiteren unterstützte AWT zunächst nur eine sehr kleine Anzahl an Bildformaten. Um diese Probleme zu minimieren, wurde Java 2D mit ersten Klassen für größere grafische Operationen entwickelt. Daraus folgte eine Unterstützung von zusätzlichen Bildformaten für den lesenden Zugriff. Mit Java 2D bekam der Programmierer die Möglichkeit, über eine einfach gehaltene Programmierschnittstelle Bilder zu analysieren und zu manipulieren. Als Erweiterung von Java 2D folgte schließlich mit dem Java Advanced Imaging Toolkit (JAI) eine umfangreiche Sammlung von Bildverarbeitungsmethoden. JAI bietet eine Vielzahl von verschiedenen Operationen, wie z.B. Punkt-, Flächen, Geometrie- und Dateioperationen. Um den Prototyp zu entwickeln, kommt zusätzlich ImageJ zum Einsatz. Dies ist ein Open-Source-Projekt von Wayne Rusband. Die Software enthält Basisoperationen zum Bearbeiten von Bildern und ist durch selbst geschriebene Java Plug-ins erweiterbar. Des Weiteren können lediglich die ImageJ Bibliotheken verwendet werden, um die Funktionalitäten in eine eigene Software einzubinden. Mithilfe dieser Frameworks ist es möglich, mit Java einen Prototyp für eine Bildvergleichssoftware zu implementieren.

6 Implementierung

Dieses Kapitel widmet sich der eigentlichen Implementierung des Prototyps. Es werden verschiedene Diagrammtypen verwendet, um Struktur und Dynamik der entwickelten Software zu beschreiben, sowie Quellcodeausschnitte, um auf die Besonderheiten bei der Implementierung hinzuweisen.

6.1 Darstellung des statischen Aufbaus anhand von Klassendiagrammen

Um die statischen Eigenschaften des Systems aufzuzeigen, kommen Klassendiagramme zum Einsatz.

Klassendiagramm des `ImageComparers`

Die `ImageComparer`-Klasse ist die zentrale Einheit der Software. Sie enthält die `main()`-Methode und eine interne Klasse, den `CompareThread`. Dieser führt den eigentlichen Vergleich aus.

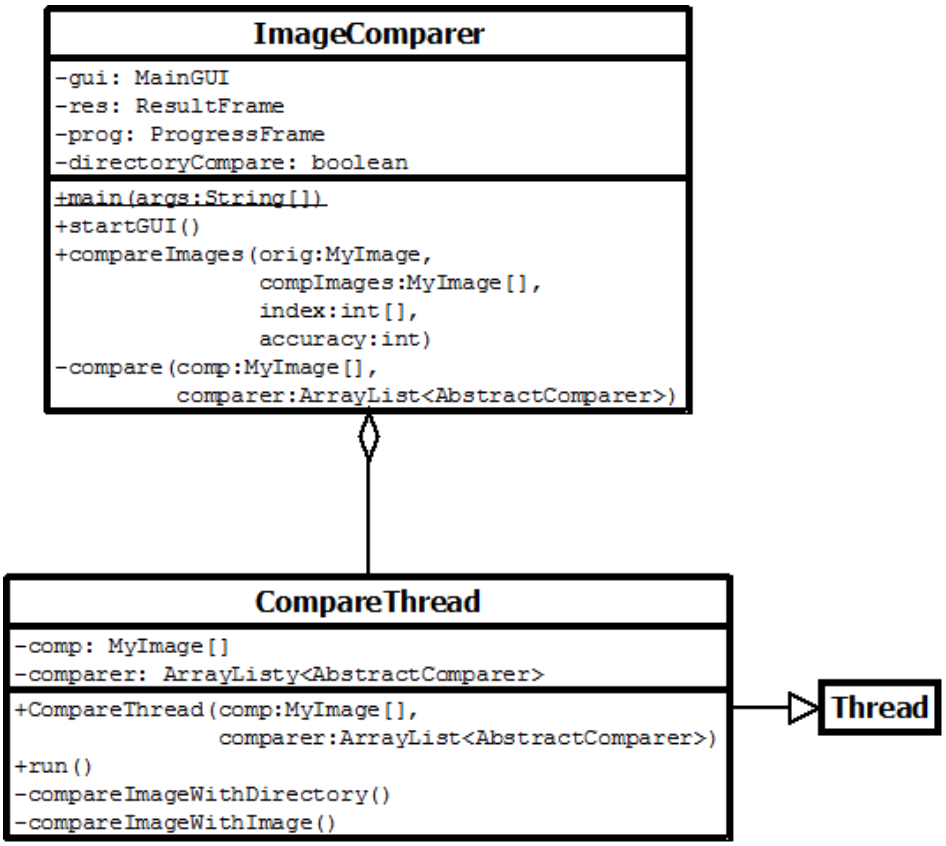


Abbildung 6.1: Klassendiagramm des `ImageComparers`

Die wichtigsten Klassen der GUI

Die grafische Oberfläche besteht aus einer Hauptklasse, der `MainGUI`, die zum Aufbau des Hauptfensters dient. Sie ist aus verschiedenen Klassen, die jeweils von der Klasse `JPanel` abgeleitet sind, aufgebaut. Zusätzlich wird die Oberfläche ergänzt durch ein `ResultFrame` – das Ergebnisfenster – und ein `ProgressFrame`, das den Fortschritt des Vergleichs anzeigt.

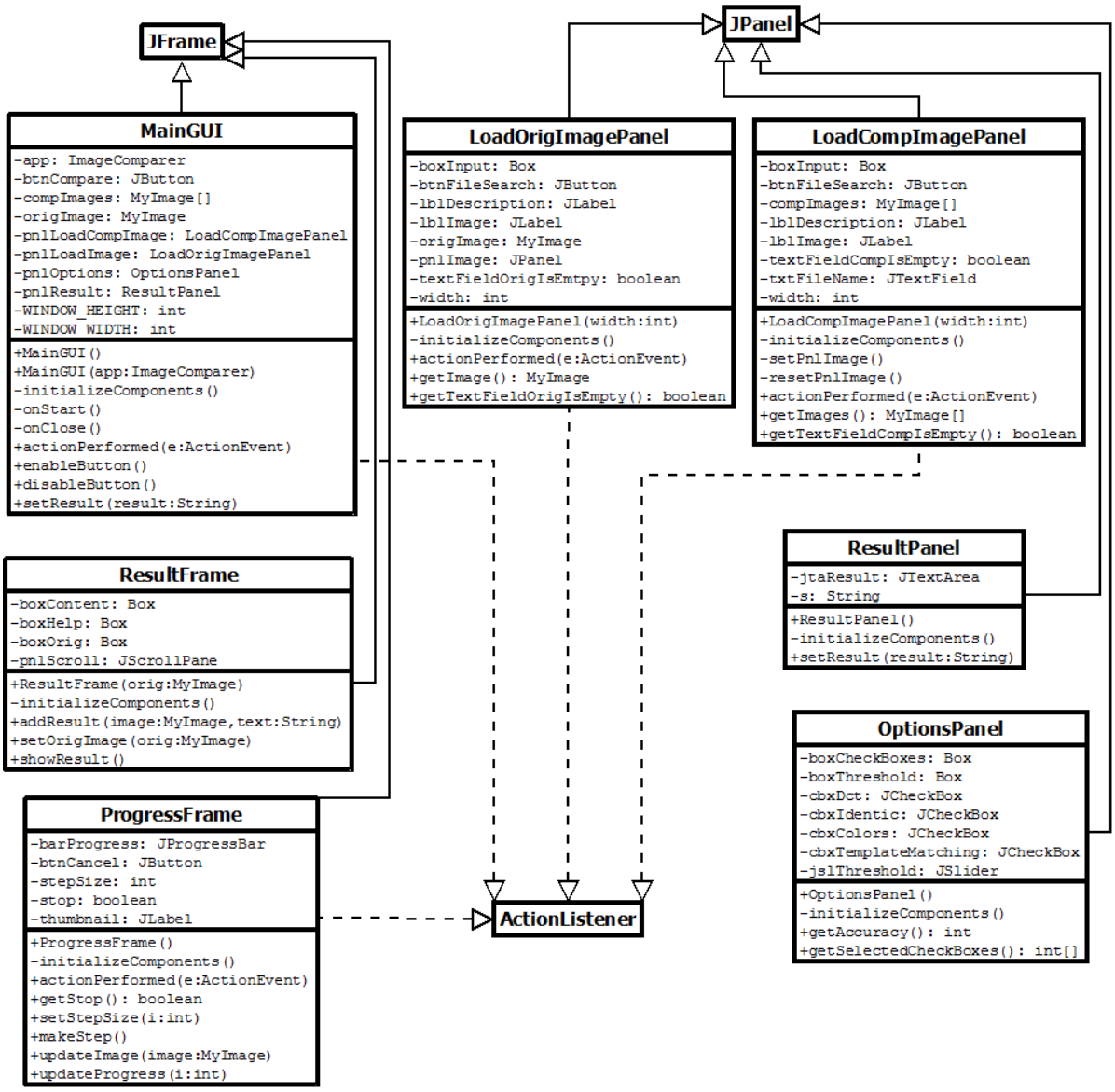


Abbildung 6.2: Klassendiagramm mit den wichtigsten Klassen der GUI

Die `Comparer`-Klassen

Es gibt eine abstrakte Klasse, den `AbstractComparer`, von dem alle separaten `Comparer`-Klassen erben. Zusätzlich enthält jeder `Comparer` eigene Attribute und Methoden.

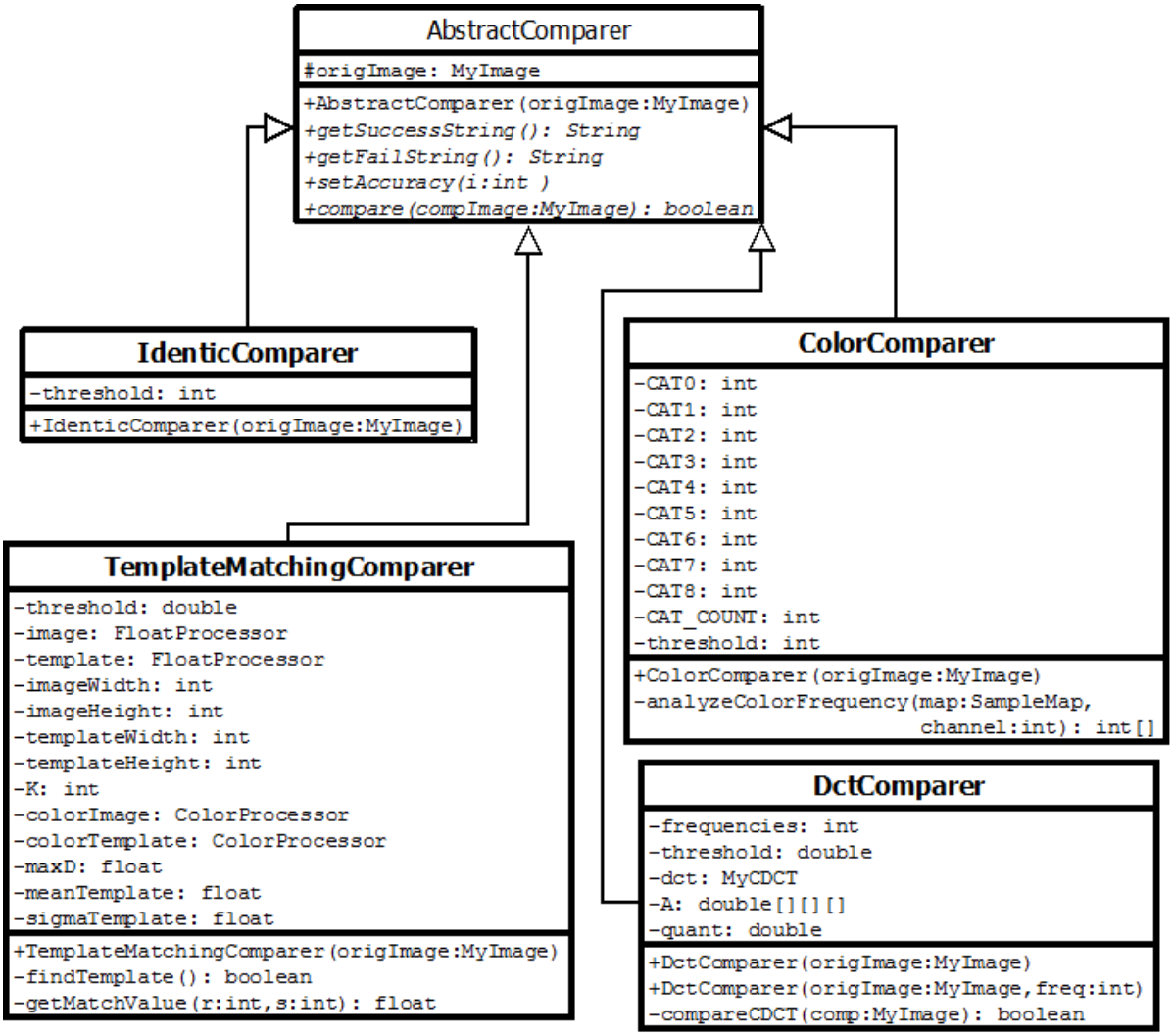


Abbildung 6.3: Klassendiagramm mit den `Comparer` Klassen

6.2 Dynamischer Ablauf – Anwendungsfall-, Aktivitäts- und Sequenzdiagramme

Mithilfe eines Anwendungsfalldiagramms ist es möglich, den Ablauf des Systems aus der Black-Box-Sicht des Anwenders zu beschreiben.

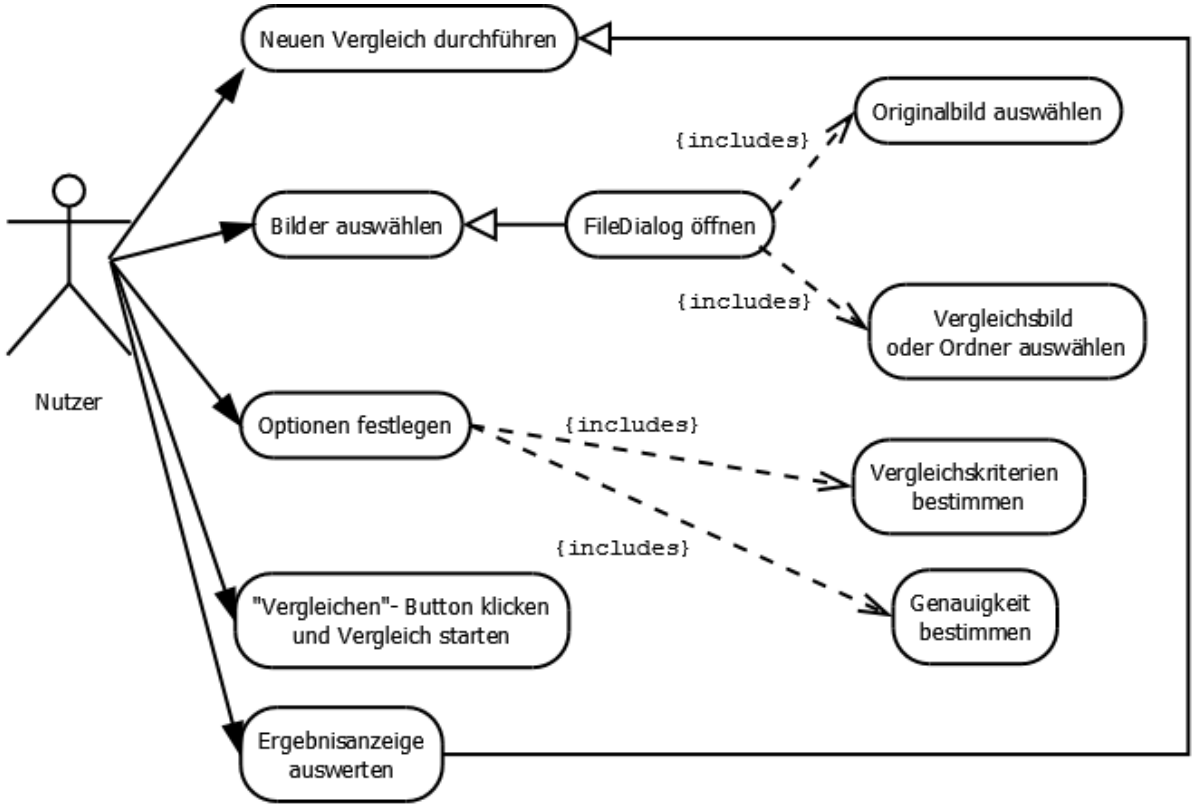


Abbildung 6.4: Anwendungsfalldiagramm

Das Diagramm in Abbildung 6.4 verdeutlicht die Tätigkeiten, die der Nutzer über die Oberfläche durchführt. Er hat die Möglichkeit, Bilder für den Vergleich auszuwählen, Vergleichskriterien festzulegen und den eigentlichen Vergleich zu starten.

Einen genaueren Überblick darüber, welche Schritte auf Systemseite durchzuführen sind, ermöglicht ein Aktivitätsdiagramm.

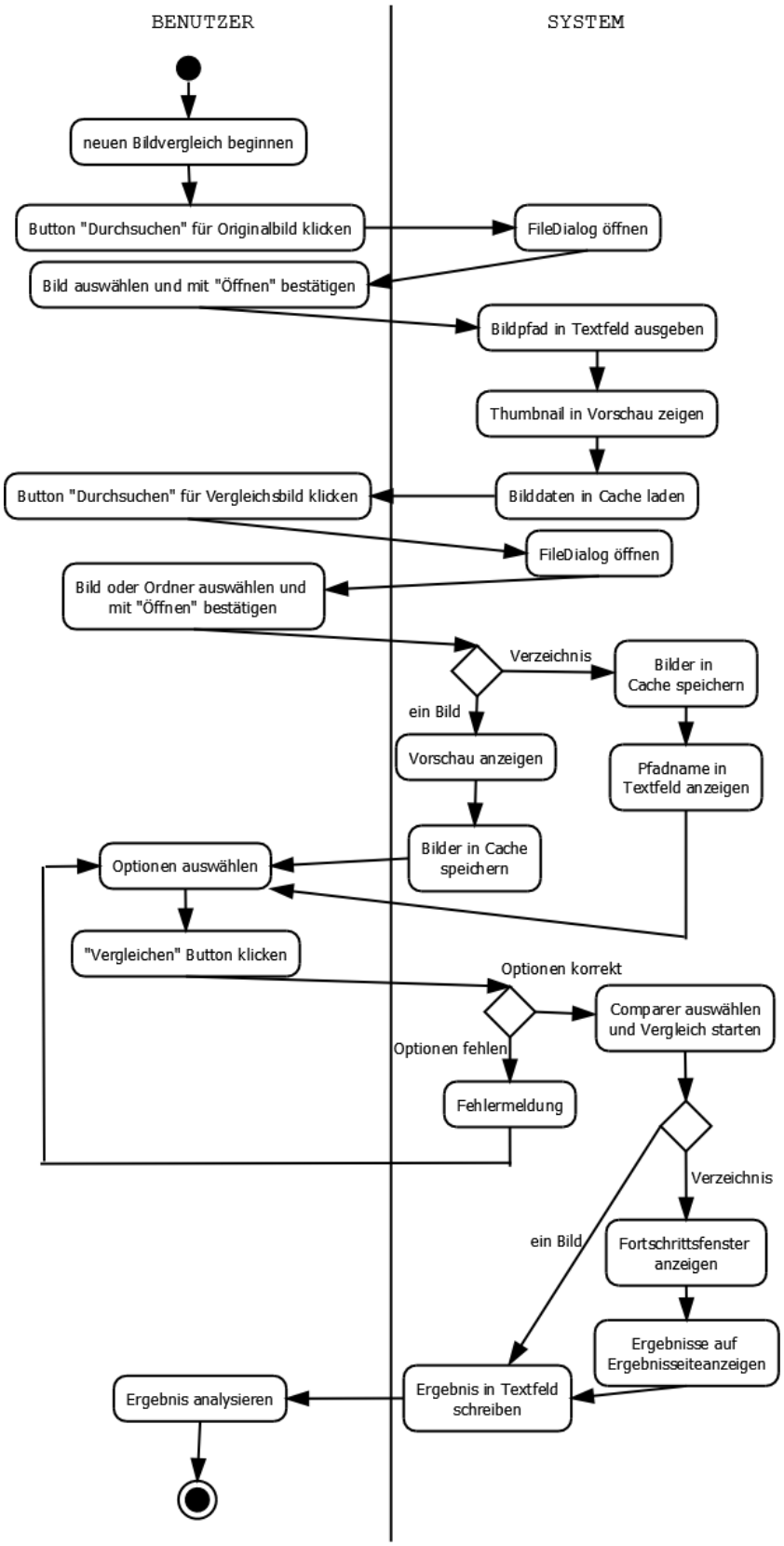


Abbildung 6.5: Aktivitätsdiagramm

Das in Abbildung 6.5 dargestellte Diagramm zeigt den genauen Ablauf der Software sowohl auf Anwender- als auch auf Systemseite.
Ein Sequenzdiagramm wird verwendet, um die Interaktion zwischen mehreren Kommunikationspartnern – im Regelfall die Objekte von Klassen – zu verdeutlichen. Im Vordergrund stehen dabei vor allem der zeitliche Ablauf der verschiedenen Aktionen.

Sequenzdiagramm zum Szenario „Bild mit Bild Vergleich“

Das in Abbildung 6.6 gezeigte Sequenzdiagramm verdeutlicht den zeitlichen Ablauf, falls ein Bild mit einem anderen Bild verglichen werden soll. In diesem Fall ist weder ein `ProgressFrame` noch ein `ResultFrame` notwendig.

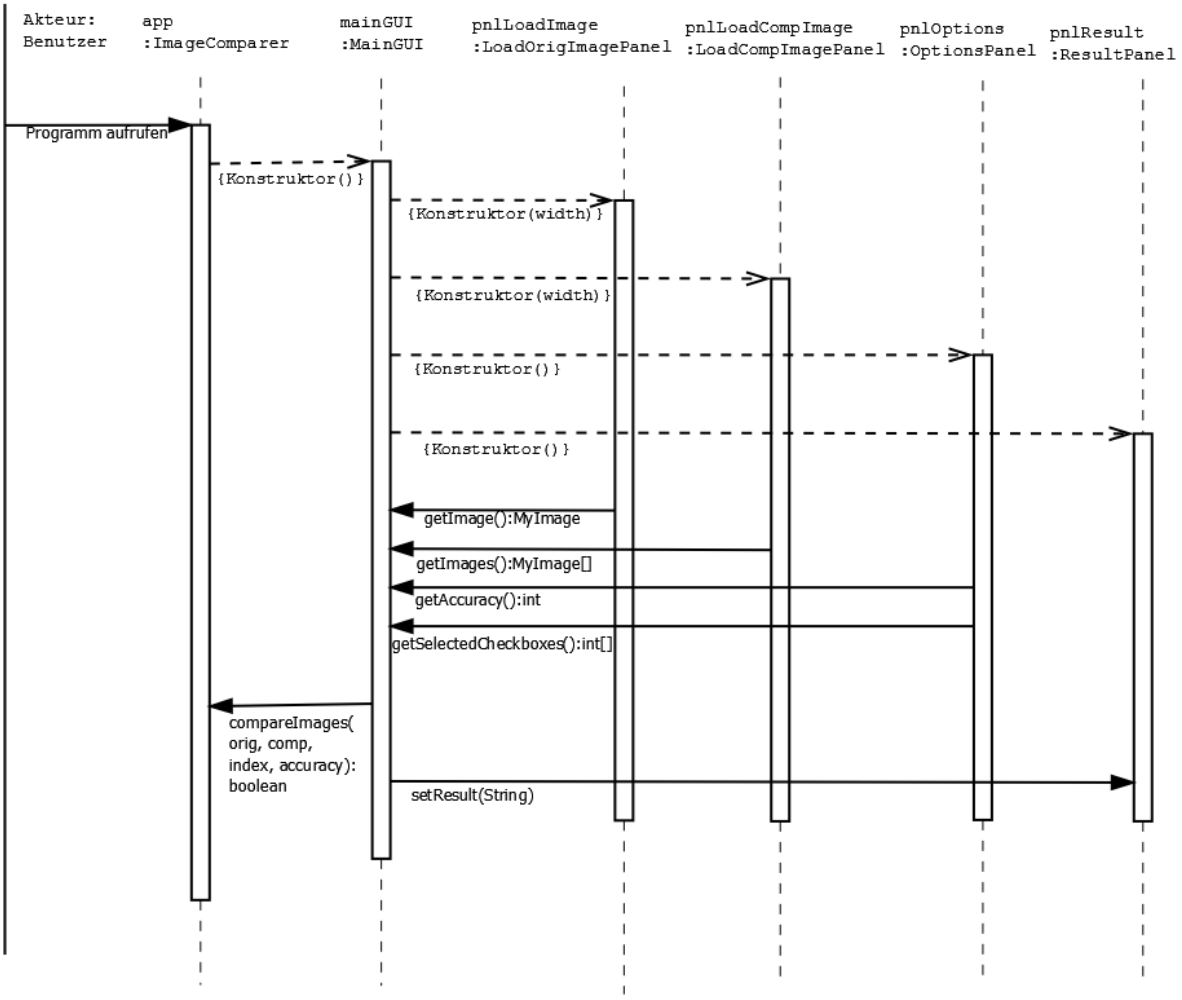


Abbildung 6.6: Sequenzdiagramm zum Szenario „Bild mit Bild Vergleich“

Sequenzdiagramm zum Szenario „Bild mit Verzeichnis Vergleich“

Im Gegensatz zu dem Sequenzdiagramm aus Abbildung 6.6 zeigt die Abbildung 6.7 das Sequenzdiagramm für den Fall, dass ein Bild mit einem gesamten Verzeichnis verglichen wird.

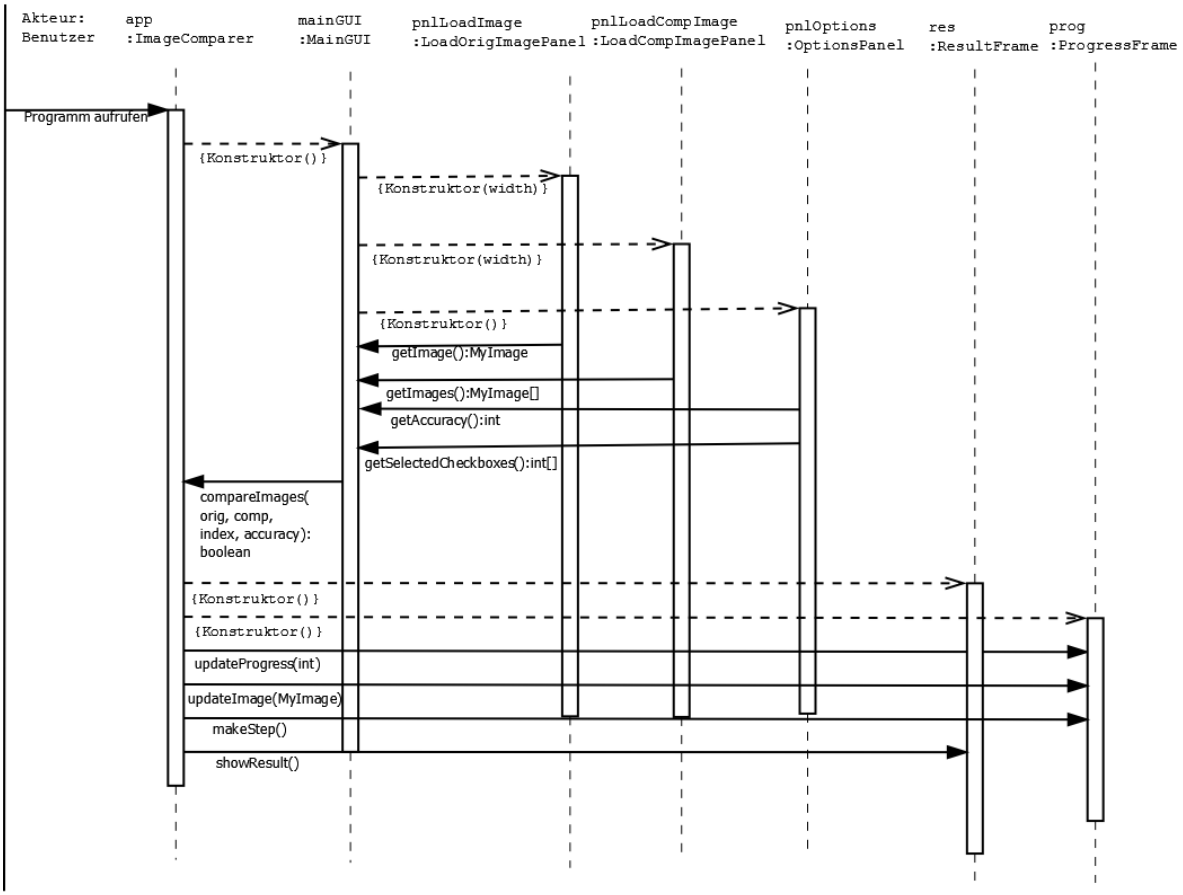


Abbildung 6.7: Sequenzdiagramm zum Szenario „Bild mit Verzeichnis Vergleich“

6.3 Implementierungsdetails

Wichtige Quellcodeausschnitte sowie Besonderheiten bei der Implementierung werden in diesem Abschnitt beschrieben.

6.3.1 Start des Programms

Der `ImageComparer` bildet die zentrale Klasse des Programms. Hier wird mit dem Aufruf der `main()`-Methode zunächst die Erzeugung der Oberfläche initiiert.

```
1  private MainGUI gui;
2
3  public static void main(String[] args)
4  {
5          new ImageComparer().startGUI();
6  }
7
8  public void startGUI()
9  {
10         gui = new MainGUI(this);
11         gui.setVisible(true);
12 }
```

Listing 6.1: Start des Programms über den ImageComparer

Im nächsten Schritt wird das Erscheinungsbild der `MainGUI` erzeugt. Diese Klasse repräsentiert das Hauptfenster des Programms.

6.3.2 Das Hauptfenster

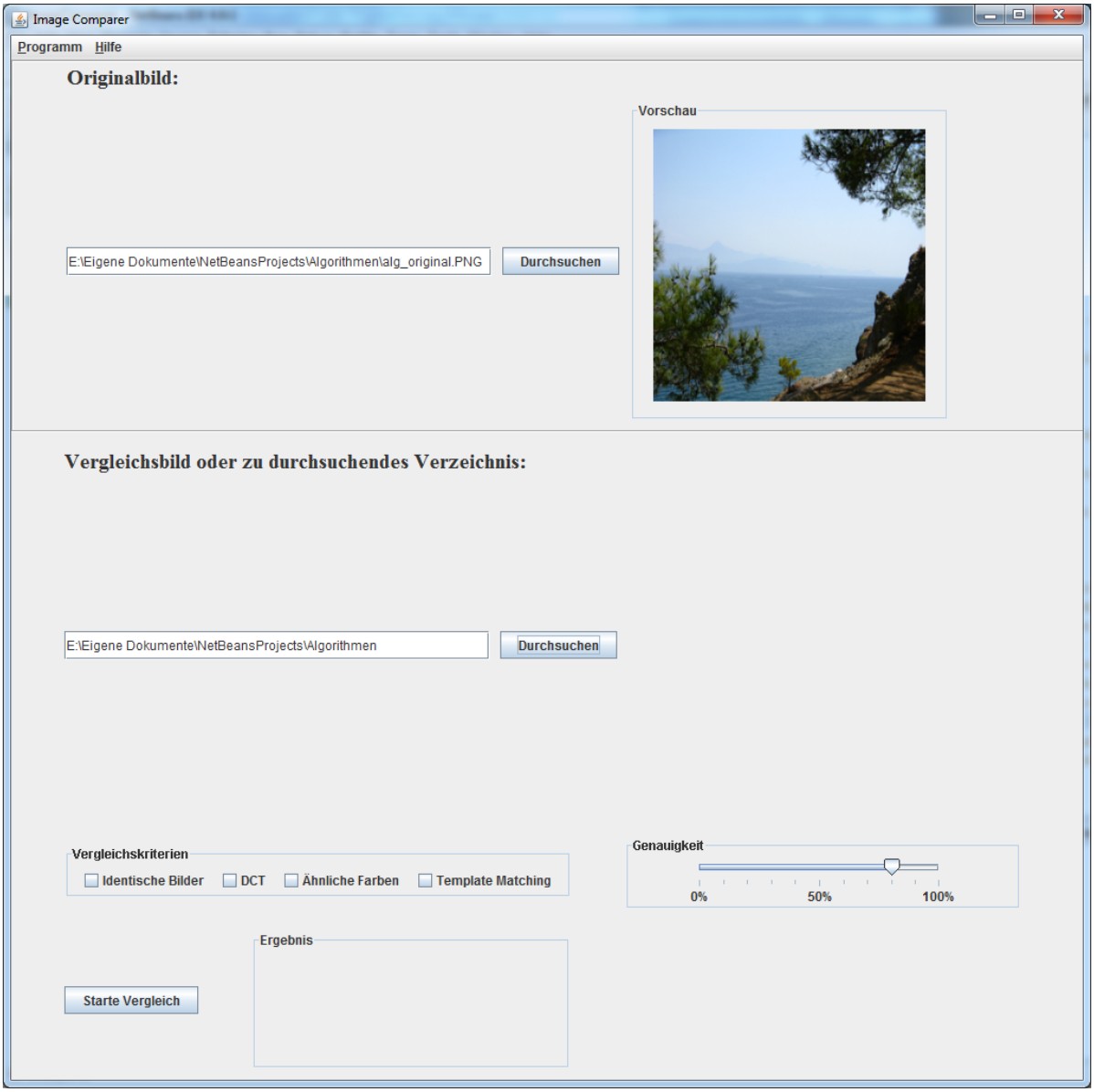


Abbildung 6.8: Das Hauptfenster

Die Oberfläche besteht aus verschiedenen Panels, die an den entsprechenden Stellen in der `MainGUI` eingefügt werden. Dazu gehören das `LoadOrigImagePanel`, das `LoadCompImagePanel`, das `ResultPanel` sowie das `OptionsPanel`.
Hier kann der Benutzer ein Bild auswählen, das als Vergleichsvorlage verwendet wird, sowie ein weiteres Bild oder einen ganzen Ordner, mit dem verglichen werden soll. Zusätzlich ist es möglich, Vergleichskriterien und eine Genauigkeit einzustellen. Dabei repräsentieren die

Vergleichskriterien die einzelnen Ähnlichkeitsgrade, die unter Abschnitt 5.1 definiert wurden. Mit einem Klick auf den Vergleichen-Button bestätigt der Nutzer seine Eingaben und der eigentliche Vergleich beginnt.

```
1  private ImageComparer app;
2  private MyImage origImage;
3  private MyImage[] compImages;
4
5  public void actionPerformed(ActionEvent e)
6  {
7          int[] index = pnlOptions.getSelectedCheckBoxes();
8
9          if ((index[0]==0) && (index[1]==0) && (index[2]==0) && (index[3]==0))
10         {
11             JOptionPane.showMessageDialog(this,
12                                 "Bitte wählen Sie Vergleichskriterien aus.",
13                                 "Fehler",
14                                 JOptionPane.ERROR_MESSAGE);
15         }
16
17         origImage = pnlLoadImage.getImage();
18         compImages = pnlLoadCompImage.getImages();
19
20         app.compareImages(origImage, compImages, index, getAccuracy());
21 }
```

Listing 6.2: Das Hauptfenster ruft `compareImages()`-Methode des `ImageComparers` auf.

Hat der Benutzer keine Vergleichskriterien ausgewählt, wird er erneut dazu aufgefordert. Anschließend wird das Originalbild in ein `MyImage`-Objekt und sämtliche Vergleichsbilder in ein Array aus `MyImage`-Objekten gespeichert. Die `MyImage`-Klasse ermöglicht es, die unterschiedlichen Typen der Bilder[19] auf einfachem Weg zu laden. Nun werden sämtliche Bilder sowie die gewählten Vergleichskriterien an den `ImageComparer` übergeben und die `compareImages()`-Methode aufgerufen.

6.3.3 Auswahl eines `Comparers` durch den `ImageComparer`

Der `ImageComparer` überprüft anhand der übergebenen Werte nun, welcher `Comparer` verwendet werden soll.

[19] Je nach verwendetem Framework bzw. nach Anwendungsgebiet kann ein Bild z.B. als `PlanarImage` oder als `BufferedImage` benötigt werden.

```
1  public void compareImages(final MyImage orig, MyImage[] comp, int[] index, int accuracy)
2  {
3
4          //Bild mit Bild vergleichen, keine Verzeichnissuche
5          if (comp.length==1)
6          {
7              directoryCompare = false;
8          }
9          else //nur bei Verzeichnissuche notwendig
10         {
11             directoryCompare = true;
12             res = new ResultFrame(orig);
13         }
14
15         // Welcher Vergleich soll durchgeführt werden?
16         ArrayList<AbstractComparer> comparer = new ArrayList<AbstractComparer>();
17         if (index[0] == 1)
18         {
19             IdenticComparer ic = new IdenticComparer(orig);
20             ic.setAccuracy(accuracy);
21             comparer.add(ic);
22         }
23         if (index[1] == 1)
24         {
25             ColorComparer cc = new ColorComparer(orig);
26             cc.setAccuracy(accuracy);
27             comparer.add(cc);
28         }
29         if (index[2] == 1)
30         {
31             TemplateMatchingComparer tmc = new TemplateMatchingComparer(orig);
32             tmc.setAccuracy(accuracy);
33             comparer.add(tmc);
34         }
35         if (index[3] == 1)
36         {
37             DctComparer dc = new DctComparer(orig);
38             dc.setAccuracy(accuracy);
39             comparer.add(dc);
40         }
41
42         //nur bei Verzeichnisssuche
43         if (directoryCompare)
44         {
45             prog = new ProgressFrame();
46             prog.setVisible(true);
47             prog.setStepSize(100/(comp.length*comparer.size()));
48         }
49         compare(comp, comparer);
50 }
```

Listing 6.3: Die `compareImages()`-Methode des ImageComparers.

Zunächst wird ermittelt, ob ein oder mehrere Bilder übergegeben wurden. Bei einem Vergleich mit einem gesamten Verzeichnis ist es notwendig, das Ergebnis in einem eigenen Fenster auszugeben, damit alle als ähnlich gewerteten Bilder angezeigt werden können. Zudem wird ein `ProgressFrame` benötigt, das anzeigt, wie weit der Vergleich bereits fortgeschritten ist. Dazu wird das Bild ausgegeben, mit dem aktuell verglichen wird. Zusätzlich kann mithilfe des Fortschrittsbalkens die restliche Dauer des Vergleichs abgeschätzt werden.

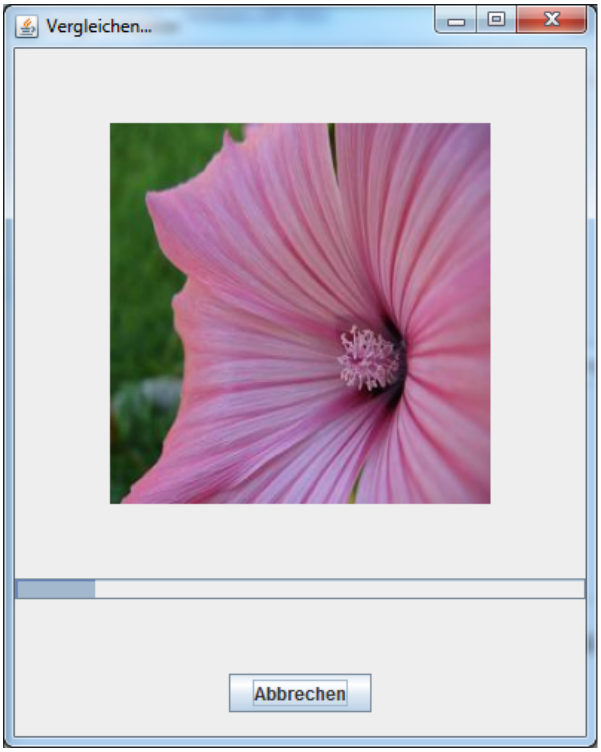


Abbildung 6.9: Das Fortschrittsfenster

Wird das Ausgangsbild hingegen nur mit einem Bild verglichen, ist ein solches Fenster nicht notwendig und das Ergebnis wird lediglich als Zeichenkette in einem Textfeld ausgegeben. Anschließend wird anhand des übergebenen `index`-Arrays ermittelt, welcher `Comparer` verwendet werden soll. Je nach Benutzereingabe werden dann die entsprechenden Objekte erzeugt, die Genauigkeit auf den vom Benutzer gewählten Wert gesetzt und jedes `Comparer`-Objekt zu einer `ArrayList` hinzugefügt. Nun wird diese Liste, sowie ein Array, das alle Bilder enthält, mit denen verglichen werden soll, an die `compare()`-Methode weitergegeben.

```
1  private void compare(MyImage[] comp, ArrayList<AbstractComparer> comparer)
2  {
3          CompareThread ct = new CompareThread(comp, comparer);
4          ct.start();
5  }
```

Listing 6.4: Die `compare()`-Methode des `ImageComparers`

Damit wird ein neuer `CompareThread` erzeugt und gestartet.

6.3.4 Der `CompareThread`

```
1  class CompareThread extends Thread
2  {
3          private final MyImage[] comp;
4          private final ArrayList<AbstractComparer> comparer;
5
6          public CompareThread(MyImage[] comp, ArrayList<AbstractComparer> comparer)
7          {
8              this.comp = comp;
9              this.comparer = comparer;
10         }
11
12         @Override
13         public void run()
14         {
15
16             // Nur zwei Bilder vergleichen
17             if (!directoryCompare)
18             {
19                 compareImageWithImage();
20             }
21             // Mit Verzeichnis vergleichen
22             else
23             {
24                 compareImageWithDirectory();
25             }
26         }
27
28 //...
```

Listing 6.5: Aufbau der internen `CompareThread`-Klasse.

Hier wird zunächst der Konstruktor und anschließend die Funktionalität der `run()`-Methode definiert. Dabei wird die `compareImageWithDirectory()`- aufgerufen, wenn das Ausgangsbild mit einem ganzen Verzeichnis verglichen werden soll. Ist dies nicht der Fall, wird die `compareImageWithImage()`-Methode aufgerufen.

```
1  private void compareImageWithDirectory()
2  {
3              int hitsCount = 0;
4              int failCount = 0;
5
6              for(int i = 0; i < comp.length; i++)
7              {
8                  if (!prog.getStop())
9                  {
10                     prog.updateImage(comp[i]);
11                     prog.updateProgress(100 / comp.length * i);
12                     for(int j = 0; j < comparer.size(); j++)
13                     {
14                         if(comparer.get(j).compare(comp[i]))
15                         {
16                             hitsCount++;
17                             res.addResult(comp[i], comparer.get(j).getSuccessString());
18                             break;
19                         }
20                         else
21                         {
22                             failCount++;
23                         }
24                      prog.makeStep();
25                     }
26                 }
27
28             }
29             if (!prog.getStop())
30             {
31                 gui.setResult("");
32                 gui.setResult(hitsCount + " von " + (hitsCount+failCount) + " Bildern sind
                       dem Ausgangsbild ähnlich.");
33                 prog.dispose();
34                 res.showResult();
35             }
36 }
```

Listing 6.6: Die `compareImageWithDirectory()`-Methode.

In dieser Methode werden alle Bilder, die sich in dem `comp`-Array – und damit in dem Verzeichnis, mit dem verglichen werden soll – befinden, durchlaufen. Dabei wird mit der Abfrage

`{if (!prog.getStop())}`

bei jedem Bild überprüft, ob der Benutzer den Vergleich abgebrochen hat. Falls dies nicht der Fall ist, wird im Fortschrittsfenster die Fortschrittsanzeige aktualisiert. Anschließend wird die Liste aller `Comparer` durchlaufen und die `compare()`-Methode des jeweiligen `Comparers` aufgerufen. Jedes als ähnlich gewertete Bild wird in das `ResultFrame` eingefügt. Dabei wird

zusätzlich der Grad angegeben, laut dem sich das Bild und das Ausgangsbild ähneln. Nachdem mit allen Bildern verglichen wurde, wird das Ergebnisfenster angezeigt.

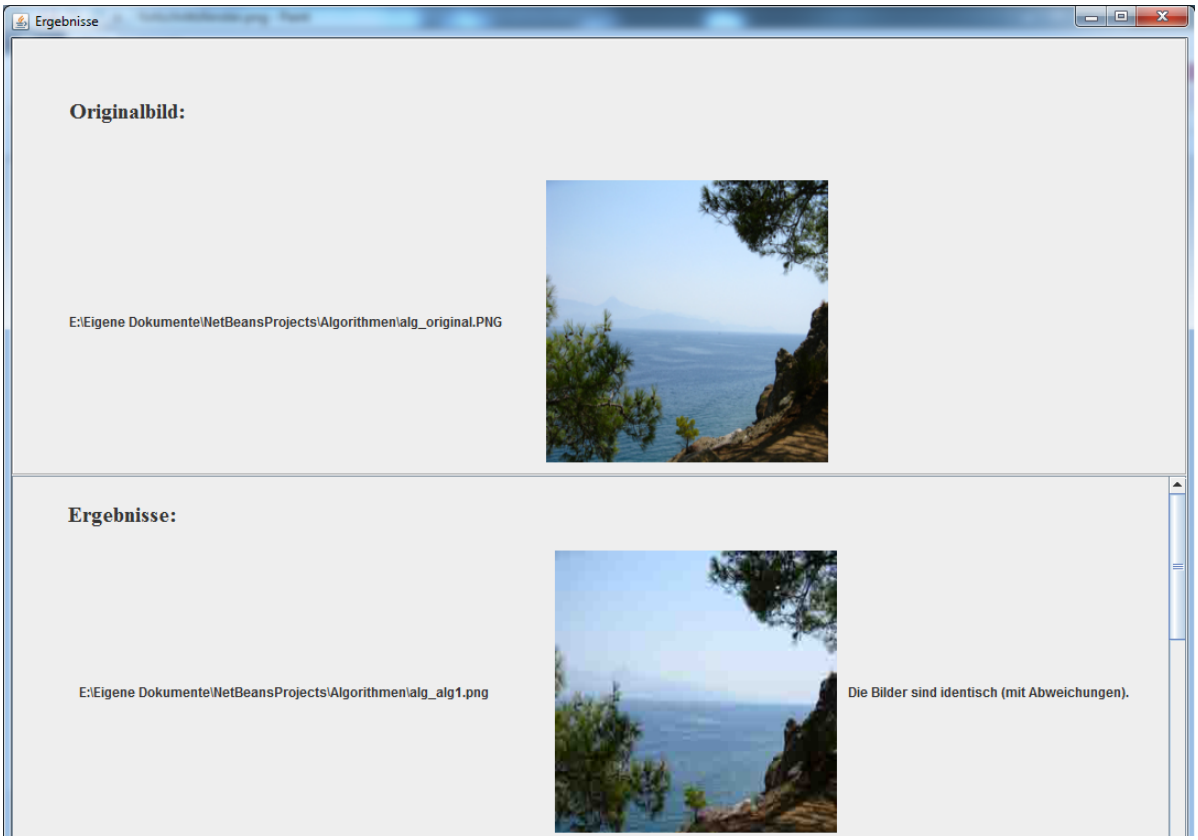


Abbildung 6.10: Das Ergebnisfenster

Die `compareImageWithImage()`-Methode ist im Gegensatz zur `compareImageWithDirectory()`-Methode recht einfach aufgebaut.

```
1  private void compareImageWithImage()
2  {
3              for(int j = 0; j < comparer.size(); j++)
4              {
5                  if(comparer.get(j).compare(comp[0]))
6                  {
7                      gui.setResult(comparer.get(j).getSuccessString());
8                  }
9                  else
10                 {
11                     gui.setResult(comparer.get(j).getFailString());
12                 }
13             }
14 }
```

Listing 6.7: Die `compareImageWithImage()`-Methode.

Diese Methode durchläuft analog zu der `compareImageWithDirectory()`-Methode zunächst die `ArrayList` aller `Comparer`. Je nachdem, ob der `Comparer` die beiden zu vergleichenden Bilder als ähnlich wertet, wird die Ausgabe in das Textfeld übertragen.

6.3.5 Die Klasse `AbstractComparer`

Alle `Comparer`-Klassen erweitern den `AbstractComparer`.

```
1  public abstract class AbstractComparer
2  {
3
4      protected MyImage origImage;
5
6      public AbstractComparer(MyImage origImage)
7      {
8          this.origImage = origImage;
9      }
10
11     public abstract String getSuccessString();
12
13     public abstract String getFailString();
14
15     public abstract void setAccuracy(int i);
16
17     public abstract boolean compare(MyImage compImage);
18
19 }
```

Listing 6.8: Der `AbstractComparer`.

Dem Konstruktor eines jeden `Comparers` wird somit das Originalbild übergeben. Dies hat den Vorteil, dass das Ausgangsbild bei einem Vergleich mit mehreren Bildern nur einmal analysiert werden muss. Zudem besitzt jedes `Comparer`-Objekt die Methoden `getSuccessString()` und `getFailString()`. Damit wird festgelegt, welche Ausgabe in dem Fall ausgegeben wird, wenn zwei Bilder als ähnlich bzw. nicht ähnlich gewertet werden. Außerdem wird jede `Comparer`-Klasse die `setAccuracy()`-Methode implementieren, damit die über den Slider ausgewählte Genauigkeit übergeben werden kann. Zu jedem `Comparer` gehört die `compare()`-Methode, der ein Vergleichsbild übergeben wird. In dem Fall, dass die Bilder ähnlich sind, liefert die Methode `true` zurück, ansonsten `false`.

6.3.6 Die Klasse `IdenticComparer`

Der `IdenticComparer` überprüft Bilder auf Identität. Er führt demnach einen Vergleich hinsichtlich der Ähnlichkeitsgrade 1 und 2 durch.

```
1  public boolean compare(MyImage compImage)
2  {
3          boolean erg = false;
4
5          SampleMap origMap = new SampleMap(origImage);
6          SampleMap compMap = new SampleMap(compImage);
7
8          if (threshold1 == 0)
9          {
10             if(origMap.compare(compMap) == 0)
11             {
12                 erg = true;
13             }
14 }
15
16 //...
```

Listing 6.9: Der `IdenticComparer`: Vergleich nach Grad 1.

Zunächst wird anhand der zu vergleichenden Bilder eine `SampleMap` erzeugt. Ähnlich wie die Klasse `MyImage` ist auch diese Klasse eine Hilfsklasse, die es erlaubt, die RGB-Werte für jedes Pixel schnell und effizient zu bestimmen. Wählt der Benutzer auf der Oberfläche eine Genauigkeit von 100% aus – ist die Toleranz also 0 –, wird automatisch laut Grad 1 verglichen. Dafür wird die `compare()`-Methode der `SampleMap` verwendet, die für jeweils zwei Pixel die Differenz der Farbwerte zwischen Original- und Vergleichsbild berechnet und daraus schließlich eine durchschnittliche Differenz ermittelt. Wenn diese Differenz 0 ist, sind die beiden Bilder identisch.
Wenn der Benutzer eine Genauigkeit von weniger als 100% auswählt, wird ein Vergleich mit Toleranz durchgeführt.

```
1  //...
2
3      else
4      {
5              int cntDismatches = 0;
6              for (int h = 0; h < origMap.getHeight(); h++)
7              {
8                  for (int w = 0; w < origMap.getWidth(); w++)
9                  {
10                     if (origMap.getPixel(w, h).compare(compMap.getPixel(w, h)) > threshold1)
11                     {
12                         cntDismatches++;
13                     }
14                 }
15             }
16
17             if (cntDismatches < 0.01 * origMap.getHeight() * origMap.getWidth())
18             {
19                 return true;
20             }
21
22    //...
```

Listing 6.10: Der `IdenticComparer`: Vergleich nach Grad 2.

In diesem Fall wird nun die SampleMap Pixel für Pixel durchlaufen und für jedes Pixel überprüft, ob die Differenz der Farbwerte zwischen Original- und Vergleichsbild größer ist, als die vom Benutzer definierte Toleranz. Falls ja, wird der Zähler für die *dismatches* um eins erhöht. Die Anzahl der *dismatches* muss unter 1% aller Pixel liegen, damit die Bilder als ähnlich gewertet werden.

6.3.7 Die Klasse `DCTComparer`

Der `DCTComparer` dient dazu, die Funktionalitäten des `IdenticComparers` zu erweitern. Dazu berechnet er die DCT-Koeffizienten der zu vergleichenden Bilder und vergleicht die dabei entstehenden Matrizen.

```
1  public boolean compare(MyImage compImage)
2  {
3          boolean result = false;
4          result = compareCDCT(compImage);
5          return result;
6  }
```

Listing 6.11: Der `DctComparer`: Aufbau der `compare()`-Methode

In dieser Methode wird zunächst lediglich die `compareCDCT()`-Methode aufgerufen.

```
1  private boolean compareCDCT(MyImage comp)
2  {
3          boolean erg = true;
4          double[][][] B;
5
6          if((dct == null)||(A == null))
7          {
8              dct = new MyCDCT(frequencies);
9              A = dct.convertToDCT(origImage);
10             dct.setQuant(quant);
11         }
12         B = dct.convertToDCT(comp);
13
14         //...
```

Listing 6.12: Der `DctComparer`: Erster Teil der `compareCDCT()`-Methode

Hier wird ein neues `MyCDCT`-Objekt erzeugt und die Berechnung der DCT-Koeffizienten für Original- und Vergleichsbilder durch dieses Objekt veranlasst. In der `convertToDCT()`-Methode wird daraufhin ein neuer Thread – der `CalcDCT`-Thread – erzeugt und gestartet. Dieser dient zur eigentlichen Berechnung der DCT-Koeffizienten. Mit Erzeugung dieses Threads wird zunächst die Erstellung der Kosinustabelle angestoßen. Diese wird ebenfalls in einem eigenen Thread (`CalcCosTable`) berechnet, dessen `run()`-Methode folgenden Aufbau hat:

```
1  public void run()
2  {
3      double[][] tmp = new double[N][N];
4      for (int n = 0; n < N; n++)
5      {
6          for (int k = 0; k < N; k++)
7          {
8              tmp[n][k] = Math.cos((Math.PI/(double)N)*((double)n+0.5)*(double)k);
9          }
10     }
11     cosTable = tmp;
12 }
```

Listing 6.13: Der `CalcCosTable`-Thread: Die `run()`-Methode zur Berechnung der Kosinustabelle

Hier wird der *cos*-Ausdruck, wie in Definition 5.1 beschrieben, für jede Position im Bild berechnet. Auf diese `cosTable` kann nun der `CalcDCT`-Thread zugreifen. Die `run()`-Methode dieses Threads ist wie folgt aufgebaut:

```
1  public void run()
2  {
3        tmp = new double[N][N][3];
4        for(int k1 = 0; k1 < N; k1++)
5        {
6            threads = new Thread[N];
7            for(int k2 = 0; k2 < N; k2++)
8            {
9                  threads[k2] = new CalcHorizontalDCTCoef(k1, k2);
10                 threads[k2].start();
11           }
12           for(int k2 = 0; k2 < N; k2++)
13           {
14                try
15                {
16                     threads[k2].join();
17                }
18                catch (InterruptedException ex)
19                {
20                     System.out.println("Fehler beim Berechnen der DCT-Koeffizienten: " + ex.
                           getMessage());
21                }
22           }
23       }
24
25     matrix = new double[N][N][3];
26
27     for(int k2 = 0; k2 < N; k2++)
28     {
29           threads = new Thread[N];
30           for(int k1 = 0; k1 < N; k1++)
31           {
32                threads[k1] = new CalcVerticalDCTCoef(k1, k2);
33                threads[k1].start();
34           }
35           for(int k1 = 0; k1 < N; k1++)
36           {
37                try
38                {
39                     threads[k1].join();
40                }
41                catch (InterruptedException ex)
42                {
43                       System.out.println("Fehler beim Berechnen der DCT-Koeffizienten: " +
                           ex.getMessage());
44                }
45            }
46       }
47 }
```

Listing 6.14: Der `CalcDCT`-Thread: Die `run()`-Methode

Zunächst wird das Bild horizontal durchlaufen und für jedes Element ein neuer Thread erzeugt, welcher die DCT-Koeffizienten berechnet. Als Zwischenergebnis entsteht eine Matrix, die im Anschluss vertikal durchlaufen wird. Für jedes Element werden dabei erneut die DCT-Koeffizienten berechnet. Die Berechnung der DCT-Koeffizienten wird am Beispiel der `run()`-Methode des `CalcHorizontalDCTCoef`-Threads deutlich:

```
1  public void run()
2  {
3          int rgb;
4          double r = 0;
5          double g = 0;
6          double b = 0;
7
8          for(int n = 0; n < N; n++)
9          {
10               rgb = bild.getRGB(k1, n);
11               r += ((rgb & (255 << 16)) >> 16) * cosTable[n][k2];
12               g += ((rgb & (255 << 8)) >> 8) * cosTable[n][k2];
13               b += ((rgb & (255))) * cosTable[n][k2];
14         }
15
16         tmp[k1][k2][0] = (r * 2.0 / (double)N);
17         tmp[k1][k2][1] = (g * 2.0 / (double)N);
18         tmp[k1][k2][2] = (b * 2.0 / (double)N);
19 }
```

Listing 6.15: Der `CalcHorizintalDCTCoef`-Thread: Die `run()`-Methode

Hier werden für jeden Farbkanal an der übergebenen Stelle die DCT-Koeffizienten berechnet und in `tmp` gespeichert. Diese Matrix wird dann durch `CalcVerticalDCTCoef` analog in die Ergebnismatrix umgewandelt und anschließend je nach der vom Benutzer gewählten Genauigkeit noch quantisiert. Diese Matrix kann nun im `DctComparer` verwendet werden, um den eigentlichen Vergleich durchzuführen, wie in Listing 6.12 gezeigt.

```
r = Math.abs(A[0][0][0] - B[0][0][0]);
g = Math.abs(A[0][0][1] - B[0][0][1]);
b = Math.abs(A[0][0][2] - B[0][0][2]);
if(r < threshold*10)&&(g < threshold*10)&&(b < threshold*10))
{
    for(int i = 0; i < frequencies; i++)
    {
        for(int j = 0; j < frequencies; j++)
        {
            if((i != 0)&&(j != 0))
            {
                r = Math.abs(A[i][j][0] - B[i][j][0]);
                g = Math.abs(A[i][j][1] - B[i][j][1]);
                b = Math.abs(A[i][j][2] - B[i][j][2]);
                if((r > threshold/2.0)||(g > threshold/2.0)||(b > threshold/2.0))
                {
                    erg = false;
                    break;
                }
            }
        }
        if(!erg)
        {
            break;
        }
    }
}
//...
```

Listing 6.16: Der `DctComparer`: Zweiter Teil der `compareCDCT()`-Methode

Der `DctComparer` überprüft zunächst, ob die Gleichanteile der Matrizen einander ähneln. Ist dies der Fall, werden auch die Wechselanteile auf Ähnlichkeit verglichen. Liegen die Abweichungen in einem Toleranzbereich, dann sind die farbigen Bilder einander ähnlich. Falls dieser Vergleich negativ ausfällt, wird im nächsten Schritt ermittelt, ob das Vergleichsbild ein Grautonbild des Originalbildes ist.

```
1  //...
2  else
3  {
4      diff = Math.abs((A[0][0][0] + A[0][0][1] + A[0][0][2]) / 3) - ((B[0][0][0] + B[0][0][1] +
           B[0][0][2]) / 3);
5      if (diff < threshold*10)/// ähnlicher Grauton
6      {
7          for(int i = 0; i < frequencies; i++)
8          {
9              for(int j = 0; j < frequencies; j++)
10             {
11                 if((i != 0)&&(j != 0))
12                 {
13                     diff = Math.abs(((A[i][j][0] + A[i][j][1] + A[i][j][2]) / 3) - ((B[i][j
                           ][0] + B[i][j][1] + B[i][j][2]) / 3));
14                     if(diff > threshold / 3.0)
15                     {
16                         erg = false;
17                         break;
18                     }
19                 }
20             }
21             if(!erg)
22             {
23                 break;
24             }
25         }
26     }
27     else
28     {
29         erg = false;
30     }
31 }
32
33 return erg;
34 }
```

Listing 6.17: Der `DctComparer`: Dritter Teil der `compareCDCT()`-Methode

Der einzige Unterschied hierbei ist, dass der Durchschnittswert aus den Rot-, Grün- und Blauanteilen gebildet wird. Anschließend wird analog wie oben beschrieben zunächst der Gleichanteil und dann der Wechselanteil der beiden Matrizen verglichen. Ist auch dieser Vergleich negativ, so sind die beiden Bilder sich nicht ähnlich laut DCT.

6.3.8 Die Klasse `TemplateMatchingComparer`

Mit diesem `Comparer` werden Bilder nach Ähnlichkeitsgrad 3 verglichen. Damit ist es also möglich, einen Ausschnitt in einem anderen Bild wiederzufinden. Für diesen Vergleich wird die *ImageJ* Bibliothek eingesetzt.

```
1  // Toleranzwert
2  private double threshold;
3  // Das Bild, in dem nach dem Ausschnitt gesucht werden soll
4  private FloatProcessor image;
5  // Ausschnitt, nach dem gesucht werden soll
6  private FloatProcessor template;
7  // Breite und Höhe des Bildes
8  private int imageWidth, imageHeight;
9  //Breite und Höhe des Templates
10 private int templateWidth, templateHeight;
11 // Anzahl der Elemente im Template
12 private int K;
13 // ColorProcessor von Bild und Template
14 private ColorProcessor colorImage, colorTemplate;
15 // Genauigkeit an Match Stelle
16 float maxD;
17 // Mittelwert des Templates
18 float meanTemplate;
19 // Quadratwurzel aus der Varianz des Templates
20 float sigmaTemplate;
21
22
23 public TemplateMatchingComparer(MyImage origImage)
24 {
25         super(origImage);
26
27         colorTemplate = origImage.getColorProcessor();
28         template = colorTemplate.getBrightness();
29         templateWidth = template.getWidth();
30         templateHeight = template.getHeight();
31         K = templateWidth * templateHeight;
32
33         // Mittelwert und Varianz des Templates berechnen
34         float sumR = 0;
35         float sumR2 = 0;
36         for (int j = 0; j < templateHeight; j++)
37         {
38                 for (int i = 0; i < templateWidth; i++)
39                 {
40                         float aR = template.getf(i,j);
41                         sumR  += aR;
42                         sumR2 += aR * aR;
43                 }
44         }
45         meanTemplate = sumR / K;
46         sigmaTemplate = (float) Math.sqrt(sumR2 - K * meanTemplate * meanTemplate);
47 }
```

Listing 6.18: Der `TemplateMatchingComparer`: Members und Konstruktor

Der Konstruktor ist hier anders aufgebaut als in den vorher beschriebenen Klassen. Hier werden zunächst sämtliche Eigenschaften des Templates – also des Originalbildes – ermittelt.

Dazu zählen auch der Mittelwert und die Varianz.

```
1  public boolean compare(MyImage compImage)
2  {
3          boolean result = false;
4          colorImage = compImage.getColorProcessor();
5
6          image = colorImage.getBrightness();
7          imageWidth = image.getWidth();
8          imageHeight = image.getHeight();
9
10         result = findTemplate();
11
12         return result;
13 }
```

Listing 6.19: Der `TemplateMatchingComparer`: Aufbau der `compare()`-Methode

In der `compare()`-Methode werden anschließend analog die Eigenschaften des Bildes, mit dem verglichen werden soll, ermittelt und die `findTemplate()`-Methode aufgerufen.

```
1  public boolean findTemplate()
2  {
3              boolean erg = false;
4              if((imageWidth-templateWidth+1 > 0)&&(imageHeight-templateHeight+1 > 0))
5              {
6                  maxD = 0;
7                  for (int r=0; r<=imageWidth-templateWidth; r++)
8                  {
9                          for (int s=0; s<=imageHeight-templateHeight; s++)
10                         {
11                                 float d = getMatchValue(r,s);
12                                 if(d > maxD)
13                                 {
14                                     maxD = d;
15                                 }
16                         }
17                 }
18                 if(maxD > threshold)
19                 {
20                     erg = true;
21                 }
22             }
23             return erg;
24 }
```

Listing 6.20: Der `TemplateMatchingComparer`: Aufbau der `findTemplate()`-Methode

Dabei wird an jeder Stelle die `getMatchValue()`-Methode aufgerufen.

```
1  private float getMatchValue(int r, int s)
2  {
3              float sumI = 0, sumI2 = 0, sumIR = 0;
4
5              for (int j=0; j<templateHeight; j++){
6                      for (int i=0; i<templateWidth; i++){
7                              float aI = image.getf(r+i,s+j);
8                              float aR = template.getf(i,j);
9                              sumI  += aI;
10                             sumI2 += aI * aI;
11                             sumIR += aI * aR;
12                     }
13             }
14             float meanI = sumI / K;
15             return (sumIR - K * meanI * meanTemplate) /
16                        ((float)Math.sqrt(sumI2 - K * meanI * meanI) * sigmaTemplate);
17 }
```

Listing 6.21: Der `TemplateMatchingComparer`: Aufbau der `getMatchValue()`-Methode

Hier wird die in Definition 5.2 vorgestellte Formel programmiertechnisch umgesetzt und damit der Korrelationskoeffizient berechnet. Der größte auf diese Art ermittelte Wert wird in `maxD` gespeichert und mit dem Toleranzwert verglichen. Liegt `maxD` über dem Schwellwert, so wurde das Template im Bild wiedergefunden. Ansonsten ist der Ausschnitt nicht in dem Bild enthalten oder das Bild weicht zu stark von dem Ausschnitt ab, z.B. durch Kompression oder Größenänderung.

6.3.9 Die Klasse `ColorComparer`

Der `ColorComparer` vergleicht Bilder hinsichtlich Ähnlichkeitsgrad 4 und ermittelt demnach, ob zwei Bilder ähnliche Farben aufweisen.

```
1  private final int CAT0 = 25;
2  private final int CAT1 = 50;
3  private final int CAT2 = 75;
4  private final int CAT3 = 100;
5  private final int CAT4 = 125;
6  private final int CAT5 = 150;
7  private final int CAT6 = 175;
8  private final int CAT7 = 200;
9  private final int CAT8 = 225;
10
11 public boolean compare(MyImage compImage)
12 {
13         boolean result = true;
14
15         SampleMap origMap = new SampleMap(origImage);
16         SampleMap compMap = new SampleMap(compImage);
17
```

```
18          int[] redOrig = analyzeColorFrequency(origMap, 0);
19          int[] greenOrig = analyzeColorFrequency(origMap, 1);
20          int[] blueOrig = analyzeColorFrequency(origMap, 2);
21
22          int[] redComp = analyzeColorFrequency(compMap, 0);
23          int[] greenComp = analyzeColorFrequency(compMap, 1);
24          int[] blueComp = analyzeColorFrequency(compMap, 2);
25
26          //...
```

Listing 6.22: Der `ColorComparer`: Erster Teil der `compare()`-Methode.

Die Pixel werden in Kategorien im Bereich von 0 bis 255 eingeteilt. Dazu werden Obergrenzen für die jeweiligen Kategorien definiert. In der `compare()`-Methode wird für beide Bilder je eine `SampleMap` angelegt. Anschließend wird jeder Farbkanal der beiden Bilder mit der `analyzeColorFrequency()`-Methode analysiert.

```
1  private int[] analyzeColorFrequency(SampleMap map, int channel)
2  {
3          int[] arr = new int[10];
4
5          for (int h = 0; h < map.getHeight(); h++)
6          {
7              for (int w = 0; w < map.getWidth(); w++)
8              {
9                    if (map.getPixel(w, h).getSamples(channel) < CAT0)
10                   {
11                         arr[0] = arr[0] + 1;
12                   }
13                   else if ((map.getPixel(w,h).getSamples(channel) >= CAT0 )&&(map.getPixel(w,
                         h).getSamples(channel) < CAT1))
14                   {
15                         arr[1] = arr[1] + 1;
16                   }
17                   else if ((map.getPixel(w,h).getSamples(channel) >= CAT1 )&&(map.getPixel(w,
                         h).getSamples(channel) < CAT2))
18                   {
19                         arr[2] = arr[2] + 1;
20                   }
21                   else if ((map.getPixel(w,h).getSamples(channel) >= CAT2 )&&(map.getPixel(w,
                         h).getSamples(channel) < CAT3))
22                   {
23                         arr[3] = arr[3] + 1;
24                   }
25                   else if ((map.getPixel(w,h).getSamples(channel) >= CAT3 )&&(map.getPixel(w,
                         h).getSamples(channel) < CAT4))
26                   {
27                         arr[4] = arr[4] + 1;
28                   }
29                   else if ((map.getPixel(w,h).getSamples(channel) >= CAT4 )&&(map.getPixel(w,
                         h).getSamples(channel) < CAT5))
30                   {
```

```
31                          arr[5] = arr[5] + 1;
32                  }
33                  else if ((map.getPixel(w,h).getSamples(channel) >= CAT5 )&&(map.getPixel(w,
                        h).getSamples(channel) < CAT6))
34                  {
35                          arr[6] = arr[6] + 1;
36                  }
37                  else if ((map.getPixel(w,h).getSamples(channel) >= CAT6 )&&(map.getPixel(w,
                        h).getSamples(channel) < CAT7))
38                  {
39                          arr[7] = arr[7] + 1;
40                  }
41                  else if ((map.getPixel(w,h).getSamples(channel) >= CAT7 )&&(map.getPixel(w,
                        h).getSamples(channel) < CAT8))
42                  {
43                          arr[8] = arr[8] + 1;
44                  }
45                  else
46                  {
47                          arr[9] = arr[9] + 1;
48                  }
49              }
50          }
51
52          return arr;
53  }
```

Listing 6.23: Der `ColorComparer`: Aufbau der `analyzeColorFrequency()`-Methode

An diese Methode wird die zu analysierende `SampleMap` sowie der gewünschte Farbkanal übergeben. Anschließend ermittelt die Methode für jedes Pixel in der `SampleMap` den Farbwert des übergebenen Farbkanals und ordnet das Pixel in eine der Kategorien ein. Die Methode liefert ein Array zurück, das für jede Kategorie angibt, wie viele Pixel sich darin befinden. Der Wert

`arr[0] = 50`

bedeutet demnach, dass sich 50 Pixel in der Kategorie 0 befinden. Nun folgt der eigentliche Vergleich.

```
1         //...
2
3         int diffRed, diffGreen, diffBlue;
4
5         for (int i = 0; i < CAT_COUNT+1; i++)
6         {
7             diffRed = Math.abs(redOrig[i] - redComp[i]);
8             diffGreen = Math.abs(greenOrig[i] - greenComp[i]);
9             diffBlue = Math.abs(blueOrig[i] - blueComp[i]);
10
11            if (diffRed >= THRESHOLD_PERCENT * origMap.getResolution())
12            {
13                result = false;
14                break;
15            }
16            if (diffGreen >= THRESHOLD_PERCENT * origMap.getResolution())
17            {
18                result = false;
19                break;
20            }
21            if (diffBlue >= THRESHOLD_PERCENT * origMap.getResolution())
22            {
23                result = false;
24                break;
25            }
26        }
27
28        return result;
29    }
```

Listing 6.24: Der `ColorComparer`: Zweiter Teil der `compare()`-Methode

Dazu wird für jede Kategorie die Differenz bestimmt, die zwischen Original- und Vergleichsbild auftritt. Nun wird überprüft, ob die ermittelte Differenz größer ist, als der vom Benutzer festgelegte Schwellwert. Ist dies der Fall, ähneln sich die Bilder nicht hinsichtlich ihrer Farben und der Vergleich wird abgebrochen. Ist die Differenz für alle Farbkanäle geringer als der Toleranzwert, so haben die Bilder ähnliche Farben.

7 Zusammenfassung und Ausblick

Die Implementierung eines Prototyps, der einen Bildvergleich anhand von Ähnlichkeitsgraden durchführt, ist abgeschlossen. In diesem Kapitel werden die dabei gewonnenen Erkenntnisse und die Ergebnisse der Implementierung analysiert. Zudem wird ein Ausblick auf mögliche Verbesserungen und Erweiterung der Software gegeben.

7.1 Was wurde erreicht?

Im Rahmen dieser Arbeit wurden zunächst einige Grundlagen zur maschinellen Bildverarbeitung gelegt. Zudem wurden verschiedene Bildvergleichs-Programme analysiert und bewertet. Die gewonnenen Erkenntnisse dienten als Inspiration für die Entwicklung einer eigenen Software. In Folge dessen ist schließlich ein Prototyp entstanden, der nach verschiedenen Kriterien und dadurch mit unterschiedlichen Methoden die Ähnlichkeit von Bildern analysiert. Dabei hat der Benutzer einerseits die Möglichkeit, einen reinen Ähnlichkeitsvergleich zweier Bilder durchzuführen. Andererseits ist es möglich, ein Bild mit allen Bildern aus einem gewählten Verzeichnis des Dateisystems zu vergleichen und alle ähnlichen Bilder auszugeben. Der Entwurf der grafischen Oberfläche ist möglichst schlicht, um eine intuitive Steuerung durch den Benutzer zu ermöglichen. Da die Benutzbarkeit eine sehr wichtige Anforderung an den Prototyp ist, enthält die Software zusätzlich eine Hilfsfunktion, welche die wichtigsten Schritte des Bildvergleichs erläutert. Auch wenn die Effizienz gegenüber der Zuverlässigkeit und Funktionalität nicht die wichtigste Anforderung ist, wird versucht, durch den Einsatz von Threads und das Skalieren von Bildern, die Rechenzeit möglichst gering zu halten. Der Benutzer muss allerdings besonders bei größeren Bildern unter dem Einsatz von Template Matching mit größeren Bearbeitungszeiten rechnen. Der zugehörige Quellcode weist eine übersichtliche Gliederung und Dokumentation auf, um eine gute Änderbarkeit und Übertragbarkeit zu gewährleisten. Die entwickelte Software liefert im Rahmen der Tests zufriedenstellende Ergebnisse.

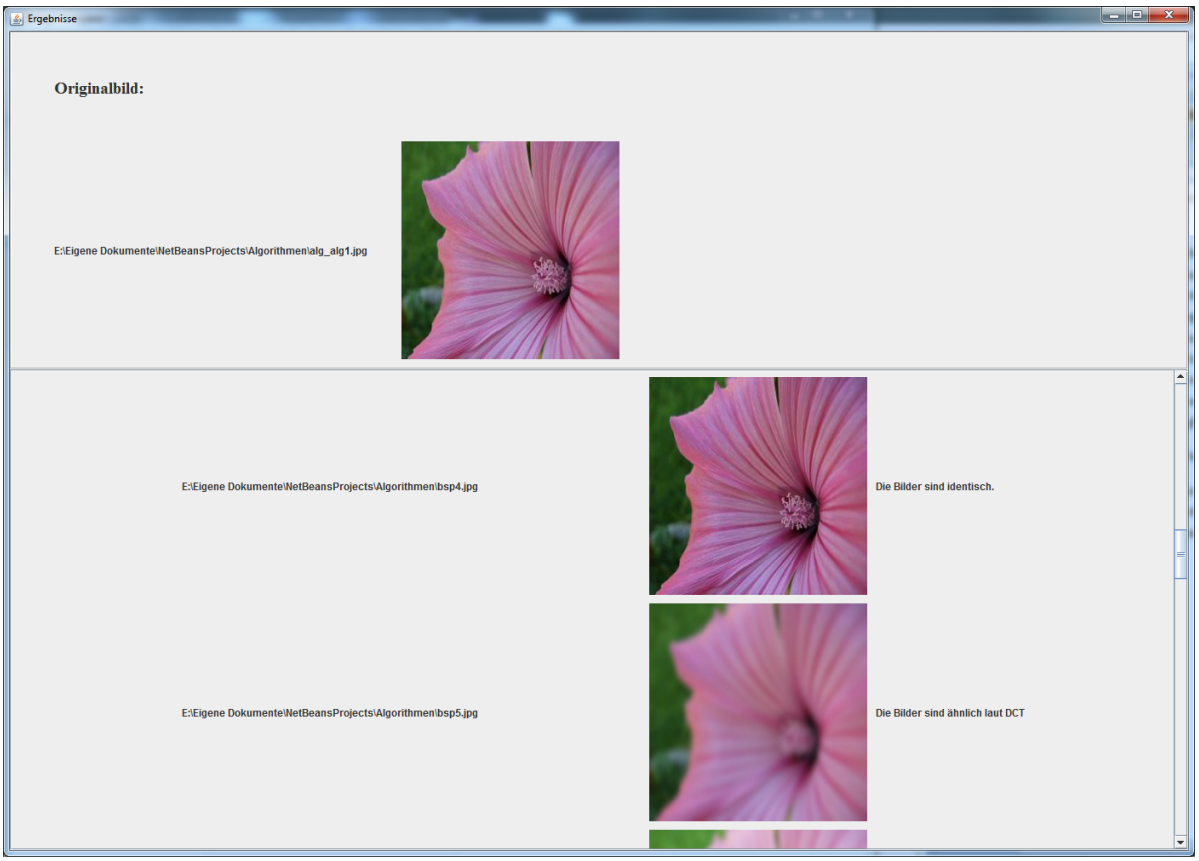


Abbildung 7.1: Auszug aus den Ergebnissen eines Vergleichs. Es wurde nach allen Vergleichskriterien verglichen. Die Genauigkeit wurde auf 80% gesetzt.

Der Benutzer hat die Möglichkeit, die Ergebnisse zu beeinflussen, indem er Vergleichskriterien und eine Genauigkeit angibt. Dabei ist festzustellen, dass die besten Ergebnisse bei einer Genauigkeit von 80% geliefert werden. Bei einer Genauigkeit unter 50% steigt die Fehlerrate, je nach gewählten Vergleichskriterien, stark.

7.2 Mögliche Verbesserungen und Erweiterungen

Es ist denkbar, noch weitere Algorithmen zu implementieren, um die Breite der Vergleichskriterien zu erweitern. So ist z.B. die Möglichkeit eines Vergleichs von Bildern, die ähnliche Strukturen, Motive und Formen aufweisen, eine mögliche Erweiterung. Um die Effizienz des Programms zu steigern, könnte eine Datenbank zum Einsatz kommen. Dies ist besonders für den DCT-Algorithmus von Vorteil. In diesem Fall könnten als eine Art digitaler Fingerabdruck zu jedem Bild lediglich die DCT-Koeffizienten in der Datenbank gespeichert werden. Während des Vergleichs müssten dann nur die Koeffizienten verglichen werden, wodurch sich die Rechenzeit stark verkürzen würde. Anstatt eine Datenbank zu verwenden, könnten dabei

auch sequentielle Textdateien zum Einsatz kommen, in denen die DCT-Koeffizienten für jedes Bild des Dateisystems gespeichert werden. Dies würde erheblich dazu beitragen, die Rechenzeit zu verkürzen. Des Weiteren besteht die Möglichkeit, die Ergebnisseite zu erweitern. Ein derartiges Bildvergleichsprogramm wird oftmals eingesetzt, um doppelte oder ähnliche Bilder zu löschen. Daher könnte auf der Ergebnisseite z.B. eine Funktion hinzugefügt werden, mit der Bilder sofort gelöscht werden können.

Literatur

[Ame11] Hitachi America. *GazoPa - similar image search*. http://www.gazopa.com. [Online; letzter Zugriff: 6. Juni 2011]. 2011.

[Aus02] Dr. Hubert Austermeier. *Bildverarbeitung mit Java*. http://www.ordix.de/ORDIXNews/3_2002/java_1.html. [Online; letzter Zugriff: 06. Juli 2011]. 2002.

[BB06] Wilhelm Burger und Mark James Burge. *Digitale Bildverarbeitung mit Java und ImageJ*. Berlin Heidelberg: Springer Verlag, 2006.

[Bis07] Christopher M. Bishop. *Pattern Recognition And Machine Learning*. New York: Springer Verlag, 2007.

[Inc11] Idée Inc. *TinEye Reverse Image Search Engine*. http://www.tineye.com. [Online; letzter Zugriff: 6. Juni 2011]. 2011.

[Jäh02] Prof. Dr. Bernd Jähne. *Digitale Bildverarbeitung*. Berlin Heidelberg: Springer Verlag, 2002.

[KKS96] Reinhard Klette, Andreas Koschan und Karsten Schlüns. *Computer Vision*. Braunschweig/Wiesbaden: Vieweg Verlag, 1996.

[Ros10] Jörg Rosenthal. *Anti-Twin Freeware: Doppelte Dateien finden und löschen*. http://www.aidex.de/software/antitwin. [Online; letzter Zugriff: 6. Juni 2011]. 2010.

[Sch10] Edgar Scherstjanoi. *Vergleich und Übersicht von Algorithmen zur Dokumentenduplikaterkennung bei Bildern*. http://www.rn.inf.tu-dresden.de/uploads/Studentische_Arbeiten/Belegarbeit_Scherstjanoi_Edgar.pdf. [Online; letzter Zugriff: 6. Juni 2011]. Okt. 2010.

[Sof11] Quality First Software. *Details des Algorithmus zum Bildvergleich*. http://www.qfs.de/qftest/manual/de/tech_imagealgorithmdetails.html. [Online; letzter Zugriff: 17. August 2011]. Mai 2011.

[Str05] Tilo Strutz. *Bilddatenkompression*. Wiesbaden: Vieweg Verlag, 2005.

[Wal11] Dr. Axel Walthelm. *PictureRelate - die etwas andere Art Bilder zu betrachten*. http://www.walthelm.net/picture-relate/de/index.php. [Online; letzter Zugriff: 6. Juni 2011]. 2011.

[Wik11] Wikipedia. *Diskrete Kosinustransformation*. http://de.wikipedia.org/wiki/Diskrete_Kosinustransformation. [Online; letzter Zugriff: 17. August 2011]. 2011.

Verwendete Tools

- NetBeans IDE 6.9.1
- Texmaker 2.1
- Dia 0.97.1

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Stellen, die wörtlich oder sinngemäß aus Quellen entnommen wurden, sind als solche gekennzeichnet. Alle Bilder wurden, wenn nicht anders angegeben, selbst erstellt.

Diese Arbeit liegt in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vor.

Mittweida, den 16. September 2011

.....
Nicole Möstel