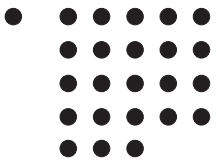


Fachhochschule Köln
University of Applied Sciences Cologne

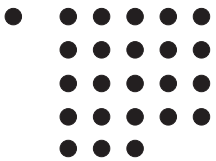
07 Fakultät für Informations-, Medien-, und Elektrotechnik
Studiengang Photoingenieurwesen und Medientechnik

Diplomarbeit

Implementierung einer Benutzerschnittstelle zur medizinischen Evaluierung eines Image Mosaicing-Verfahrens in der Endoskopie

Christian Zimmermann

Köln im Juli 2008



Fachhochschule Köln
University of Applied Sciences Cologne

Department of Imaging Sciences and Media Technology

Thesis

Subject: Implemention of a grafical user interface for medical evaluation of an image mosaicing process.

Author:	Christian Zimmermann
Matr.-Nr:	11043102

Referent:	Prof. Dr.-Ing. D. Kunz
Korreferent:	Prof. Dr.-Ing. W. Konen

Abgabedatum: 31. Juli 2008

Hiermit versichere ich, dass ich die Diplomarbeit selbständig angefertigt und keine anderen als die angegebenen und bei Zitaten kenntlich gemachten Quellen und Hilfsmittel benutzt habe.

Christian Zimmermann

Zusammenfassung

Im Rahmen dieser Arbeit wurde ein bestehendes Programm zum Image Mosaicing für die medizinische Endoskopie, d.h. zur Konstruktion eines Übersichtsbildes aus einer Folge von endoskopischen Einzelaufnahmen, weiterentwickelt. Eine neu erstellte Benutzerschnittstelle und die erweiterten Möglichkeiten des Datenflusses erlauben einen flexiblen Zugriff sowohl auf gespeicherte Aufnahmen als auch auf eine angeschlossene Kamera. Die Implementierung erfolgte betriebssystemunabhängig in Java unter Verwendung der Programmierschnittstelle Java Media Framework (JMF). Die erstellten Programmerweiterungen erlauben die systematische Evaluierung des Verfahrens auf vorhandenen Datensätzen und stellen einen wesentlichen Schritt in Richtung einer klinischen Erprobung dar.

Abstract

An image mosaicing software for endoscopic images, i.e. a software generating panoramic views from a sequence of images, has been refined. A new user interface and improved data flow capabilities allow a flexible access to both saved streams and a connected camera. The program has been implemented in Java to be platform independent and uses the Java Media Framework (JMF) API. The created program extensions allow a systematic evaluation of the mosaicing process on recorded data. That means a major step forward towards clinical tests.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Durch Image Mosaicing in der Medizin und hier speziell in der Endoskopie ist es möglich, dem operierenden Mediziner neben der klassischen Darstellung des Kamerabildes der Endoskopkamera, zusätzlich ein Übersichtsbild zu liefern, welches ihm eine bessere Orientierung verschafft und ihm somit die Arbeit erleichtert.
Ziel dieser Arbeit ist die Implementierung einer grafischen Benutzerschnittstelle die es gestattet, unterschiedliche Videoformate und Videoquellen einzulesen und dem bestehenden Image Mosaicing Plugin zu übergeben, um so eine medizinische Evaluierung der Software zu ermöglichen.

1.1 Motivation

Das hier vorliegende PlugIn, welches basierend auf der Grundlage eines von Kourogi[5] vorgestellten Algorithmus entwickelt wurde, ist ein erster Schritt hin zur einer echtzeitfähigen Image Mosaicing Software.
In der weiteren Entwicklung werden sich hier noch eine Reihe weite-

rer Arbeiten anschließen (müssen), deren Ziel es sein wird, den Prozess des Image Mosaicings zu optimieren und so einen realen Einsatz zu ermöglichen.
Um alle diese Änderungen und Verbesserungen in der nahen Zukunft genauer evaluieren zu können, beziehungsweise erste Testversuche unter Realbedingungen durchführen zu können, wäre es deshalb wünschenswert eine grafische Benutzerschnittstelle zur Verfügung zu haben, mit der es möglich ist reproduzierbare und aussagekräftige Tests durchzuführen. Dabei sollte eine möglichst einfache und strukturierte Bedienung gegeben und die Möglichkeiten der Ankopplung von Mediencontent maximal sein.

1.2 Zielsetzung

Wie bereits erwähnt, ist es das Ziel dieser Arbeit eine Benutzerschnittstelle zur Verfügung zu stellen, die es ermöglicht, ein großes Spektrum an Medienformaten zu verarbeiten und mit Hilfe einer grafischen Benutzeroberfläche eine medizinische Evaluierung von Bildmaterial der Neuroendoskopie zu ermöglichen.

1.3 Aufbau der Arbeit

Um den Aufbau der vorliegenden Arbeit kurz wiederzugeben und dem Leser somit einen ersten Überblick zu verschaffen, soll im Folgenden kurz auf die Inhalte der einzelnen Kapitel eingegangen werden.
Im 2. Kapitel werden zunächst notwendige Grundlagen für diese Arbeit behandelt. Diese beinhalten sowohl die verschiedenen Möglichkeiten der Medienverarbeitung in Java, als auch die mathematischen Grundlagen des Image Mosaicings. Auf die implementierte Benutzerschnittstelle wird in

Kapitel 3 konkret eingegangen. Nach einer Anleitung zur Benutzung der einzelnen Hauptfunktionen werden die Methoden detailliert beschrieben. Ein erster Funktionstest der Benutzerschnittstelle und ein Fazit runden diese Arbeit ab.
Im Anhang finden sich neben allen Quellcodes auch Installationsbeschreibungen zur benötigten Software.

Kapitel 2

Grundlagen

2.1 Medienverarbeitung in Java

Für die Verarbeitung von Medien (vornehmlich Bild und Ton) stehen eine Reihe von Erweiterungen für Java zur Verfügung. Es werden in diesem Abschnitt die beiden bedeutendsten Vertreter[1] vorgestellt.

2.1.1 QuickTime for Java

Mit dem von Apple entwickelten und 1991 veröffentlichten QuickTime for Java (QTJava) existiert ein bedeutendes Framework für die Verarbeitung von Medien in Java. So existieren in QTJava neben Methoden zur Verarbeitung von Bild und Ton auch einfache Methoden zur Bildverarbeitung, Animation und 3D-Modellierung. Als Anwendungsbeispiele wären hier Video-Overlays und Filterungen zu nennen[2].
Hier wird also schon ein wesentlicher Vorteil, nämlich der große Funktions-

[1] Neben den hier vorgestellten gibt es eine Reihe weiterer Frameworks. Dazu zählen noch OpenML und DirectX/DirectShow.[4]

[2] siehe [1]

umfang von QTJava deutlich. Nachteilig zu bewerten, sind allerdings die Tatsachen dass es faktisch nicht plattformunabhängig ist, da es nur auf Windows- und Mac-Systemen lauffähig ist und keine Netzwerkprotokolle unterstützt werden. Außerdem unterstützt es nicht alle gängigen Dateiformate, wobei wiederum zu erwähnen ist, dass moderne und mittlerweile schon sehr gängige Formate wie MP3 und MPGEG-4 im Gegensatz zum Java Media Framework sehr wohl unterstützt werden.

2.1.2 Java Media Framework

Die Firma Sun Microsystems[3] liefert mit dem Java Media Framework (JMF) ein Medienverarbeitungspaket welches es erlaubt, Medien in Java völlig plattformunabhängig zu verarbeiten.
War es vorher nur möglich Audio- und Videodaten abzuspielen, so ist es seit der 1999 veröffentlichten Version 2.0 ebenfalls möglich, eine Vielzahl von Formaten nicht nur zu öffnen, sondern auch zu schreiben. Außerdem gibt es Methoden zum Capturing von Daten und zum Senden und Empfangen von Daten via RTP[4].
Wie im vorherigen Abschnitt bereits angedeutet, besitzt auch das JMF einige Schwachstellen gegenüber seinen Konkurrenzprodukten. So werden weder MPEG-4 noch MP3, letzteres aufgrund von Lizenzschwierigkeiten, unterstützt. Grundsätzlich sind Videofiles in den Formaten MPEG-1, sowie den Containerformaten AVI und MOV abspielbar. Genauere Informationen, auch zu gängigen Codecs der einzelnen unterstützten Formate, können Abbildung 2.1 entnommen werden.

[3]Ein in den USA ansässiger Softwarekonzern.

[4]Realtime Transport Protocol: Protokoll zur kontinuierlichen Übertragung von audiovisuellen Daten über IP-basierte Netzwerke

AVI (.avi)	**read/write**	**read/write**	**read/write**
Audio: 8-bit mono/stereo linear	D,E	D,E	D,E
Audio: 16-bit mono/stereo linear	D,E	D,E	D,E
Audio: DVI ADPCM compressed	D,E	D,E	D,E
Audio: G.711 (U-law)	D,E	D,E	D,E
Audio: A-law	D	D	D
Audio: GSM mono	D,E	D,E	D,E
Audio: ACM**	-	-	D,E
Video: Cinepak	D	D,E	D
Video: MJPEG (422)	D	D,E	D,E
Video: RGB	D,E	D,E	D,E
Video: YUV	D,E	D,E	D,E
Video: VCM**	-	-	D,E
MPEG-1 Video (.mpg)	-	**read only**	**read only**
Multiplexed System stream	-	D	D
Video-only stream	-	D	D
QuickTime (.mov)	**read/write**	**read/write**	**read/write**
Audio: 8 bits mono/stereo linear	D,E	D,E	D,E
Audio: 16 bits mono/stereo linear	D,E	D,E	D,E
Audio: G.711 (U-law)	D,E	D,E	D,E
Audio: A-law	D	D	D
Audio: GSM mono	D,E	D,E	D,E
Audio: IMA4 ADPCM	D,E	D,E	D,E
Video: Cinepak	D	D,E	D
Video: H.261	-	D	D
Video: H.263	D	D,E	D,E
Video: JPEG (420, 422, 444)	D	D,E	D,E
Video: RGB	D,E	D,E	D,E

Abbildung 2.1: Von JMF unterstützte Videoformate (ohne Audioformate). Quelle:[7]

write indicates the media type can be generated as output
read indicates the media type can be generated as input
D indicates the format can be decoded and presented
E indicates the format can be encoded in the format

2.2 Image Mosaicing

Für ein umfassendes Verständnis des vorliegenden ImageJ PlugIns ist es notwendig, auch die grundlegende mathematische Vorgehensweise beim Image Mosaicing, also beim Matching und Zusammensetzen von Einzelbildern einer Sequenz zu einem Übersichtsbild, zu verstehen.
Das hier verwendete Prinzip[9] des Image Mosaicing hat den verbesserten Algorithmus von Kourogi[5] zur Grundlage. Dieser schätzt das Bewegungsvektorfeld zwischen zwei aufeinander folgenden Frames einer Videosequenz. Grundlage ist die Gleichung des Optischen Flusses:

$$I(x+u, y+v, t) - I(x, y, t-1) = 0 \tag{2.1}$$

Da in der vorangegangenen Formel aber die beiden Unbekannten u und v nicht ohne weiteres bestimmt werden können, führt Kourogi hier den sogenannten Pseudo Motion Vektor ein. Um diesen zu berechnen wird für u der v-Term zu Null gesetzt und entsprechend für v verfahren. Die beiden Vektoren errechnen sich dann aus:

$$u_p = -I_t / I_x \tag{2.2}$$

und

$$v_p = -I_t / I_y \tag{2.3}$$

Die partiellen Ableitungen der Lichtstärke nach der x- und y-Richtung sowie der Zeit, werden folgendermaßen berechnet:

$$I_t = I(x, y, t) - I(x, y, t-1) \tag{2.4}$$

$$I_x = (I(x+1, y, t-1) - I(x-1, y, t-1))/2 \tag{2.5}$$

$$I_y = (I(x, y+1, t-1) - I(x, y-1, t-1))/2 \tag{2.6}$$

Um hier Ausreißer abzufangen welche das Ergebnis verfälschen könnten, wird eine Grauwertschwelle angelegt und in folgender Gleichung getestet:

$$|I(x + u_p, y + y_p, t) - I(x, y, t-1)| < T \tag{2.7}$$

Kourogi wählt für diese Schwelle den Wert 5[5]. Dieser erwies sich als guter Kompromiss zwischen der Lauffähigkeit in Echtzeit und einer soliden Schätzung. Alle Vektoren die diesen Test bestehen, können nun für die weitere Verarbeitung verwendet werden.
Da mit Hilfe dieser Vektoren aber nur sehr geringe Verschiebungen geschätzt werden können, wird von Kourogi eine weitere Schätzung eingeführt. Diese, als Compensated Motion bezeichnete Anwendung, beruht auf der Annahme, dass sich das entsprechende Verschiebungsfeld aus affinen Transformationen zusammensetzt. Aus der Pseudo Motion wird nun also eine globale Compensated Motion geschätzt. Diese lässt sich nun wie folgt darstellen:

$$\begin{pmatrix} u_c \\ v_c \end{pmatrix} = \begin{pmatrix} a_x + a_2 y + a_3 \\ a_4 x + a_5 y + a_6 \end{pmatrix} \tag{2.8}$$

[5]In dem von Naderi implementierten PlugIn[8] ist dieser Wert in einem Bereich von 3 bis 7 einstellbar.

Abschließend müssten nun noch die Gleichung für die partielle Ableitung nach der Zeit

$$I_t^{(c)} = I(x + u_c, y + v_c, t) - I(x, y, t - 1) \tag{2.9}$$

und die Gleichungen zur Berechnung der Pseudo Motion

$$u_p = (-I_t^{(c)}/I_x) + u_c \tag{2.10}$$

und

$$v_p = (-I_t^{(c)}/I_y) + v_c \tag{2.11}$$

geändert werden.
Konkret sieht der Ablauf der Berechnung im Falle des Image Mosaicing PlugIns nun wie folgt aus:

- Für jedes Pixel der Endoskopmaske wird zuerst die Pseudo Motion berechnet.
- Es werden nur die Pixel akzeptiert und zur weiteren Berechnung herangezogen, welche die folgenden Kriterien erfüllen.
  I_x und I_y sind ungleich 0.
  $(x + u_p, y + v_p)$ befindet sich innerhalb der Endoskopmaske.
  $|I(x + u_p, y + y_p, t) - I(x, y, t - 1)| < T$, wobei T der bereits angesprochene Schwellenwert (z.B. 5) ist.
- Bestimmung der affinen Parameter $a = \{a_1, ..., a_6\}$ für das globale Bewegungvektorfeld mit Hilfe des folgenden Gleichungssystems:

$$a_1 x_i + a_2 y_i + a_3 = u_{p,i}$$
$$a_4 x_i + a_5 y_i + a_6 = u_{p,i}$$

Das daraus erhaltende Bewegungsvektorfeld wird als neue Schätzung für (u_c, v_c) angenommen. Entsprechend der Anzahl der eingestellten Iterationen werden diese Bearbeitungsschritte nun wiederholt.

Dies geschieht solange bis entweder die Anzahl der zuvor eingestellten Iterationen erreicht ist, oder die Veränderung im globalen Bewegungsvektorfeld unter einen bestimmten Schwellenwert fällt.

2.3 Allgemeine Software

Im Folgenden wird auf die, für die Inbetriebnahme des Image Mosaicing PlugIns notwendige Software eingegangen. Da das Ergebnis dieser Arbeit vor allem dafür genutzt werden soll, das Programm im Einsatz bzw. mit realen Endoskopiebildern zu testen und hier eine Bedienung beziehungsweise auch Einrichtung der Software[6] durch einen Mediziner erfolgt, soll recht detailliert auf die benötigte Software eingegangen werden, um die weitere Arbeit möglichst einfach und reibungslos anknüpfen zu können. Zu erwähnen ist noch, dass alle folgenden Softwarepakete public domain und somit frei verfügbar sind.

2.3.1 Java

Java ist eine von Sun entwickelte, objektorientierte Programmiersprache, welche 1996 der Öffentlichkeit vorgestellt wurde und innerhalb kurzer Zeit begeisterten Zuspruch fand. Die Vorteile von Java liegen in den ebenso

[6] siehe Anhang

vielfältigen Möglichkeiten wie sie beispielsweise C oder C++ aufweisen, umgehen dabei allerdings fehleranfällige Funktionen wie Mehrfachvererbung und Zeiger.
Aktuell liegt die Java Platform Standard Edition in der Version 6 vor.

2.3.2 ImageJ

ImageJ ist ein leistungsfähiges Bildbearbeitungs- und Bildverarbeitungsprogramm, welches von Wayne Rasband am U.S. Nation Institute of Health (NIH) entwickelt wurde. Es wurde in Java geschrieben und ist somit vollkommen plattformunabhängig. Durch sein PlugIn-Konzept, welches auch für dieses Projekt genutzt wurde, lässt es sich sehr einfach um eigene Komponenten erweitern[3].

2.3.3 Java Media Framework

Wie weiter oben bereits erwähnt, wird das Java Media Framework (JMF) von der Firma Sun bereitgestellt und erweitert die Standard Edition von Java um eine Vielzahl von Methoden zur Be- und Verarbeitung von Medien. Das JMF liegt aktuell in der Version 2.1.1e vor.

2.3.4 Panasonic VFW DV-Codec

Um die Funktionalität und Kompatibilität des PlugIns um eine nicht unerhebliche Funktion zu erweitern, wird zusätzlich zu denen von JMF unterstützten Videocodecs ein DV Codec benötigt.
Zur medizinischen Auswertung von Endoskopmaterial wird dieses während einer Operation aufgezeichnet, DV-codiert und als AVI gespeichert. Vorgabe der Medizin ist es nun, ein Image Mosaicing dieses Materials aus medizinischer Sicht zu evaluieren um entsprechende Schwachstellen

aufdecken und verbessern zu können.
Der DV Codec von Panasonic arbeitet hier sehr gut im Zusammenspiel mit Java, JMF und der VfW-Schnittstelle[7] und ist zudem frei verfügbar.

[7]Video for Windows (VfW) ist eine Programmierschnittstelle von Microsoft Windows. Sie ist die Standard-Schnittstelle für das AVI-Containerformat und erlaubt es Videosignale von Aufnahmegeräten einzulesen.

Kapitel 3

Das PlugIn

Im folgenden Abschnitt wird konkret auf das PlugIn und besonders auf die veränderten Klassen des PlugIns eingegangen. Es wird gezeigt welche Veränderungen notwendig waren, um einem Benutzer die Möglichkeit zu geben verschiedenste Medienformate, sowohl direkt von angeschlossenen Kameras (Online) als auch aus Files (Offline) zu importieren, bildgenaue Auswahlen zu treffen und diese, nach einer Reihe von weiteren spezifischen Optionen, einem Image Mosaicing zu unterziehen.
Es wird auf die Funktionalität der Benutzeroberfläche eingegangen und deren Bedienmöglichkeiten in den unterschiedlichen Modi beschrieben. Anschließend wird die, in dieser Version noch provisorische, Bereitstellung der Bildmasken beschrieben. Eine detaillierte Beschreibung der Methoden rundet dieses Kapitel ab.

3.1 Die veränderten Klassen des PlugIns

Ausgehend vom ImageJ PlugIn von M Naderi[8], dessen Klassen nach dem MVC Prinzip[1] unterteilt wurden, mussten für diese Arbeit ausschließlich bestehende Klassen der Ebenen Präsentation(view) und Steuerung(controller) verändert werden.
Konkret handelt es sich dabei zum einen um die Klasse $GUI.java$. Diese musste komplett neu strukturiert werden. War die bisherigere Version der Benutzeroberfläche nur für eine allgemeine Demonstration des Mosaicing Algorithmus konzipiert, in dem eine Reihe festgelegter Bildsequenzen verarbeitet werden konnten, so wurde dieser Version nun eine weit größere Bandbreite an Möglichkeiten zur Benutzerinteraktionen hinzugefügt. Diese reichen vom auslesen und Speichern einzelner (oder mehrerer) Frames aus einer Videodatei (im folgenden als Grabbing bezeichnet) über die Auswahl und Verarbeitung beliebiger im Speicher abgelegter Bildsequenzen, bis hin zur direkten Übergabe von Kamerabildern an den Image Mosaicing Algorithmus.
Desweiteren wurde die Klasse $Control\_.java$ einiger Veränderungen unterzogen. Wurden die Bilder einer gewählten Sequenz bisher zuerst in einen ImageStack geladen, der dann Bild für Bild ausgelesen wurde, so werden sie nun direkt von der Festplatte gelesen und der Bearbeitung zugefügt.
Die besonderen Anforderungen einer direkten Übergabe von Kamerabildern an den Mosaicing Prozess erforderten eine zusätzliche Methode *onlineprocess*, in welcher, simultan zur Methode *process* für die Offlineübergabe, die vom Benutzer übergebenen Parameter gesammelt und alle

[1] Model View Controller(MVC) bezeichnet die Strukturierung von Software in die drei Einheiten Datenmodell, Präsentation und Programmsteuerung. Mit einem solchen genormten Entwurf werden Übersichtlichkeit, Variabilität und eine spätere Erweiterung des Programms erleichtert.

weiteren Verarbeitungsschritte gesteuert werden.
So wird in dieser Methode ein nebenläufiger Thread gestartet, welcher regelmäßig Frames von einer angeschlossenen (Endoskop-)Kamera grabbt und diese übergibt.
Eine Veränderung der bestehenden Klassen der Einheit Datenmodell(model) war nicht nötig. Ihr wurden lediglich die Klassen *captureFromDevice* und $frameGrab$ hinzugefügt.
Erstere grabbt regelmäßig Frames von der angeschlossenen Kamera und wird, wie bereits angesprochen, aus *onlineprocess* als nebenläufiger Thread gestartet.
$frameGrab$ hingegen, grabbt die vom Benutzer festgelegten Frames eines Streams und legt sie in einem Verzeichnis ab.

3.2 Funktionalität und Bedienmöglichkeiten der grafischen Oberfläche

Bei der Konzeption und Realisierung der grafischen Benutzeroberfläche wurde versucht, eine modulare Struktur zu verwirklichen welche es dem Anwender ermöglicht, sich seinen Arbeitsplatz nach Belieben anzuordnen. Es sollte möglich sein, einzelne Fenster ein- oder auszublenden und so nebeneinander anzuordnen, dass beispielsweise eine gleichzeitige Betrachtung von Kamerabild und Panoramabild möglich ist.
Die Vorteile großer Panels oder Multi-Monitoring-Lösungen können so perfekt ausgenutzt werden und verschaffen dem Benutzer einen sehr komfortablen und übersichtlichen Arbeitsplatz.
Um dies zu realisieren wurden einem DesktopFrame mehrere ChildFrames untergeordnet. Dies sind einerseits ControlFrame und SelectFrame, welche der Steuerung des Programms dienen, indem sie Eingaben des

Benutzers entgegennehmen und zur weiteren Verarbeitung an das Programm übergeben. Auf der anderen Seite PanoramaFrame, JMFrame und Results, die wiederum der Ausgabe von Videofiles, Kamerabildern, Panoramasequenzen und Messergebnissen dienen.
Um die Ergonomie der Benutzeroberfläche den Anforderungen an die tägliche Bedienung anzupassen, fand während der Entwicklung der grafischen Benutzeroberfläche ein Treffen mit dem bedienenden Personal statt, in welchem konkrete Verbesserungsvorschläge gesammelt und im weiteren Verlauf dieser Arbeit entsprechend umgesetzt wurden.
Die nachfolgende Abbildung (3.1) zeigt die grafische Bedienoberfläche. Mit Verweis auf diese, wird anschließend detailliert auf die einzelnen Abläufe bei der Bedienung des PlugIns in den verschiedenen Modi eingegangen. Somit werden alle relevanten Funktionen erklärt.

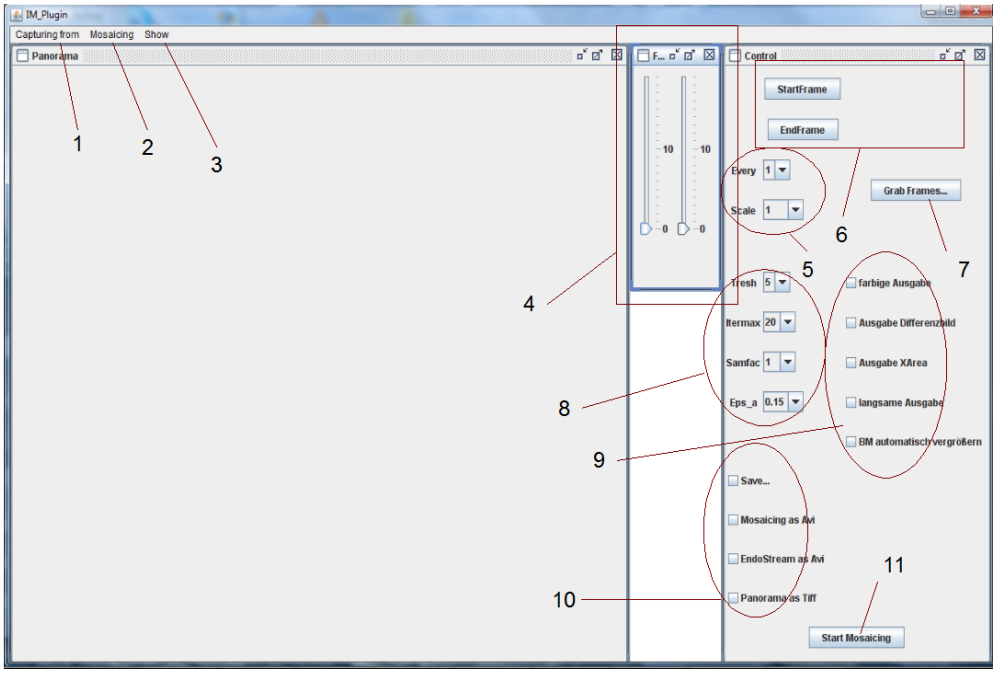


Abbildung 3.1: Die grafische Benutzeroberfläche des PlugIns

3.2.1 Framegrabbing

Eine Funktion, die vor allem in der aktuellen Entwicklungsphase des PlugIns von großer Bedeutung ist, stellt die Möglichkeit des bildweisen Grabbings aus vorhandenem Videomaterial dar. Die Möglichkeit vorhandenes Bildmaterial aus vergangenen Endoskopoperationen nutzen zu können um Mosaicingbilder aus medizinischer Sicht evaluieren zu können, stellt einen entscheidenden Teil dieser Arbeit dar. Die nachfolgende Beschreibung lässt sich auch Anhand des entsprechenden Flussdiagramms in Abbildung 3.2 nachvollziehen. Über den Menüzeileneintrag $Capture from(1^2)$ und den Eintrag $File$, gelangt man zu einem gewöhnlichen Auswahldialog.

[2] Alle bis Abschnitt 3.3 folgenden Zahlen in Klammern, beziehen sich auf Abbildung 3.1.

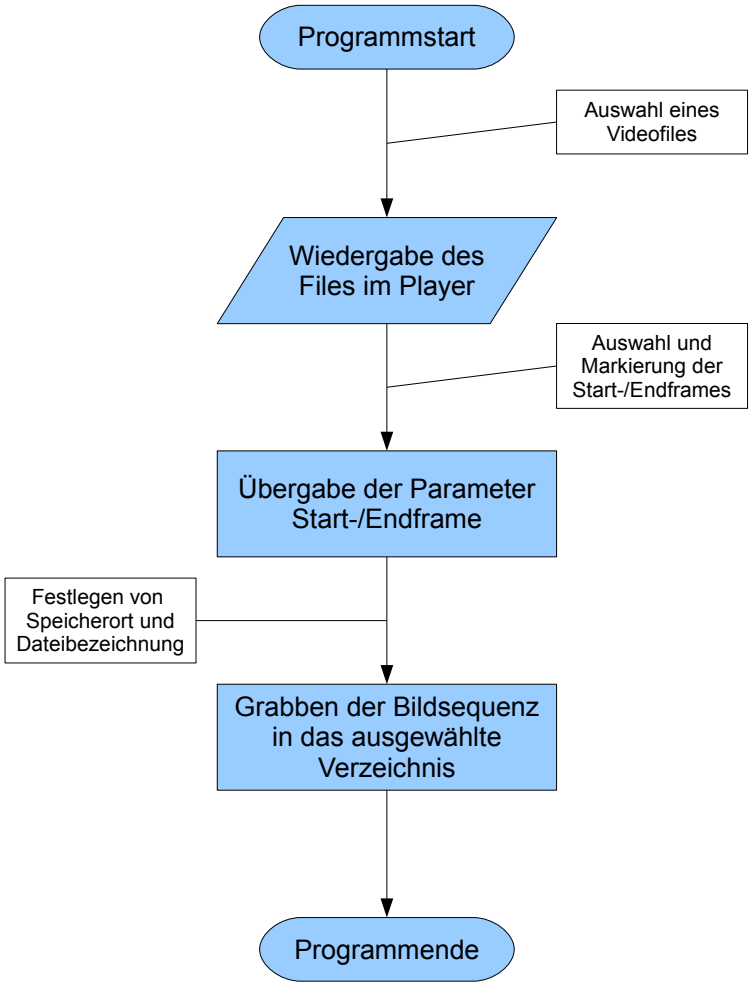


Abbildung 3.2: Flussdiagramm zum Bedienungsablauf beim Framegrabbing

Hier lässt sich ein beliebiger[3] im Speicher abgelegter Videofile auswählen. Nach Bestätigung wird der entsprechende Stream in einem Standardplayer von JMF auf dem DesktopFrame geöffnet und abgespielt.
Dieser Player unterstützt die allgemeinen Funktionen eines Medienplayers, d.h. die Funktion Play, Pause, Skip (vorwärts/rückwärts) sowie einen Fader, mit dem man direkt an eine bestimmte Stelle im Stream springen kann.
Mit Hilfe der Buttons $Startframe$ und $Endframe$(6), welche auf dem ControlFrame angeordnet sind, lässt sich nun der interessierende Bereich der Sequenz und damit auch der Bereich aus dem anschließend eine Bildfolgesequenz erstellt werden soll, eingrenzen. Bei Betätigung der Buttons wird jeweils rechts davon das geloggte Frame angezeigt.
Nun hat man mit den Auswahlboxen $Every$ und $Scale$(5) zum einen die Möglichkeit nur jedes n-te Frame (every first, second, third,...) des interessierenden Bereichs zu grabben. Dies ist vor allem bei relativ statischen Sequenzen sinnvoll. Man deckt somit einen größeren Zeitbereich ab, ohne unnötig viel Speicher für ohnehin fast ausschließlich redundante Informationen zu verschwenden. $Scale$ bietet die Möglichkeit die Bilder vor dem Speichern zu skalieren und zwar mit den Faktoren 1(keine Skalierung), 0.75, 0.5 und 0.25. Für eine Evaluierung der Möglichkeiten den PlugIns, gerade im Hinblick auf die Echtzeitfähigkeit, ist diese Funktion sehr geeignet, da sie Aufschluss darüber gibt ob eine beispielsweise relativ geringe Skalierung des Bildmaterials schon eine ungleich höhere Framerate beim Mosaicing ermöglicht.
Abschließend wird der Prozess mit dem Button $GrabFrames...$(7) gestartet, wobei vorher noch ein Dialogfenster geöffnet wird in dem Speicherort- und Name zu übergeben sind. Nach Bestätigung wird Bild für Bild in das

[3]Unter Berücksichtigung der unterstützten Formate. vgl. Tabelle 2.1

vorher festgelegte Verzeichnis gespeichert[4].
Dabei werden die Dateinamen nach dem Muster $Dateiname000.bmp$ angelegt, wobei Dateiname für den im Dialogfenster übergebenen Namen steht und die anschließende dreistellige Zahl die Bilder einer Sequenz durchnummeriert. Die erste und letzte Datei einer beispielhaften Sequenz „Testsequenz“ mit 25 Frames heißt dann entsprechend $Testsequenz000.bmp$ und $Testsequenz024.bmp$.
Sollten während des Arbeitsprozesses das Control- oder PanoramaFrame geschlossen werden, können diese mit den entsprechenden Einträgen im Menüpunkt $Show(3)$ wieder geöffnet und dargestellt werden.

3.2.2 Offline-Untersuchungen

Um die im vorigen Schritt erstellten Bildsequenzen einem Image Mosaicing zu unterziehen, folgt man nun nachstehenden Bedienschritten. In Abbildung 3.3 lassen sich diese grafisch nachvollziehen.
Über den Eintrag $Offline$ im Menüpunkt $Mosaicing(2)$ gelangt man wieder in ein Dialogfenster in welchem man die gewünschte Bildsequenz auswählt. Dabei ist es nicht entscheidend welches Bild einer Sequenz gewählt wird. Es wird stets die Sequenz übergeben, in der sich das selektierte Bild befindet. Nach Bestätigung erscheint auf dem Desktop ein SelectionFrame(4). Mit Hilfe der hier dargestellten Schieberegler ist es nun möglich einen bildgenauen Bereich für das Mosaicing auszuwählen. Der linke Regler legt dabei das erste Frame aus der Sequenz fest, der rechte entsprechend das letzte. Die Skala hilft dabei der Orientierung. Sie entspricht der Bildfolgenummer.
Mit den Funktionen der in Abbildung 3.1 markierten Bereiche (8) und (9), lassen sich nun noch diverse Einstellungen vornehmen und

[4]Die Bilder werden im Format Bitmap(.bmp) gespeichert.

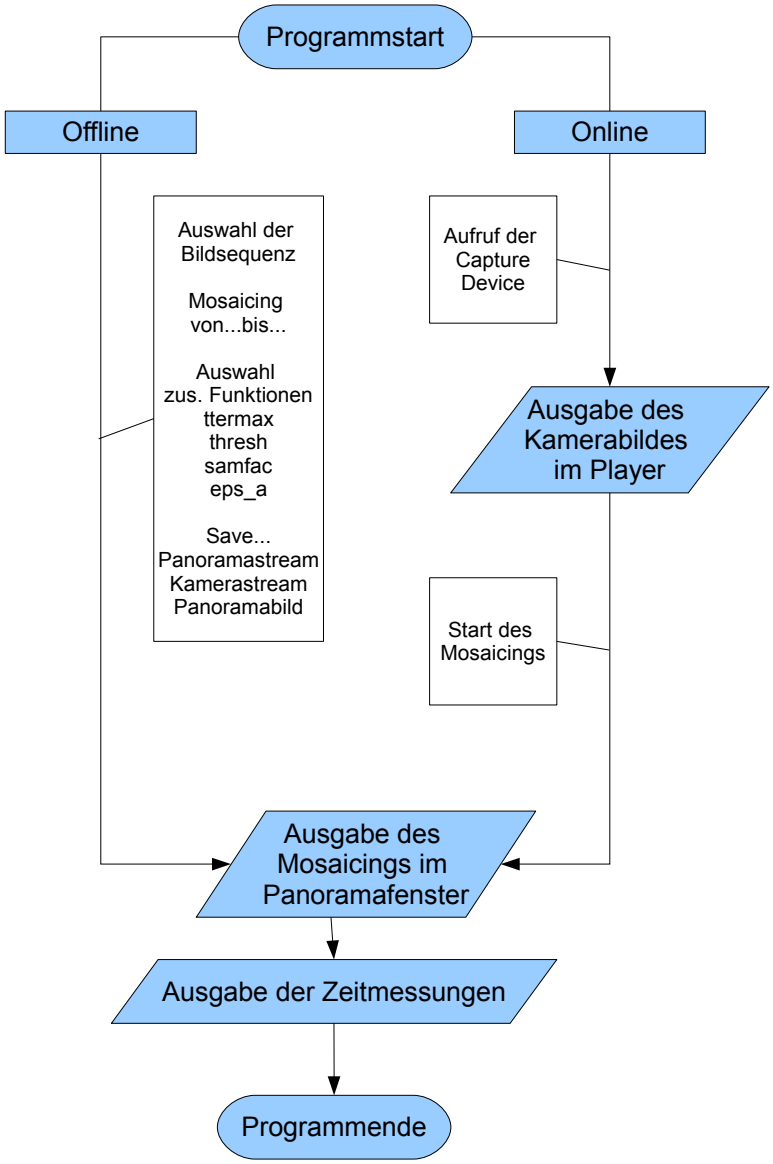


Abbildung 3.3: Flussdiagramm zum Bedienungsablauf beim Mosaicing

zusätzliche Bildausgaben erzeugen. So haben Einstellungen an den Comboboxen(8) Einfluss auf die Berechnung im Mosaicing Algorithmus, während durch Anwahl der Checkboxen(9) zusätzliche Informationen ausgegeben werden können[5].
Mit den Checkboxen im unteren Teil des ControlFrames(10) hat man nun zusätzlich die Möglichkeit, eine Speicherfunktion zu aktivieren. Dabei können im einzelnen, der im SelectionFrame festgelegte Bereich der Kamerasequenz sowie dessen Panoramaerzeugung als AVI und das Panoramaendbild als TIFF[6] gespeichert werden. Hierzu werden, bei getätigter Anwahl der Boxen, nach Abschluss des Mosaicings in gleicher Reihenfolge Dialogfenster geöffnet um vom Benutzer Speicherplatz und Dateiname zu erfragen.
Mit dem Button *StartMosaicing*(11) kann der entsprechende Prozess nun gestartet werden. Das errechnete Panorama wird dabei im PanoramaFrame dargestellt.
Weiterhin erscheint nach Abschluss des Mosaicings stets ein Fenster namens Result, welches alle vom Programm zurückgegebenen und für eine Auswertung relevanten Daten ausgibt. Vor dem Schließen des Fensters hat man die Möglichkeit diese Ergebnisse im XLS-Format abzuspeichern, um sie so direkt zur Auswertung in eine Tabelle exportieren zu können.

3.2.3 Online-Untersuchungen

Während die beiden bisher beschriebenen Funktionen vor allem für erste Tests und Versuche des PlugIns unter Realbedingungen benötigt werden, so ist die Funktion der direkten Übergabe von Kamerabildern an den

[5] siehe [8] für Details
[6] Tagged Image File Format: Dateiformat zur verlustfreien Speicherung von Bilddaten

Mosaicing Algorithmus diejenige, die in einem späteren, regulären Einsatz des PlugIns im Klinikalltag eine Hauptfunktion darstellen wird. Doch auch in dieser ersten Phase wird sie dringend benötigt, gerade um zum Beispiel live zu testen, welche Schwenks und Bewegungen mit der Kamera noch möglich sind, so dass noch eine sinnvolle Verarbeitung und Ausgabe eines Mosaicing Panoramas stattfinden kann.
Durch den Menüeintrag *Device* im Menü $Capture from(1)$ wird eine über FireWire oder USB angeschlossene Kamera registriert und das entsprechende Kamerabild in einem Standard JMF-Player auf dem Desktop dargestellt.
Ein weitere Betätigung von *Online* im Menü *Mosaicing* (2) startet augenblicklich den Verarbeitungsprozess. Das entstehende Panorama wird im PanoramaFrame abgebildet.

3.3 Bereitstellung der Masken

Da Endoskopkameras in der Regel ein kreisrundes Bild liefern, gleichzeitig aber auch einem rechteckigen Bildschirm abgebildet werden, wird über den nicht vom Kamerabild abgedeckten Bereich des Bildschirms eine Maske gelegt. Da die Informationen in diesem Bereich aber für ein Mosaicing nicht relevant, da ohne Informationen sind, muss hier eine Trennung beider Bildteile vorgenommen werden. Mit Hilfe einer erstellten Maske wird also der Bereich des Kamerabildes herausgefiltert.
Die automatische, beziehungsweise semiautomatische Bereitstellung dieser Bildmasken ist ein wesentlicher Bestandteil für eine uneingeschränkt einsetzbare ImageMosaicing Software. Da der Maskenausschnitt jeder Kamera eine unterschiedliche Lage hat und auch mehrere Kameras unterschiedlicher Auflösungen verwendet werden, können hier keine standardisierten Bildmasken verwendet werden. Ferner müssten sowohl vor dem

Start eines Mosaicings einer gewählten Sequenz als auch bei Anschluss einer neuen Kamera beziehungsweise bei Veränderung von deren Auflösung, jeweils entsprechende Masken erstellt werden.
Da dies nicht Teil der vorliegenden Arbeit war, trotzdem aber bis zur Verfügbarkeit einer solchen Erweiterung ein allgemeines Mosaicing möglich sein sollte, wurden eine Reihe von Standardmasken erstellt. So liegen im Ordner *masks* Masken für die Offline-Verarbeitung von SD Material[7] sowie für entsprechend skalierte Bilder[8].
Der Übersichtlichkeit halber sollten Masken welche für bestimmte Bildsequenzen erstellt werden, in Zukunft im gleichen Verzeichnis abgelegt werden. So müssen diese bei mehrfachem Mosaicing einer Sequenz nur einmal angelegt werden. Durch einen entsprechenden Dateinamen bleiben sie lokalisier- und zuweisbar. Die Zählung der zu matchenden Frames wird dabei durch eine im gleichen Verzeichnis abgelegte Maske nicht beeinflusst.

3.4 Methodenbeschreibungen

Im nachfolgenden Abschnitt wird auf alle signifikanten Methoden eingegangen. Eine detaillierte Beschreibung hilft dabei, nachfolgende Arbeiten am PlugIn zu erleichtern.

3.4.1 process/onlineprocess

Die Methoden *process* und *onlineprocess* steuern den Bearbeitungsablauf mit Beginn des eigentlichen Mosaicing Prozesses.
Dabei steuert *process* den Ablauf der Verarbeitung von Offline-

[7] Standard Definition (SD) mit einer Auflösung von 720x576 Pixel
[8] Skalierungsfaktoren 0.75, 0.5 und 0.25

Sequenzen. Diese Methode war bereits Bestandteil der Implementierung von Naderi[8] und musste lediglich um einige Funktionen erweitert werden. Aus ressourcenschonenden Gründen wurde es vermieden, die gesamte ausgewählte Bildsequenz in einen ImageStack zu laden. Hier würde gerade bei längeren Bildsequenzen ein erheblicher Speicherbedarf entstehen. Stattdessen werden die einzelnen Bilder innerhalb einer Schleife direkt aus dem Speicher einem ImageProcessor übergeben (Listing 3.1). Die beiden Prozessoren *curr* und *next*, beschrieben mit zwei aufeinander folgenden Bildern einer Sequenz, werden nun dem Mosaicing-Prozess *Proc.motion* zugeführt(Zeile 4). Am Ende der Schleife wird der Wert von *next* in den ImageProcessor *curr* geschrieben(18). So muss im neu beginnenden Schleifendurchlauf nur ein neues Bild aus dem Speicher gelesen werden(3).

Listing 3.1: Übergabe der zu verarbeitenden Bilder an *Proc.motion*

```
1  for (int i =startfr;i<endfr;i++){
2     img = imgLoader.openImage(Dir + imgName + form.format(i) + imgFmt);
3     next = img.getProcessor().convertToByte(false);
4     Proc.motion(curr, next, rect, par, maski, maskn);
5     consoleDataOutput(imgCount);
6     if (outputRGB)   Extend.ext(w, next, Proc.a, outputRGB, diffImg, doExt
          , par.XMethod);
7          else        Extend.ext(w, next, Proc.a, outputRGB, diffImg, doExt
               , par.XMethod);
8     GUI.paintPan(w.panoim, 10, 10);
9     if (xArea)     GUI.paintXarea(w.X_Area, width, 10);
10    if (diffImg)   GUI.paintDiff(w.Xerr, width, 10);
11    if (delay)   try{Thread.sleep(500);}catch(InterruptedException e){};
12    if (savePanoStream){
13          savematch.addSlice("Image"+i, w.panoim.duplicate());
14     }
15    if (saveEndoStream){
16          endocam.addSlice("", curr);
17     }
18         curr = next;
19 }
```

Zusätzlich werden innerhalb dieser Schleife noch zwei weiter oben angelegte Stacks gefüllt(12-16). *savematch* wird, sofern der Bediener diese Funktion angewählt hat, das aktuelle Panoramabild *w.panoim* übergeben, *savecam* das aktuelle Kamerabild.

Listing 3.2: Übergabe der ImageStacks an den AVIWriter

```
if (savePanoStream){
        imp.setStack("", savematch);
        imp.show();
        IJ.runPlugIn("ij.plugin.filter.AVIWriter", "");
}
if(saveEndoStream){
        imp.setStack("", endocam);
        imp.show();
        IJ.runPlugIn("ij.plugin.filter.AVIWriter", "");
}
if(savePanorama){
        ImagePlus panoimage = new ImagePlus();
        panoimage.setProcessor("", w.panoim);
        FileSaver panoimg = new FileSaver(panoimage);
        panoimg.saveAsTiff();
}
```

Nach dem Schleifendurchlauf werden diese Stacks, dem von ImageJ zur Verfügung gestellten PlugIn AVIWriter übergeben(Listing 3.2). Dieses PlugIn erzeugt von den Bildern der Stacks, wie der Name schon sagt, Videofiles im AVI-Format.

Nach Aufruf von *onlineprocess* wird, wie in Listing 3.3 zu sehen, die Methode *captureFromDevice* als nebenläufiger Thread gestartet. Dazu wird eine neue Instanz cfd der Methode aufgerufen und diese der Klasse Thread übergeben. Thread stellt dabei einen eigenständigen Ausführungsstrang dar, der anschließend gestartet wird(3).

Listing 3.3: Starten des nebenläufigen Threads

```
captureFromDevice cfd = new captureFromDevice();
Thread cap = new Thread(cfd);
cap.start();
```

Ähnlich wie in *process* werden hier nun Maskenregistrierung und ähnliches durchgeführt. Der Unterschied liegt im Aufruf der zu übergebenden Bilder. Dazu wird in *onlineprocess* ein ColorProcessor angelegt, welchem direkt ein Integer-Array übergeben wird. Dieses wurde zuvor aus dem aktuellen Bild der angeschlossenen Kamera erstellt (siehe Abschnitt 3.4.4). Die Übergabe von *curr* und *next* an den Verarbeitungsprozess sowie der Umschreibevorgang der beiden Prozessoren, laufen dann

wieder simultan zu *process* ab.

3.4.2 openFile

Gerade für den Einsatz eines Programms im Umfeld von ausschließlichen Bedienern, ist es wichtig alle eventuell auftretenden Ausnahmen abzufangen und den Benutzer bei Problemen entsprechend zu informieren.
Die Methode *openFile* (Listing 3.4) ist ein gutes Beispiel für das Abfangen aller möglichen Ausnahmen. Beim Eintreten einer Ausnahme wird der Benutzer jeweils über ein Dialogfenster über das entsprechende Problem informiert.

Listing 3.4: Fehlerbehandlung beim Öffnen eines Files

```
1  public void openFile(String filename) {
2   String mediaFile = filename;
3   Player player = null;
4   URL url = null;
5    try {
6      if ((url = new URL(mediaFile)) == null) {
7          JOptionPane.showMessageDialog(null, "Can´t build URL for " +
               mediaFile);
8          return;
9      }
10   try {
11         player = Manager.createPlayer(url);
12     } catch (NoPlayerException e) {
13         JOptionPane.showMessageDialog(null,"Error: " + e);
14     }
15   }catch (MalformedURLException e) {
16         JOptionPane.showMessageDialog(null,"Error: " + e);
17   }catch (IOException e) {
18         JOptionPane.showMessageDialog(null,"Error: " + e);
19   }
20   if (player != null) {
21      this.filename = filename;
22      JMFrame jmframe = new JMFrame(player, filename);
23      desktop.add(jmframe);
24   }
25  }
```

Ein fehlender oder falscher Dateiname löst in Zeile 6-7 eine erste Exception aus. Die Methode *createPlayer*, die einen Player für die übergebende URL anlegt, wirft nun zwei weitere Fehlermeldungen aus. Kann kein

entsprechender Player gefunden werden[9] wird eine $NoPlayerException$ ausgeworfen. Bei sonstigen Problemen mit dem Verbinden der Datenquelle mit dem Player, kommt es zu einer $IOException$.
Desweiteren wird von der Klasse URL noch eine $MalformedURLException$ ausgeworfen, wenn die URL in einem falschen Format angegeben wurde oder ein unbekanntes Übertragungsprotokoll vorliegt.

3.4.3 JMFrame

Anhand der Initialisierung und anschließenden Darstellung eines Medienplayers der JMF Klasse *Player*, lassen sich die internen Verarbeitungsketten[4] sehr gut veranschaulichen.
Ein Player befindet sich während seiner Existenz ständig in einem bestimmten Zustand. Der Übergang von einem in den nächsten Zustand wird durch Methodenaufrufe initiiert.
So befindet sich der Player bis Zeile 15 im *Unrealized*-Zustand. Dies ist der Anfangszustand jedes Players. Durch den Aufruf der Methode *realize* erfolgt eine Analyse des Eingangsmediums. Diesem, *Realizing* genannten Zustand, folgt nach erfolgreicher Analyse der Zustand *Realized*. Die Mediendatenstrukturen sind nun vollständig erzeugt und bereit zur Weiterverarbeitung.
Der anschließende Aufruf der Methode $prefetch(24)$ überführt den Player in den Zustand $Prefetching$, in welchem der Inhalt der übergebenen URL gelesen wird und schließt nach Abschluss des Lesevorgangs den Zustand $Prefetched$ an.
Um zu gewährleisten, dass ein neuer Zustand erst gestartet wird wenn der vorangegangene abgeschlossen wurde, werden in den Zeilen 22 und 27

[9] bedeutet im allgemeinen, dass das Format nicht gelesen werden kann, es also von JMF nicht unterstützt wird

vor dem entsprechenden Aufruf die Ereignisse *RealizeCompleteEvent* und *PrefetchCompleteEvent* abgefragt.

Listing 3.5: Anlegen einer Instanz eines JMPlayers

```
1  class JMFrame extends JInternalFrame implements ControllerListener {
2      Component visual = null;
3      Component control = null;
4      int videoWidth = 0;
5      int videoHeight = 0;
6      int controlHeight = 30;
7      int insetWidth = 10;
8      int insetHeight = 30;
9      public JMFrame(Player player, String title) {
10         super("IMPlayer", true, true, true, true);
11         setResizable(false);
12         setVisible(true);
13         mplayer = player;
14         mplayer.addControllerListener((ControllerListener) this);
15         mplayer.realize();
16         addInternalFrameListener( new InternalFrameAdapter() {
17             public void internalFrameClosing(InternalFrameEvent ife) {
18                 mplayer.close();
19             }
20         } );
21         }
22     public void controllerUpdate(ControllerEvent ce) {
23         if (ce instanceof RealizeCompleteEvent) {
24             mplayer.prefetch();
25         } else if (ce instanceof PrefetchCompleteEvent) {
26             if (visual != null)
27                 return;
28             if ((visual = mplayer.getVisualComponent()) != null) {
29                 Dimension size = visual.getPreferredSize();
30                 videoWidth = size.width;
31                 videoHeight = size.height;
32                 getContentPane().add("Center", visual);
33             } else
34                 videoWidth = 320;
35             if ((control = mplayer.getControlPanelComponent()) != null) {
36                 controlHeight = control.getPreferredSize().height;
37                 getContentPane().add("South", control);
38             }
39             setSize(videoWidth + insetWidth,
40                     videoHeight + controlHeight + insetHeight);
41             validate();
42             mplayer.start();
43         } else if (ce instanceof EndOfMediaEvent) {
44             mplayer.setMediaTime(new Time(0));
45             mplayer.start();
46         }
47     }
48 }
```

Die beiden grafischen Komponenten des Players, *visual* und *control*, werden nun dem JInternalFrame übergeben(28-38). Die einzelnen Größen der Elemente werden addiert(39-40) und das JInternalFrame entsprechend der Gesamtgröße angepasst.
Mit der Methode *start*(42) schließlich beginnt der Zustand *started*. Die Verarbeitung läuft nun.

3.4.4 captureFromDevice

Die in Listing 3.6 abgebildete Klasse *captureFromDevice* welche, als separater Thread aus $Control\_.onlineprocess$ gestartet wird, implementiert in Zeile 1 das Interface *Runnable*, welches wiederum die abstrakte Methode $run()$(2) implementiert. Durch das Überschreiben dieser Methode erhält man eine Instanz, welche als nebenläufiger Thread ausgeführt wird.
Innerhalb der $run()$-Methode werden nun in einer Endlosschleife Frames einer angeschlossenen Kamera gegrabbt und diese übergeben. Dazu wird das vom JMF bereitgestellte Interface $FrameGrabbingControl$ verwendet.
Mit $grabFrame$, der einzigen Methode die $FrameGrabbbingControl$ implementiert, wird das aktuelle Bild nun aus dem Videostream gegrabbt und einem Buffer übergeben(11). Mit Hilfe der Methode $instanceof$ wird überprüft, ob es sich bei dem Inhalt des Buffers um ein Format der Form $javax.media.RGBFormat$ handelt. Ist dies der Fall, schließt sich ein cast von $javax.media.Format$ in $javax.media.RGBFormat$ an(15). Um das zur Übergabe der Bilddaten angelegte Integer-Array $GUI.imgData$ zu deklarieren, werden mit Hilfe der Methoden $getSize()$, $getHeight()$ und $getWidth()$ die Dimensionen der Kamerabilder abgefragt und entsprechend übergeben(16-22).

Listing 3.6: Framegrabbing im nebenläufigen Thread

```
1  public class captureFromDevice implements Runnable{
2   public void run(){
3     int bytePerPel, bitPerPel, bytePerLine, redOffset, greenOffset,
          blueOffset, li, co, a, b;
4     int dv_h = 576;
5     int dv_w = 720;
6     boolean flipped;
7     try{
8      for(int i=0;; i++){
9        FrameGrabbingControl fgc =
10       (FrameGrabbingControl)GUI.player.getControl("javax.media.control.
             FrameGrabbingControl");
11       Buffer buf = fgc.grabFrame();
12       javax.media.Format form = buf.getFormat();
13       javax.media.format.VideoFormat videoformat = (javax.media.format.
             VideoFormat) form;
14       if(form instanceof javax.media.format.RGBFormat){
15         javax.media.format.RGBFormat rgbForm = (javax.media.format.
               RGBFormat)form;
16         double Height = rgbForm.getSize().getHeight();
17         double Width = rgbForm.getSize().getWidth();
18         bitPerPel = rgbForm.getBitsPerPixel();
19         flipped = rgbForm.getFlipped() == javax.media.Format.TRUE;
20         int height = (int)Height;
21         int width = (int)Width;
22         GUI.imgData = new int[width*height];
23         if (rgbForm.getDataType() == javax.media.Format.byteArray &&
               bitPerPel == 24){
24           redOffset = rgbForm.getRedMask()-1;
25           greenOffset = rgbForm.getGreenMask()-1;
26           blueOffset = rgbForm.getBlueMask()-1;
27           bytePerLine = rgbForm.getLineStride();
28           bytePerPel = rgbForm.getPixelStride();
29           byte[] dataIn = (byte[])buf.getData();
30           for(li=0,b=0;li<height;li++){
31             a=(flipped ? height - 1- li :li) * bytePerLine;
32             for(co =0; co < width; co++, b++){
33               GUI.imgData[b] =(dataIn[a+redOffset]&0xff) <<16 |
34                               (dataIn[a+greenOffset]&0xff)<<8 |
35                               (dataIn[a+blueOffset]&0xff);
36                               a+=bytePerPel;
37             }
38            }
39           }
40         }else if(videoformat instanceof com.sun.media.format.
               AviVideoFormat){
41                Object dataObj = buf.getData();
42                if (dataObj instanceof byte[]){
43                    GUI.data = (byte[])dataObj;}
44                    for(int p=0, j=0; p<dv_w*dv_h;p++,j+=3){
45                       GUI.imgData[p] = (0xff&GUI.data[j]<<16)|
46                       (0xff&GUI.data[(j)+1]<<8)|
47                        0xff&GUI.data[(j)+2];
48                     }
49                }
50       }
51     }catch (Exception e){
52              JOptionPane.showMessageDialog(null, "Error: " + e);
```

```
53          }
54      }
55  }
```

In der anschließenden if-Anweisung wird getestet ob es sich bei dem übergebenem Format um ein byte-Array mit 24Bit pro Pixel handelt. Der Inhalt des Buffers wird nun in ein Byte-Array gecastet und entsprechend den Vorgaben aus RGBFormat geschrieben. Konkret muss das Bild gedreht werden, wenn es $flipped$ übertragen wurde(31). Außerdem wird entsprechend der Anordnung der Werte für R, G und B im Array, eine Maskierung durchgeführt. Die Maskierungsvorschriften werden in den Zeilen 24-26 mit Hilfe der Methoden $getRedMask$, $getGreenMask$ und $getBlueMask$ abgefragt.
Im eigentlichen Schreibprozess des zu übergebenden Integer-Arrays $GUI.imgData$(33-35), wird (für jedes Pixel) jede Farbkomponente genommen, die entsprechende Farbmaske addiert, eine Bitmaskierung durchgeführt um die Komponente zu isolieren und anschließend eine entsprechende Verschiebung vorgenommen. In Zeile 36 wird zum nächsten Pixel gesprungen, worauf der Schreibprozess von neuem beginnt.
Eine weitere Verarbeitung des Arrays schließt sich nun in *onlineprocess* an.
Eine während der Arbeiten an dieser Benutzerschnittstelle angeschlossene DV-Kamera wird von Java mit dem Format $sun.com.media.format.AVIVideoFormat$ detektiert. Entsprechend schließt sich ab Zeile 40 eine entsprechende $else\ if$-Anweisung an, ich welcher Frames dieser Kamera übergeben werden.
Zukünftig angeschlossene Kameras müssten hier entsprechend ihrer übergebenen Formate noch berücksichtigt werden. Dazu können die jeweils detektierten Formate in der JMFRegistry eingesehen und

die Videodaten entsprechend der übergebenen Formateigenschaften[10] verarbeitet werden.

3.4.5 frameGrab

Nach Auswahl einer bestimmten Sequenz in einem Videostream wird eine Instanz der Methode *frameGrab* aufgerufen. Hier werden zunächst zwei Interfaces instanziiert.
Zum einen *FramePositioningControl*(Zeile 6), eine Schnittstelle zur Kontrolle der exakten zeitlichen (bildgenauen) Positionierung in einem Videostream. In einer Schleife, die von zuvor eingestelltem Startframe bis Endframe läuft, wird nun mit Hilfe der von *FramePositioningControl* implementierten Methode *seek* das entsprechende Frame im Stream angezeigt(9), um es anschließend zu grabben.
Um dieses Grabbing zu realisieren wurde zuvor das zweite Interface namens *FrameGrabbingControl*(7) instanziiert. Es wird nun mit der Methode *grabFrame* das gerade angezeigte Frame in einem Buffer abgelegt.
In den Zeilen 17 bis 41 werden die Frames vor einem Speichern nach entsprechender Vorgabe des Benutzers skaliert.
Die Methode *getScaledInstance* verfügt über eine Reihe von Skalierungsarten. Neben einem Standardalgorithmus(SCALE-DEFAULT) gibt es beispielsweise auch einen auf höhere Geschwindigkeit ausgelegten Algorithmus (SCALE-FAST). Da bei dem Prozess des Framegrabbings in dieser Applikation aber die Geschwindigkeit eine eher untergeordnete Rolle spielt, es dafür aber umso wichtiger ist Bilder mit möglichst hoher Qualität vorliegen zu haben, wurde hier ein weiterer zur Verfügung stehender Algorithmus (SCALE-SMOOTH) verwendet, der mit eben dieser Präferenz aufwartet.

[10] Diese umfassen im wesentlichen Auflösung, Datentyp, Framerate, Maskierungen, Pixel- und Linestrides.

Listing 3.7: Grabbing und Speichern der Frames von Videostreams

```
1  public frameGrab(int startfr, int endfr, int mod, int Scale, String
       filename, String Dir, Player mplayer) throws Exception {
2      int countfr = startfr;
3      int i = 0;
4      int Height, Width;
5      DecimalFormat form = new DecimalFormat("000");
6      FramePositioningControl fpc = (FramePositioningControl)mplayer.
           getControl("javax.media.control.FramePositioningControl");
7      FrameGrabbingControl fgc = (FrameGrabbingControl)mplayer.getControl("
           javax.media.control.FrameGrabbingControl");
8      for(countfr = startfr; countfr>=startfr & countfr<=endfr; countfr++ )
           {
9        fpc.seek(countfr) ;
10       if (countfr % mod ==0 ){
11         Buffer buf = fgc.grabFrame();
12         Format format = buf.getFormat();
13         javax.media.format.RGBFormat rgbformat = (javax.media.format.
               RGBFormat) format;
14         double height=rgbformat.getSize().getHeight();
15         double width=rgbformat.getSize().getWidth();
16         Image img = (new BufferToImage((VideoFormat)buf.getFormat()).
               createImage(buf));
17         if(Scale == 1){
18             height = height*1.0;
19             width = width*1.0;
20             Weight = (int)height;
21             Width = (int)width;
22             img=img.getScaledInstance(Width, Height, Image.SCALE_SMOOTH);
23          }else if(Scale == 2){
24             height = height*0.75;
25             width = width*0.75;
26             Height=(int)height;
27             Width=(int)width;
28             img=img.getScaledInstance(Width, Height, Image.SCALE_SMOOTH);
29          }else if(Scale == 3){
30             height = height*0.5;
31             width = width*0.5;
32             Height=(int)height;
33             Width=(int)width;
34             img=img.getScaledInstance(Width, Height, Image.SCALE_SMOOTH);
35          }else if(Scale == 4){
36             height = height*0.25;
37             width = width*0.25;
38             Height=(int)height;
39             Width=(int)width;
40             img=img.getScaledInstance(Width, Height, Image.SCALE_SMOOTH);
41          }
42          BufferedImage buffImg = new BufferedImage(img.getWidth(null),img
                .getHeight(null), BufferedImage.TYPE_INT_RGB);
43          Graphics2D g = buffImg.createGraphics();
44          g.drawImage(img, null, null);
45          ImageIO.write(buffImg, "bmp", new File(Dir + filename + form.
                format(i) + ".bmp"));
46          i++;
47        }
48     }
49 }
```

In Zeile 42 wird nun ein Objekt der Klasse $BufferedImage$ angelegt. $BufferedImage$ beschreibt dabei einen ansprechbaren Buffer des Typs $Image$. Neben der aktuellen Breite und Höhe wird zusätzlich noch der Typ des zu schreibenden Frames übergeben. In diesem Fall $TYPE\_INT\_RGB$, also ein Bild mit 8-bit RGB Farbkomponenten, geschrieben als Integer-Pixel.
Mit dem Aufruf der Methode $drawImage$ wird das erzeugte $Graphics2D$ Objekt gerendert(44). $Graphics2D$ wurde vorher der Inhalt von $BufferedImage$, welcher mit der Methode $createGraphics()$ in ein $Graphics2D$-Objekt gecastet wurde, übergeben.
Schließlich wird das aktuell gerenderte Bild mit der Methode $write$ der Klasse $ImageIO$ in das angegebene Verzeichnis geschrieben(45).

Kapitel 4

Funktionstest

Eine Zusammenfassung der Erkenntnisse einer ersten Inbetriebnahme der grafischen Benutzerschnittstelle soll im Folgenden, Aufschluss über Bedienbarkeit, Funktionalität und Ergonomie geben.
Fängt man in der Verarbeitungskette des PlugIns vorne an, so steht hier zuerst einmal die Kamera. Für die meisten Testzwecke während der Entwicklung der Schnittstelle wurde eine übliche Webcam, welche über einen USB-Anschluss verfügte, verwendet. Diese wurde stets vom JMF erkannt und konnte so problemlos über das PlugIn aufgerufen und gestartet werden. Zum Testen der Konnektivität des JMF mit einer FireWire-Schnittstelle wurde desweiteren eine Canon XL-1[1] angeschlossen. Auch hier gab es keinerlei Probleme bezüglich dem Ansprechen der Kamera aus dem PlugIn.
Ein weiterer Grund für diesen Anschlusstest war die nötige Bestätigung für die Funktionalität des DV-Codecs. Nach einem bereits vorher erfolgreich durchgeführtem Test mit DV-codiertem Material, welches von Festplatte eingelesen wurde, konnte nun auch die einwandfreie Arbeit des Codecs bestätigt werden.

[1] Digitaler MiniDV-Camcorder

Diese verwendeten Kameras können hier jedoch nicht stellvertretend für alle in Zukunft verwendeten Geräte angenommen werden. Der Anschluss neuer Kameras zieht hier unter Umständen noch weitere Veränderungen im Programm mit sich.

Die Abbildung von Streams sowohl aus Dateien als auch von Kamera werden stets im Player eingefügt und die Größe des Fensters der Auflösung angepasst.
Allerdings kommt es hier bei der Darstellung der Kamerabilder zum Teil zu rasterähnlichen Strukturen die mit zunehmender Skalierung des Playerfensters gröber werden und schließlich verschwinden. Die Ursache dafür konnte nicht endgültig geklärt werden. Es scheint jedoch auch mit der Integration des Players in ein JFrame zusammenzuhängen.
Wichtig zu erwähnen ist jedoch, dass sich diese Artefakte nur auf die Darstellung am Monitor beschränken und bei einem Capturing nicht vorhanden sind. Sie beeinträchtigen also den Bildverarbeitungsprozess des Mosaicings nicht.

Die Mosaicing-Funktion von gespeicherten Bildsequenzen lässt sich problemlos starten. Erste Eindrücke der vom Programm ausgegebenen Framerate überzeugen auch hinsichtlich der Echtzeitfähigkeit.
Beim Aufbau des Panoramas erkennt man jedoch, dass der verwendete Algorithmus hier schnell an seine Grenzen stößt. So lässt sich bei der Wahl einer beliebigen Sequenz, die sich unter Umständen noch durch Kontur- oder Kontrastschwäche, schneller Kamerabewegung oder störende Glanzlichter auszeichnet, schon nach wenigen Frames kein sinnvolles Panorama mehr erkennen.

Die Funktion der direkten Übergabe von Frames einer Kamera an

den Mosaicing-Prozess ist ebenfalls möglich. Hier kommen zu eben genannten Problemen noch weitere hinzu. So fehlt die in 3.3 bereits angesprochene Erstellung der Masken. Auch eine genaue zeitliche Steuerung der Abläufe innerhalb des gesamten Prozesses muss in einem weiteren Schritt noch organisiert werden.
Die hier realisierten Möglichkeiten der Anbindung von Mediencontent an das bestehende Image Mosaicing PlugIn können nun dazu genutzt werden, die Entwicklung des PlugIns weiter voranzutreiben.

Kapitel 5

Fazit

5.1 Zusammenfassung

Die mir im Rahmen dieser Diplomarbeit gestellten Aufgaben konnten hier weitestgehend umgesetzt werden. So wurde mit der vorliegenden grafischen Benutzerschnittstelle ein Werkzeug geschaffen, dass in einem weiteren Schritt, hin zu einem echtzeitfähigen Image Mosaicing PlugIn dazu verwendet werden kann, erste Endoskopsequenzen zu verarbeiten und somit vor allem auch, die Ergebnisse aus medizinischer Sicht und unter Berücksichtigung der daraus resultierenden, spezifischen Anforderungen zu evaluieren.
Um hier anknüpfen und weitere Arbeiten innerhalb dieses Projekts möglichst reibungslos fortsetzen zu können, wurde bewusst eine detaillierte Darstellung der einzelnen Arbeiten gewählt. Auch sämtliche Funktionsbeschreibungen wurden eingehend beschrieben. Dieser umfassenden Dokumentation wurde hier besondere Beachtung geschenkt.

5.2 Ausblick

Mit der Realisierung dieser Arbeit wurde ein weiterer Schritt hin zu einer uneingeschränkt einsetzbaren Image Mosaicing Software für die medizinische Endoskopie getan. Im Laufe der Bearbeitungszeit wurde dabei mehr und mehr ersichtlich wie umfangreich dieses Projekt werden würde. Je tiefer ich in dieses Projekt einstieg, umso mehr Fragen und Probleme wurden deutlich, die es noch zu lösen gilt, bevor ein Einsatz unter Realbedingungen möglich ist. Im folgenden einige Überlegungen und Anregungen für weitere Arbeitsgebiete innerhalb dieses Projekts:

- Der bereits von Naderi angesprochene Punkt des Multithreadings wurde in dieser Arbeit ansatzweise umgesetzt. In einem ersten Schritt wurde der Prozess des Framegrabbings einem nebenläufigen Thread zugewiesen. Hier lassen sich sicher noch mehrere Stränge parallel ausführen, beziehungsweise deren Rechenaufwand auch auf unterschiedliche Prozessoren verteilen.

- Ein nicht zu unterschätzender Punkt, ist die Verarbeitungskette der Bilddaten innerhalb des gesamten Mosaicing Prozesses. Gemeint sind damit die Bilddaten von der Aufnahme in der Kamera bis zum fertigen Panoramabild. Für ein zufriedenstellendes Ergebnis des Matchings wäre es wichtig die Bildqualität am Kameraausgang möglichst bis zur Übergabe an den Mosaicing-Prozess zu erhalten und Artefakte und andere Bildstörungen weitestgehend auszuschliessen[1].
  Die Verwendung der VfW-Schnittstelle wäre hier als Beispiel anzubringen. Die Anbindung von Mediencontent lässt sich damit relativ einfach gestalten, allerdings kann nicht genau beurteilt werden, in

[1] siehe hierzu die Vorüberlegungen von Breiderhoff[2]

welcher Weise das Bildmaterial hier verändert wird. Eine konkrete Analyse und entsprechende Optimierung der gesamten Verarbeitungskette wäre also sinnvoll und wünschenswert.

- Ein Weiterer, in einem Gespräch mit Herrn Scholz, von ihm erörterter Punkt, stellt die Möglichkeit dar, die Raumkoordinaten der Endoskopkamera, welche vom VN-System[2] ermittelt und ausgegeben werden, in der Weise mit dem Image Mosaicing Prozess zu verknüpfen, dass eine Art 3D-Image Mosaicing möglich wäre[3].

[2]Visuelles Navigationssystem für die Endoskopie
[3]siehe hierzu [6]

Abbildungsverzeichnis

Listings

Literaturverzeichnis

[1] Chris Adamson. *QuickTime for Java A Developer´s Notebook*. O´Reilly, 2005.

[2] Beate Breiderhoff. Verfahren zum automatisierten image mosaicing bei endoskopischen videoaufnahmen. Master's thesis, University of Applied Sciences Cologne, 2006.

[3] W Burger M J Burge. *Digitale Bildverarbeitung Eine Einführung mit Java und ImageJ*. x media press Springer Verlag, 2006.

[4] Horst M Eidenberger Roman Divotkey. *Medienverarbeitung in Java*. dpunkt.verlag, 2004.

[5] M Kourogi T Kurata J Hoshino. Real-time image mosaicing from a video sequence. *Procs ICIP99*, 4:133–137, 1999.

[6] Wolfgang Konen. 3d-navigation und bildverarbeitung in der medizinischen endoskopie. 2008.

[7] SUN Microsystems. Jmf 2.1.1 - supported formats.

[8] Martin Naderi. Implementierung eines echzeitverfahrens zur erstellung von bildmosaiken aus endoskopischen videosequenzen, 2007.

[9] Wolfgang Konen Beate Breiderhoff Martin Scholz. Real-time image mosaic for endoscopic video sequences. 2007.

Anhang A

Anhang

Neben der hier folgenden Anleitung zur Inbetriebnahme der einzelnen Softwarekomponenten, ist dem Anhang eine CD-ROM beigefügt, in der sich dieses Dokument als PDF, die Quellcodes aller Klassen, deren generierte Javadoc und alle relevanten Literaturverweise aus dem Internet befinden.

A.1 Inbetriebnahme

Es folgt eine knappe Installationsanweisung sowie einige Hinweise,die bei der Installation der benötigten Softwarekomponenten zu beachten sind. Wichtig ist dabei, die Schritte in der hier aufgelisteten Reihenfolge zu bearbeiten.

A.1.1 Systemvoraussetzungen

Da alle verwendeten Komponenten plattformunabhängig lauffähig sind, ist hier eine sehr große Systemvariabilität gegeben. So wird das PlugIn sowohl auf allen gängigen Versionen der Betriebssysteme von Microsoft Windows als auch auf UNIX-Plattformen zum Einsatz kommen können. Demgegenüber stehen die Hardwareanforderungen, welche schon spe-

zifischer sind. Zwar wird man ein einfaches Grabbing und Matching von einigen Frames auf einem aktuellen Standard-PC realisieren können, doch für ein Arbeiten mit langen Sequenzen oder der Onlineübergabe von Frames werden hier bedeutend größere Rechen- und Speicherkapazitäten benötigt. Gerade im Hinblick auf die zukünftige Verwendung von Bildmaterial hochauflösender Kameras, werden hier noch weit mehr Kapazitäten benötigt.
Auch wäre es sicherlich lohnend, das Programm entsprechend den aktuellen Multicore-Prozessoren anzupassen und die Bearbeitung der verschiedenen Threads auf unterschiedliche Prozessoren aufzuteilen. Aktuell werden die Vorteile dieser Prozessorgeneration noch nicht genutzt.

A.1.2 Installation

Sinnvollerweise wird für eine finale, vom Klinikpersonal während einer Operation eingesetzten Version dieses PlugIns, keinerlei Entwicklungsumgebung oder Compiler benötigt. So würde dann also nur die von Java benötigte Runtime Environment (JRE) installiert werden müssen[1]. Im momentanen Stadium allerdings ist es als durchaus sinnvoll zu erachten, die oben genannten Javakomponenten für Änderungen am Programm zur Verfügung zu haben.
So ist es sicherlich das Beste die aktuelle Version des Java Development Kit (JDK)[2], zu installieren. Hierzu sind, wie bei allen nachfolgenden Schritten, die Installationsanleitungen der Hersteller zu befolgen. Eventuell muss nach der Installation des JDK noch die Lokation des Java Compilers im Systempfad gesetzt werden. Auch hierzu bitte die Anwei-

[1] Dieses wird bei der Installation von ImageJ mitgeliefert, und muss dann nicht separat installiert werden.

[2] Aufgrund einiger etwas verwirrender Produktnamenwechsel auch unter dem Namen Java(2) Standard Edition (J(2)SE) zu finden.

sungen des Herstellers Sun beachten.
Für die anschließende Installation des JMF ebenfalls die herstellerspezifischen Anweisungen beachten. Es ist darauf zu achten, dass für eine Installation unter Windows, das Windows Performance Pack und für eine Installation unter Linux/Solaris, ebenfalls das entsprechende Performance Pack ausgewählt wird. Zwar gibt es eine plattformunabhängige Cross Platform Version(welche auch auf einem Mac-System lauffähig wäre), diese unterstützt allerdings kein Capturing von Videodaten. Die entsprechenden Systemvariablen werden bei der Installation des JMF automatisch gesetzt, sollten aber nach Abschluss der Installation überprüft werden. Folgende Werte sollten zu finden sein:

CLASSPATH=%JMFHOME%/lib/jmf.jar;
%JMFHOME%/lib/sound.jar;.%CLASSPATH%

und

PATH=%WINDIR&/System32;PATH%

wobei JMFHOME in der ersten Zeile für das Verzeichnis steht, in dem das JMF installiert wurde. Einen abschließenden Funktionstest kann man mit dem von Sun zur Verfügung gestellten Tool JMF Diagnostics[3] durchführen.
Um DV-codierte Videoformate lesen zu können, muss nun noch der entsprechende Codec von Panasonic[4] installiert werden. Dazu Rechtsklick auf $Panadv.inf$ und anschließend auf Installieren. Der Codec installiert und registriert sich nun selbstständig.
Alternativ bietet JMF mit dem in der Standardinstallation mitgeliefertem Tool JMFRegistry[5] auch die Möglichkeit, Codecs und auch Devices

[3]Zu finden unter http://java.sun.com/products/java-media/jmf/2.1.1/jmfdiagnostics.html.
[4]Zu finden unter http://users.tpg.com.au/mtam/install_panvfwdv.htm
[5]Nach der Standardinstallation zu finden unter $Start > Programme > JavaMediaFramework2.1.1e > JMFRegisry$.

selbst zu registrieren, beziehungsweise zu detektieren.
Um dem ImageJ PlugIn möglichst alle zur Verfügung stehenden Systemressourcen zur Verfügung zu stellen, empfiehlt es sich den Wert der Speicherzuweisung, der beim Starten von ImageJ registriert wird, manuell zu ändern. Unter $Edit > Options > Memory...$ kann hier ein maximaler Wert angegeben werden. Es empfiehlt sich, nicht mehr als 75% des tatsächlich vorhandenen Arbeitsspeichers zu übergeben.
Eine Wertänderung in diesem Menüpunkt, wirkt sich auf die Konfigurationsdatei von ImageJ[6]aus. Die betroffene Zeile sieht (bei der Übergabe eines maximalen Wertes von beispielsweise 700MB) folgendermaßen aus:

-Xmx700m -cp ij.jar ij.ImageJ

Alternativ kann die entsprechende Konfigurationsdatei natürlich auch manuell in einem Editor geändert werden.

[6]ImageJ.cfg; Zu finden im Installationsverzeichnis von ImageJ