GEORG-AUGUST-UNIVERSITÄT
GÖTTINGEN

ISSN 1612-6793
Nr. zai-bsc-2017-27

Bachelorarbeit

im Studiengang "Angewandte Informatik"

Entwicklung eines Video-Export Plugins für die Modellierungssoftware GroIMP

Dominick Leppich

Institut für Informatik

Bachelor- und Masterarbeiten
des Zentrums für angewandte Informatik
an der Georg-August-Universität Göttingen

29. September 2017

Georg-August-Universität Göttingen
Institut für Informatik

Goldschmidtstraße 7
37077 Göttingen
Germany

☎ +49 (551) 39-172000
FAX +49 (551) 39-14403
✉ office@informatik.uni-goettingen.de
🌍 www.informatik.uni-goettingen.de

Erstbetreuer: Prof. Dr. Winfried Kurth
Zweitbetreuer: Dr. rer. nat. Patrick Harms

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Göttingen, den 29. September 2017

Inhaltsverzeichnis

Kapitel 1

Einleitung

Das computergestützte Erzeugen von fotorealistischen Bildern von Pflanzen und natürlichen Umgebungen ist schon seit vielen Jahren Thema vieler Forschungen und Entwicklungen. Zwar gibt es eine Vielzahl an Programmen, die eine naturgetreue Modellierung erlauben, jedoch ist das Resultat eines solchen Arbeitsvorganges stets dasselbe: ein einziges Modell. Möchte man aus diesem Modell (beispielsweise einem Baum) einen computergenerierten Wald erzeugen, sähe dies wenig natürlich aus, da ein Baum dem nächsten gleichen würde.

Um die Modellierung von natürlichem Wachstum und daraus resultierenden Pflanzen zu erleichtern und weiterhin die Möglichkeit zu bieten eine Zufallskomponente mit einfließen zu lassen, wurde die Modellierungssoftware GroIMP entwickelt. Sie erweitert „normale“ 3D-Modellierungsprogramme um die Möglichkeit, sogenannte Wachstumsgrammatiken zu verwenden.

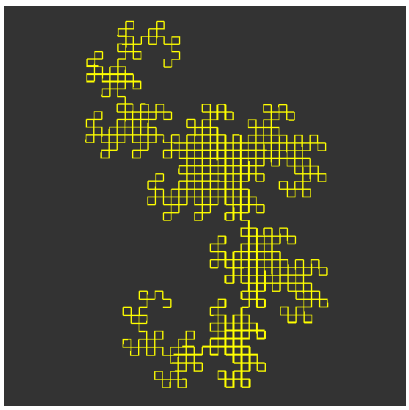

Abbildung 1.1: Drachenkurve erstellt mit GroIMP [1]

In der Natur beobachtetes Verhalten beim Wachstum kann in Form von wachstumsgrammatikalischen Regeln festgehalten werden, aus denen sich mit GroIMP dann Bilder generieren lassen.

Die Drachenkurve zum Beispiel ist ein Fraktal und kann mithilfe simpler Regeln erzeugt werden (Abbildung 1.1).

Obwohl GroIMP hauptsächlich für naturbezogene Projekte verwendet wird, ist die Verwendung keinesfalls darauf beschränkt. Die Wachstumsgrammatiken werden in der hierfür entworfenen eXtended L-system language (XL) implementiert, die auf Java aufbaut. Dadurch können auch alle Sprachfeatures dieser Programmiersprache verwendet werden, wodurch GroIMP auch in der Lage ist das bekannte Spiel „Mensch-Ärgere-Dich-Nicht“ darzustellen und sogar spielbar zu machen. Eine von GroIMP erzeugte 3D-Darstellung dieses Spiels ist in Abbildung 1.2 zu sehen.

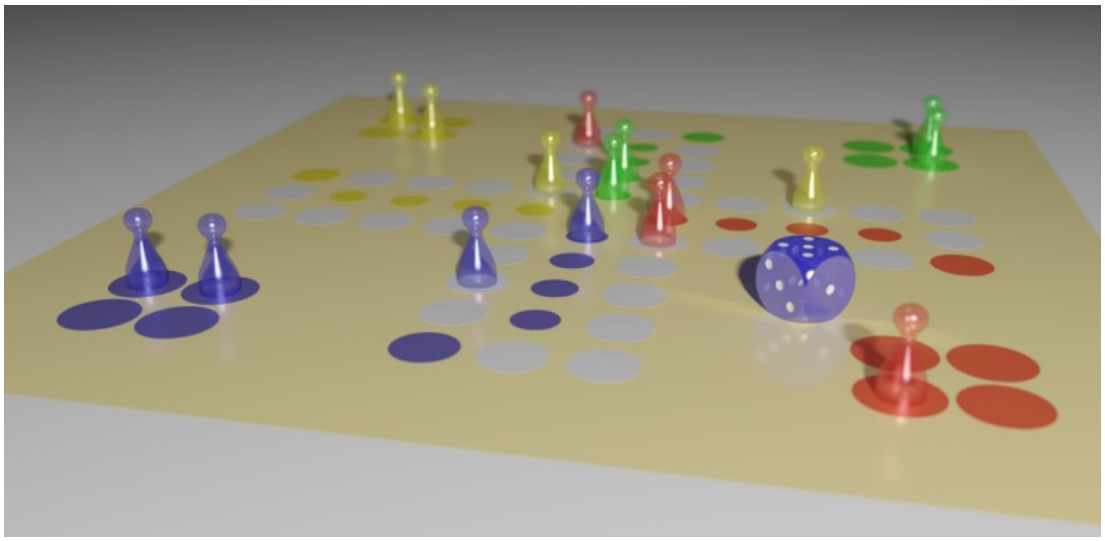

Abbildung 1.2: Mensch-Ärgere-Dich-Nicht erstellt mit GroIMP (spielbar) [1]

1.1 Motivation

Wie ein erster Einblick bereits gezeigt hat (und in Abbildungen im weiteren Verlauf dieser Arbeit noch zeigen wird), ist mit GroIMP eine regelbasierte Simulation und anschließende Erzeugung von Bildern möglich.

Sind die Regeln definiert, können zum Schluss beispielsweise Einzelbilder während des Wachstums einer Pflanze generiert werden. Zu Präsentationszwecken liegt der Wunsch nahe, diese Bildfolge in Form eines Videos zeigen zu können. Genau das ist Gegenstand dieser Arbeit.

1.2 Zielsetzung

Es soll ein Plugin für die GroIMP Software implementiert werden, welche das Programm um eine Video-Export Funktion erweitert.

Der Export soll sowohl mithilfe einer einfach bedienbaren grafischen Benutzeroberfläche, als auch programmgesteuert über XL Code möglich sein. Wichtige Parameter wie die Bildauflösung, Auswahl des Bilderzeugungswerkzeuges (Wahl des Renderers), die Bestimmung der Anzahl an Bildern pro Sekunde und das Videodatei-Format sollen auswählbar sein. Für den eigentlichen Export soll das externe Programm *FFmpeg* verwendet und an GroIMP angebunden werden. Um bei geringer Bildmenge dennoch ein „flüssiges“ Video erzeugen zu können, soll die Möglichkeit geboten werden für die gesammelten Bilder Zwischenbilder zu berechnen.

1.3 Verwandte Arbeiten

Für die Modellierung von Pflanzen existieren neben GroIMP natürlich auch weitere Programme. Das Open-Source Projekt „OpenAlea“ [2] ist eines davon. Es handelt sich hierbei um eine Sammlung von Python Bibliotheken zum computergestützten Entwerfen von Pflanzen. Die Dokumentation dieses Softwarepakets ist etwas lückenhaft und lässt keine Videoexport-Funktionalität vermuten. Obwohl durch die Verwendung von Python eine Plattformunabhängigkeit geschaffen wurde, konnte die Software aufgrund von fehlerhaften Installationsassistenten nicht installiert und getestet werden.

Die Software „LStudio“ [3], welche an der Universität von Calgary entwickelt wird, bietet neben der wachstumsgrammatikgestützten Simulation zusätzlich die Möglichkeit, ein Projekt automatisch in einem Animationsmodus anzuzeigen. Der Export von Bildern in unterschiedliche Dateiformate ist ebenfalls möglich. Die Bilder werden von LStudio im Animationsmodus automatisch gespeichert und einheitlich benannt (falls die Funktion „Recording“ eingeschaltet wird), sodass mit Zuhilfenahme von externen Programmen eine Videoerzeugung ohne Umwege möglich ist. Von Haus aus bietet die Software hierfür jedoch keine Unterstützung.

Zu guter Letzt soll noch ein vergleichender Blick auf das Programm „AMAPStudio“ [4] geworfen werden. Für einen Download ist eine Registrierung erforderlich. Neben einer aufgeräumten Benutzeroberfläche ist es auch mit AMAPStudio möglich, Bilder zu erzeugen und auf der Festplatte zu speichern. Für das Erzeugen der Bilder stehen mehrere Einstellungsmöglichkeiten zur Verfügung, beim Speichern muss jedoch ein Dateiname angegeben werden. Da auch hier eine Videoexport-Funktion nicht vorhanden ist und durch das manuelle Speichern aller Bilder auch die Verwendung eines externen Programms nur mühselig möglich ist (aufgrund der potenziell hohen Anzahl an zu speichernden Bilder), kommt AMAPStudio für einen Videoexport weniger infrage.

Mit der Entwicklung dieses Plugins für GroIMP wird die Videoerzeugung nicht neu implementiert, da von dem externen Programm FFmpeg Gebrauch gemacht wird. Jedoch wird GroIMP dadurch um eine durchaus sinnvolle Funktion erweitert, die in den verglichenen Softwarelösungen derzeit nicht vorhanden ist.

1.4 Aufbau dieser Arbeit

Im zweiten Kapitel werden die Grundlagen geklärt, welche zum Verständnis dieser Arbeit notwendig sind. Dabei wird zunächst über den Begriff des Videos im Allgemeinen gesprochen. Weiterhin werden einige Besonderheiten der Programmiersprache Java aufgegriffen, die im Zusammenhang mit der tatsächlichen Umsetzung von Bedeutung sind. Im Sinne der Wartbarkeit und guten Struktur des zu entwickelnden Plugins wurden einige bekannte Entwurfsmuster verwendet, die knapp skizziert werden. Als Letztes wird in diesem Kapitel über L-Systeme gesprochen, welche die Basis der Wachstumsgrammatiken in GroIMP bilden.

Das dritte Kapitel beschäftigt sich mit dem Entwurf des Plugins. Es werden Designentscheidungen erklärt und diskutiert, die vor der Implementation entschieden wurden. Hier wird die Architektur vorgestellt, auf der das Plugin final aufbauen wird. Außerdem werden die Benutzeroberfläche und ihre Funktionen vorgestellt.

In Kapitel 4 werden nähere Details bei der konkreten Implementation des Plugins gezeigt. Es wird darauf eingegangen, wie das Plugin in die restliche GroIMP Software integriert wird. Auch wird die Kommunikation der Plugin Komponenten untereinander an einigen Stellen genauer erklärt, um das Verständnis zu erleichtern.

Das fünfte Kapitel befasst sich mit einem umfassenden Test des fertiggestellten Plugins in verschiedenen Testszenarien. Es wird geprüft, ob das Plugin seine Funktion sowohl isoliert als auch in Verbindung mit GroIMP erfüllt. Etwaige Probleme und deren Lösungsmöglichkeiten finden außerdem ihren Platz in diesem Kapitel.

Die Arbeit schließt mit einer Zusammenfassung und gibt einen Ausblick auf mögliche Erweiterungen und Verbesserungen des geschaffenen Plugins.

Kapitel 2

Grundlagen

In diesem Kapitel werden einige Themen beleuchtet, die die Grundlage für diese Arbeit bilden.

2.1 Video

Ein Video, wie es jeder kennt, ist ein Zusammenspiel von Bild und Ton. Da sich diese Arbeit mit der Entwicklung eines Video-Export Plugins befasst, ist es sinnvoll über ein Video im Allgemeinen zu sprechen.

> „Videoformate sind eigentlich Datei-Container mit verschiedenen Inhalten, zunächst natürlich Bild und Ton, die getrennt aufbewahrt sind. Die Container können noch weitere Informationen beinhalten wie etwa Untertitel, Menü-Strukturen, Timecodes oder Anmerkungen zur weiteren Bearbeitung.“ [5]

Diese Datei-Container werden häufig auch als Video-Container bezeichnet.

> „Bewegte Bilder nennt man eine Folge von Bildern, die durch Anzeigen in kurzen Zeitabständen mit geeigneter Technik für den Betrachter die Illusion der Bewegung erzeugen. Meist wird der Begriff synonym zu „Filmbildern“ verwendet. Für die menschliche Wahrnehmung genügen bereits etwa 16 bis 18 Bilder pro Sekunde, um die Illusion von fließender Bewegung zu erzeugen, sofern sich die Einzelbilder nur geringfügig voneinander unterscheiden.“ [6]

Grund hierfür: „Der Phi-Effekt, also das Verlangen des Menschen nach Kontinuität und kausalen Zusammenhängen, ist der Grund dafür, dass Menschen die auf dem Film in einzelne Phasen aufgelösten Bewegung im Gehirn zu einer Gesamtbewegung zusammensetzen. [7]“ Unter dem Begriff der Interpolation im Zusammenhang mit Videos oder auch *Motion Interpolation* versteht man die Berechnung von Zwischenbildern zur künstlichen Erhöhung der Bildfrequenz, um ein flüssigeres Video erhalten zu können [8]. In vielen neuen TV-Geräten kommen Algorithmen zur Berechnung

solcher Bilder bereits zum Einsatz. Auch in GroIMP soll diese Technik verwendet werden, um zum Beispiel das Wachstum von Pflanzen in einem erzeugten Video natürlicher aussehen zu lassen. Das Problem besteht nämlich darin, dass viele Simulationen Einzelbilder generieren können, die sich stark voneinander unterscheiden. Die Illusion des fließenden Wachstums ist dadurch erheblich gestört, wenn Bildkomponenten „wie aus dem Nichts auftauchen“. In Abbildung 2.1 wird gezeigt, wie zwischen einem blauen Quadrat und einem roten Kreis interpolierte Zwischenbilder aussehen können, um einen angenehmen Übergang zwischen den sowohl geometrisch als auch farblich unterschiedlichen Figuren zu erzeugen.

Abbildung 2.1: Interpolation zwischen blauem Quadrat und rotem Kreis [9]

FFmpeg [10] ist eine Sammlung von Programmen und Bibliotheken zum Arbeiten mit Videodateien [11]. Zu den wichtigen Funktionen von FFmpeg gehören das Kodieren, Dekodieren, Umwandeln, Muxen und Demuxen (Zusammenfügen oder Aufteilen von Bild- oder Tonsignalen), Streamen und Filtern von Videodateien. Es können aus einzelnen Spuren (Bild oder Ton) neue Videodateien in einer großen Menge von möglichen Formaten erzeugt oder bereits vorhandene Dateien in ihre Bestandteile zerlegt werden. Es besteht weiterhin die Möglichkeit aus einer Serie von Bilder eine Videospur zu generieren, die abschließend in einem Video-Container gespeichert wird. Hierbei können eine Vielzahl von Einstellungen wie die Bilder pro Sekunde, verwendete Codecs und verwendete Video-Container gesteuert werden. Von diesem Prinzip macht das entwickelte Plugin Gebrauch. Das Programm wurde vom FFmpeg-Projekt erstmalig im Jahr 2000 veröffentlicht und steht unter der *GNU Lesser General Public License (LGPL) version 2.1* (oder neuer). Es steht für die Plattformen Linux, Mac OS X, Microsoft Windows, BSD, Solaris und diversen anderen zur Verfügung [10] [11]. Aufgrund der großen Abdeckung an Betriebssystemen sowie der großen Menge an unterstützten Formaten fiel die Wahl des externen Programms zur Erzeugung des Videos auf FFmpeg.

2.2 Java

> „Am Anfang war das Wort. Viel später, am 23. Mai 1995, stellten auf der SunWorld in San Francisco der Chef vom damaligen Science Office von Sun Microsystems, John Gage, und Netscape-Mitbegründer Marc Andreessen die neue Programmiersprache Java und deren Integration in den Webbrowser Netscape vor. Damit begann der Siegeszug einer Sprache, die uns elegante Wege eröffnet, um plattformunabhängig zu programmieren und objektorientiert unsere Gedanken auszudrücken. Die Möglich-

> keiten der Sprache und Bibliotheken sind an sich nichts Neues, aber so gut verpackt, dass Java angenehm und flüssig zu programmieren ist und Java heute zur populärsten Programmiersprache auf unserem Planeten zählt.“ [12, p. 31]

Seitdem wurde Java von Oracle übernommen und ist zum jetzigen Zeitpunkt in der Version 8 Update 144 auf der Internetpräsenz von Java verfügbar [13]. Dieser Abschnitt soll keinesfalls eine Einführung in Java sein, nicht einmal eine sehr kleine. Das würde den Rahmen dieser Arbeit bei weitem sprengen.

Die Java Virtual Machine (JVM) spielt eine entscheidende Rolle für die Plattformunabhängigkeit von Java und wird zur Ausführung einer jeden Java-Applikation benötigt. Sie ist eine abstrakte Rechenmaschine mit einer Menge an Instruktionen und kann auf den Speicher zur Laufzeit zugreifen. Eine JVM führt jedoch keinen Java-Code aus, sondern den daraus erzeugten *Bytecode* (welcher in `.class` Dateien gespeichert wird). Für die Generierung von Bytecode aus einer Java-Datei ist der Java Compiler zuständig. Die verschiedenen Plattformen werden nun dadurch unterstützt, dass für diese eine JVM zur Verfügung steht. Die plattformabhängige virtuelle Maschine kann die abstrakten plattformunabhängigen Anweisungen des Bytecodes in systemabhängige Anweisungen übersetzen und ausführen.

Im Gegensatz zu anderen Programmiersprachen muss sich ein Java Programmierer nicht explizit um das Löschen nicht mehr benötigter Objekte kümmern. Hierfür ist der *Garbage Collector* von Java zuständig, der in einem eigenen, niedrig priorisierten Thread läuft und Objekte im Speicher freiräumt, die nicht länger referenziert werden. Probleme wie das Löschen eines Objektes, welches aber noch referenziert wird (*dangling pointer*), oder das konträre Löschen aller Referenzen eines Objektes, welches aber noch im Speicher existiert (*memory leak*), sind somit automatisch nicht mehr möglich [12, p. 438]. Für diese Vorteile muss mit dem Preis erhöhten Verwaltungsaufwandes bezahlt werden.

Exceptions (im deutschen „Ausnahmen“) sind in Java die Antwort auf die Frage, wie auf unvorhersehbare Probleme reagiert werden kann. In Java kann auf solche Ausnahmen auf zwei Arten reagiert werden: Behandeln oder Weiterreichen. Kann das Problem an einer bestimmten Stelle behandelt werden, wird dies in Java mit einem `try` und `catch` Block realisiert. Liegt die Verantwortung der Problembehandlung jedoch nicht an einer bestimmten Stelle, muss auf den Fehler dennoch reagiert werden. Hierfür kann eine Methode explizit definieren, dass sie Fehler einer bestimmten Art an den Aufrufer weiterleiten (`Exception`s in Java sind hierarchisch erbend aufgebaut und bieten zu gängigen Problemen passende Unterklassen). Das wird mit dem Schlüsselwort `throws` in der Methodendeklaration erreicht [12, p. 533].

Grafische Benutzeroberflächen können in Java unter anderem mit dem *Swing* Framework entwickelt werden. Wichtige Graphical User Interface (GUI) Komponenten wie beispielsweise Schaltflächen, Panels, Beschriftungen und Eingabefelder sind als Klassen des Packages `javax.swing` verfügbar. Swing Objekte können ineinander geschachtelt werden. So ist meist ein Fenster (`JFrame`)

oder Dialog (`JDialog`) das erste Objekt, das erzeugt wird. Diesem werden dann darin liegende Objekte der Reihe nach hinzugefügt. Wie genau das Resultat aussieht, wird über sogenannte `LayoutManager` festgelegt. Java stellt hier einige Varianten zur Verfügung, es können bei Bedarf aber auch eigene Manager implementiert werden. Interaktionen mit der GUI werden mit *Listenern* realisiert, welche den jeweiligen Komponenten hinzugefügt werden. So kann beispielsweise für eine Schaltfläche eine Aktion beim Betätigen dieser definiert werden. Um das Ausführen dieser Aktionen kümmert sich der *AWT-Event-Thread* (auch *Event-Dispatching-Thread* genannt). Diese Aktionen sollten deshalb nicht zu viel Rechenzeit in Anspruch nehmen, da sich sonst zu viele unbearbeitete Ereignisse sammeln und die GUI nicht schnell reagieren kann [12, p. 1016].

2.3 Design Patterns

Um das zu entwickelnde Plugin erweiterbar und wartbar zu implementieren, werden einige bewährte Techniken und Strukturen aus dem Bereich der Programmierung verwendet, die sogenannten *Design patterns* („Entwurfsmuster“ im Deutschen).

> „**[D]esign patterns** [...] describe[s] simple and elegant solutions to specific problems in object-oriented software design.“ [14, p. xi]

Durch die Verwendung dieser Patterns wird versucht die Verbindungen von Softwarekomponenten untereinander soweit wie möglich zu lösen und dadurch eine Struktur zu schaffen, die einfacher zu testen und verstehen ist. Ein Modul (oder Klasse), welches kaum Verbindungen zu anderen Modulen hat, kann recht einfach ausgetauscht werden. Design Patterns, die in die Implementation des Video-Export Plugins eingeflossen sind, werden nun in den folgenden Unterabschnitten kurz erklärt.

2.3.1 Adapter

Ein *Adapter* wird verwendet, um das Interface einer Klasse an ein anderes zu „adaptieren“. So kann ein Objekt, obwohl es prinzipiell die gleiche Funktionsweise bietet, aber eine inkompatible Schnittstelle besitzt, dennoch verwendet werden [14, p. 139].

Die „Adapter“-Klasse implementiert das Zielinterface „Target“ und ruft in dessen implementierter Methode `Request()` die zugehörige Methode des inkompatiblen Objektes (`SpecificRequest()`), des „Adaptees“, auf (Abbildung 2.2).

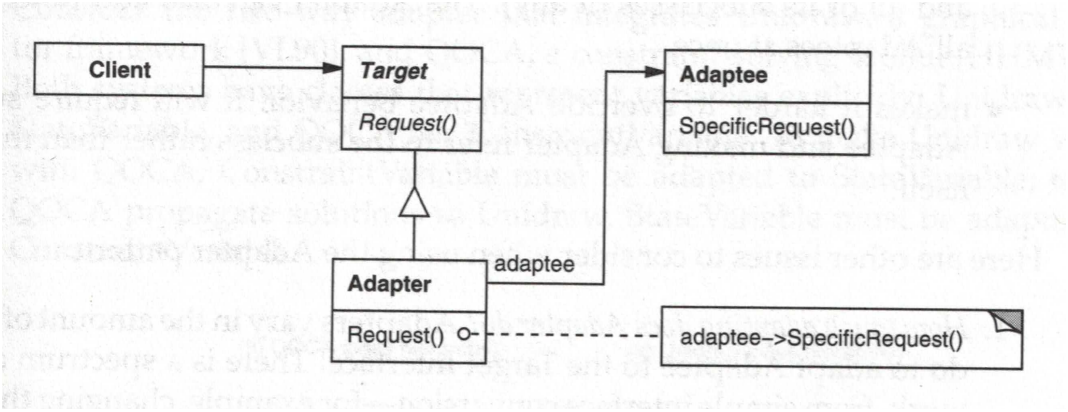

Abbildung 2.2: Das *Adapter* Design Pattern [14, p. 141]

2.3.2 Observer

Das Aufteilen eines Systems in mehrere Komponenten bringt das Problem mit sich, dass die Daten untereinander konsistent sein müssen. Dies soll jedoch nicht dadurch gelöst werden, dass diese Komponenten die Verantwortung hierüber selbst übernehmen und eng mit den anderen betroffenen verbunden sind. Das Zusammenspiel der Softwarebestandteile wäre ansonsten sehr statisch und nicht mehr gut erweiterbar.

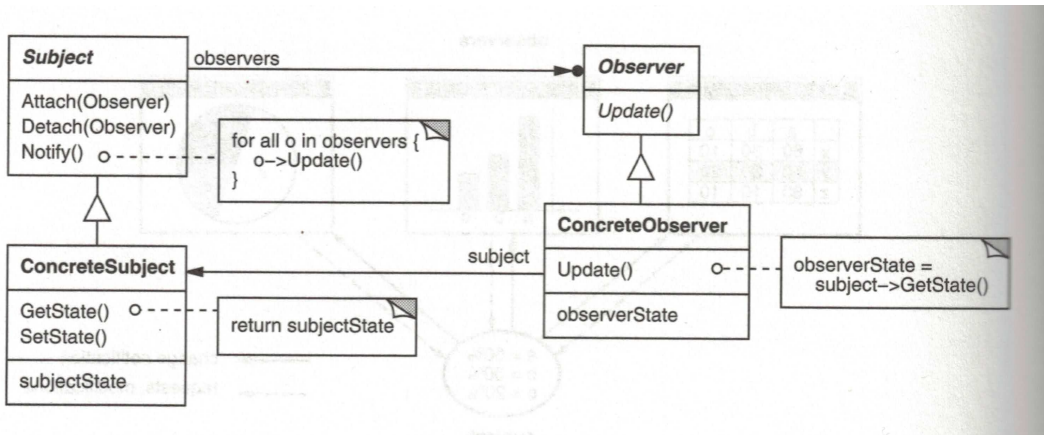

Abbildung 2.3: Das *Observer* Design Pattern [14, p. 294]

Eine Lösung hierfür ist das *Observer* Pattern. Objekte, deren Zustände sich ändern können (Instanzen der Klasse „ConcreteSubject“), informieren einfach die anderen Objekte, welche auf diese Änderungen reagieren sollen („ConcreteObserver“ Objekte). So können in einem Programm beispielsweise mehrere Darstellungen für zentrale Daten vorliegen (etwa in Form von verschiedenen Diagrammen oder Tabellen). Alle Darstellungen melden sich bei den Daten (einem „Subject“) als

„Observer“ an und werden automatisch bei Änderung der Daten über diese informiert. Hierzu muss das Datenobjekt, welches eine abgeleitete Klasse von „Subject“ ist, nach einer Änderung die Methode `Notify()` aufrufen. So ist es recht einfach möglich die Daten an beliebigen Stellen zu ändern und dennoch eine globale Datenkonsistenz zu schaffen. Im klassischen Design Pattern können die darstellenden Klassen den neuen Status des Objektes dann aktiv, beispielsweise mit der Methode `GetState()`, erfragen (Abbildung 2.3) [14, p. 293].

In Java gibt es bereits eine fertige Implementation dieses Patterns mit der Klasse `java.util.Observable` und dem Interface `java.util.Observer`. Die Klasse `Observable` entspricht hierbei der Klasse „Subject“ der Abbildung 2.3, die Klasse `Observer` der gleichnamigen Klasse. Ein Objekt, welches andere über Zustandsänderungen informieren will, erbt von der Klasse `Observable` und ruft bei einer auftretenden Änderung die Methoden `setChanged()` und `notifyObservers()` auf. Erstere legt fest, dass eine Änderung eingetreten ist, die zweite informiert alle beobachtenden Objekte über die Änderung. Hier kann optional durch eine überladene Funktion gleichen Namens ein beliebiges Argument in Form eines `Object`s als Parameter übergeben werden, um die Art der Änderung genauer zu spezifizieren. Beobachter hingegen müssen das Interface `Observer` implementieren und definieren in der darin enthaltenen Methode `update(Observable o, Object arg)`, wie sie auf eine Änderung des Objektes `o` reagieren wollen, wobei die Änderung in Form eines `Object`s im Parameter `arg` übergeben wird (der optionale Parameter der Methode `notifyObservers`) [12, p. 813].

2.3.3 Singleton

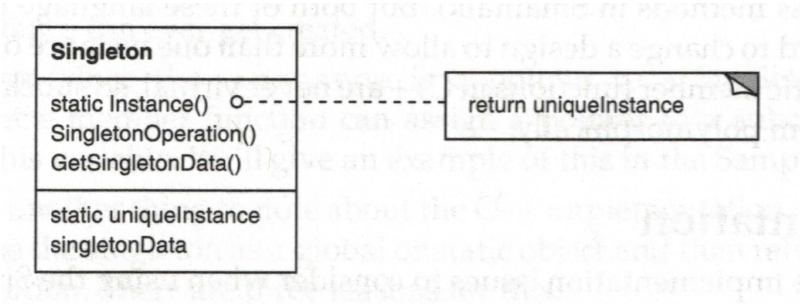

Abbildung 2.4: Das *Singleton* Design Pattern [14, p. 127]

Das *Singleton* Pattern (Abbildung 2.4) stellt sicher, dass von einer festgelegten Klasse maximal eine Instanz erzeugt werden kann, auf die von jeder Stelle im Programm ein einfacher Zugriff möglich ist. Diesen würde man zwar auch mithilfe einer globalen Variable erreichen können, jedoch ist es auf diese Weise nicht möglich sicherzustellen, dass lediglich eine Instanziierung vonstattengeht. Singleton überträgt die Verantwortung hierfür der Klasse selbst über eine hierfür vorgesehene statische Methode (`Instance()`), welche die Erzeugung eigener Objekte unterbindet [14, p. 127].

2.4 Lindenmayer Systeme und XL

Lindenmayer Systeme (kurz *L-Systeme*) wurden 1968 von Aristid Lindenmayer erfunden. Sie sind eine Spezialisierung von Stringersetzungssystemen (auch „Semi-Thue-System“ genannt [15]). Sie werden verwendet, um einen Ausgangsstring durch Ersetzung mithilfe einer Grammatik umzuformen, beispielsweise gegeben durch eine Reihe von Regeln. Gibt man den Symbolen der Zeichenfolge eine spezielle Bedeutung, stellen sich diese L-Systeme als nützliche Werkzeuge in der Simulation von pflanzlichem Wachstum heraus. Für gewöhnlich wird den Symbolen eine geometrische Bedeutung zugewiesen, die sogenannte *Turtle Interpretation*. Hierfür stelle man sich eine imaginäre Schildkröte (engl. „turtle“) vor die sich auf einer ebenen Fläche befindet. Das Symbol `F` könnte der Befehl dafür sein, die Schildkröte einen Schritt vorwärtsgehen zu lassen und dabei eine Spur zu zeichnen. Die Zeichen + und − stehen für eine Rechts- bzw. Linksdrehung der Schildkröte mit einem bestimmten Winkel (in diesem Fall $60°$).
Durch die Symbolfolge `F + + F + + F` läuft die imaginäre Schildkröte den Pfad eines Dreiecks ab. Durch Anwendung der Ersetzungsregel `F` $\rightarrow$ `F − F + + F − F` lässt sich auf diese Weise beispielsweise eine Schneeflocke (die Kochkurve) fraktal artig erzeugen. In jedem Ersetzungsschritt wird die Pfadbeschreibung geändert, woraus sich eine komplexere Figur ergibt. In Abbildung 2.5 ist das initiale Dreieck, die Figur nach einmaliger Anwendung der Regel und nach weiterer dreimaliger Anwendung der Regel dargestellt.

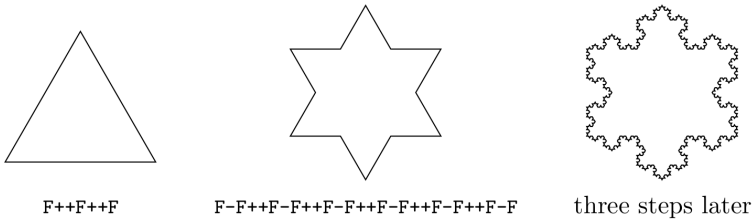

Abbildung 2.5: Die Kochkurve initial, nach einem und drei weiteren Schritten [16] [17]

Durch geeignete Grammatiken und die Einführung weiterer Symbole lassen sich auf diese Weise viele Wachstumsprozesse realitätsnah nachbilden. Pioniere der Computergrafik, darunter auch der polnische Informatiker Przemyslaw Prusinkiewicz [18], erweiterten die Turtle-Befehle um Polygone, Texturen und diverse andere. Dieser Ansatz der Simulation ist jedoch nicht geeignet für globale Interaktionen wie beispielsweise Schatten oder umweltbedingte Einflüsse.

Die relationalen Wachstumsgrammatiken (Relational Growth Grammars (RGG)) sind eine Weiterentwicklung der L-Systeme und weiten das bekannte Ersetzungsverfahren auf Graphen aus. Die grundlegende Datenstruktur ist ein Graph, das heißt eine Menge von Knoten und Kanten, welche diese verbinden. Die Knoten sind hierbei Objekte der darunterliegenden Programmiersprache. Einfache Kanten ergeben dabei eine Relation, komplexere Relationen können bei Bedarf definiert

werden. Es stehen weiterhin verbreitete Datenstrukturen wie beispielsweise „Strings“, „Trees“ und „Multisets“ zur Verfügung, die sich als spezielle Graphen implementieren lassen.

Die an der BTU Cottbus und an der Georg-August Universität Göttingen entwickelte Sprache *XL* ist eine Implementation dieser Wachstumsgrammatiken. Sie wurde auf Basis der Sprache Java entwickelt und wird als Teil der Software GroIMP veröffentlicht, kann aber bei Bedarf auch eigenständig verwendet werden. Innerhalb von XL ist die Einbettung imperativen Javacodes erlaubt. So ist es durch Nutzung externer Bibliotheken einfach möglich, die Fähigkeiten von RGG zu erweitern [19].

Kapitel 3

Entwurf

In diesem Kapitel wird die Architektur des Plugins erklärt sowie wichtige Faktoren und Entscheidungen, die zu der finalen Architektur beigetragen haben.

3.1 Anforderungen an das Plugin

Das zu implementierende Video-Export Plugin soll die 3D-Modellierungssoftware GroIMP um die Funktionalität der Videoerzeugung erweitern.

Es sollen Bilder aus GroIMP gesammelt und anschließend mit dem externen Programm FFmpeg zu einem Video zusammengesetzt werden. Die Bilder aus GroIMP sollen aus verschiedenen Quellen mit einstellbarer Auflösung bezogen werden können. GroIMP stellt als Bildquellen mehrere Renderer zur Verfügung. Es besteht zusätzlich die Möglichkeit die aktuelle Darstellung der Szene unbearbeitet „abzufotografieren“ („Szeneschnappschuss“).
Der aktuelle Stand der gesammelten Bilder soll für den Benutzer erkenntlich sein. Gesammelte Bilder sollen sowohl einzeln als auch in ihrer Gesamtheit gelöscht werden können. Weiterhin ist vorgesehen, dass bereits zur Programmlaufzeit eine Vorschau des künftigen Videos möglich ist.
Ein RGG Projekt in GroIMP bietet häufig Regelblöcke zur Simulation der Szene (beispielsweise dem schrittweisen Wachstum einer Pflanze) an. Das Plugin soll die Möglichkeit bieten diese Simulation automatisch mit einer festlegbaren Anzahl an Wiederholungen durchzuführen, dabei Bilder zu erzeugen und diese anschließend zu speichern.
Vor dem Export soll es weiterhin möglich sein Zwischenbilder zu berechnen, um das Zielvideo zu glätten. Hierfür soll das Plugin Strategien vorsehen, welche problemlos um weitere ergänzt werden können. Dieses Glätten soll sowohl auf die gesamte Menge aller Bilder, als auch auf eine Teilmenge angewendet werden können.
Der eigentliche Export soll die gesammelten Bilder zu einem Video zusammensetzen. Dafür wird das Programm FFmpeg verwendet, welches aber einfach durch andere Programme austauschbar

sein soll. Für den Export sollen Parameter wie die Anzahl an Bildern pro Sekunde, das Dateiformat und natürlich auch der Dateipfad einstellbar sein.
Insbesondere länger andauernde Aufgaben sollen über einen Fortschrittsbalken nachvollzogen werden können, um den Benutzer bei Wartezeiten über die Aktivität auf dem Laufenden zu halten. Es soll des Weiteren möglich sein, Aufgaben zu unterbrechen.

Aufgrund dieser Anforderungen wurde eine grafische Benutzeroberfläche entworfen, die in Abbildung 3.1 zu sehen ist. Auf die Funktionalität der einzelnen Komponenten wird im nächsten Abschnitt eingegangen.

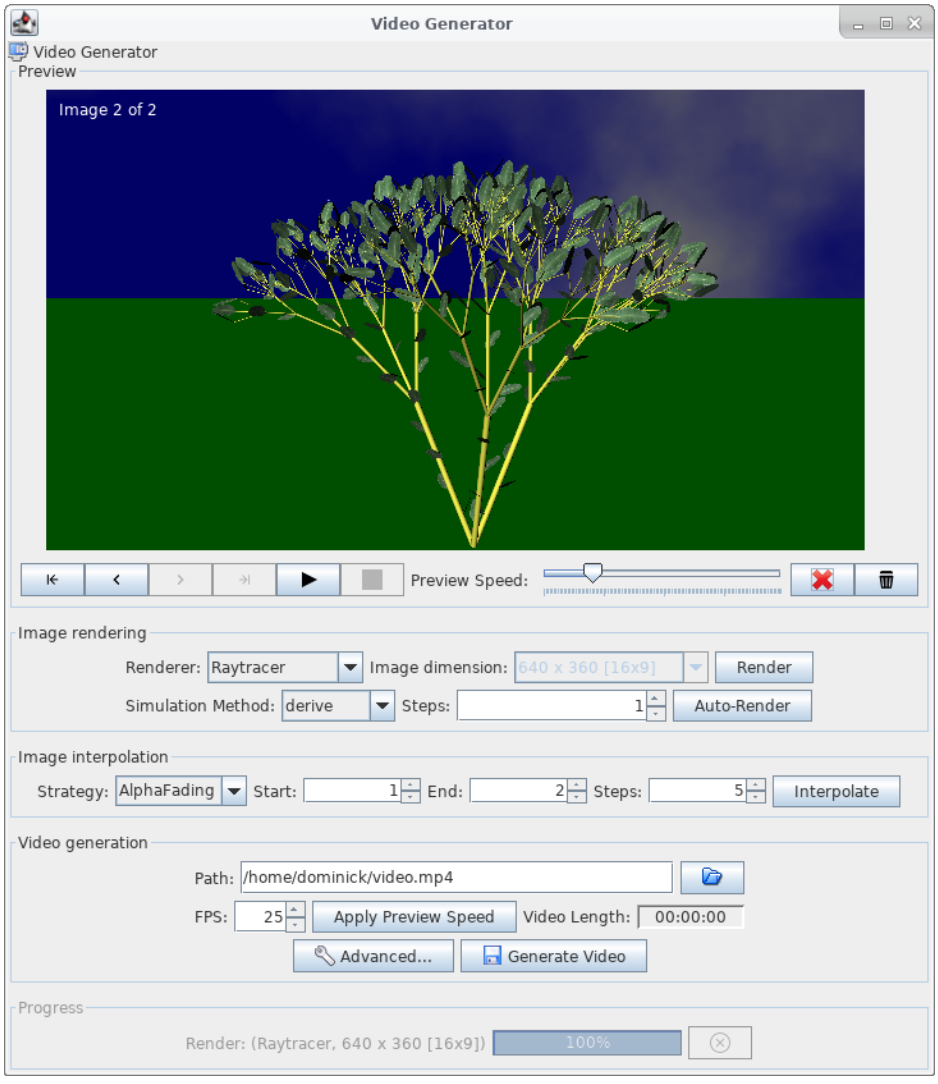

Abbildung 3.1: Die grafische Benutzeroberfläche des Video-Plugins

In Abbildung 3.2 ist dargestellt, wie GroIMP und das Plugin miteinander zusammenhängen. Die verschiedenen Renderer „Raytracer“, „POV-Ray“ und „Flux“ sowie die Möglichkeit der Szeneschnappschüsse werden dem Plugin einheitlich als „Bildquellen“ angeboten. Diese Bildquellen können dann Bilder in einstellbarer Auflösung erzeugen. Die verfügbaren Wachstumsregeln eines RGG-Projektes stehen dem Plugin über „Simulationsmethoden“ zur Verfügung. Die Simulationsmethoden führen stets eine Änderung an der Szene des Projektes durch und erzeugen anschließend mithilfe einer Bildquelle ein Bild. Für die Interpolation existieren im Plugin sogenannte „Interpolationsstrategien“, die innerhalb des Plugins definiert sind und keinen Bezug zu GroIMP haben. Das Plugin erzeugt das Video dann mit Übergabe der Bilder und Angabe der Parameter mit dem externen Progarmm FFmpeg.

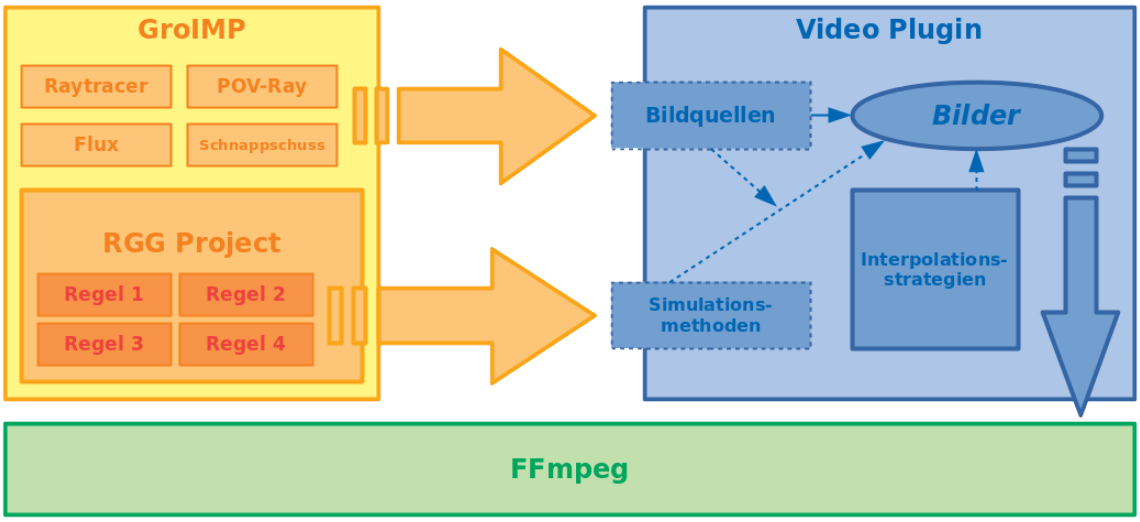

Abbildung 3.2: Eine Übersicht über die Kommunikation von GroIMP, Plugin und FFmpeg

3.2 Benutzeroberfläche

Im Folgenden sollen die einzelnen Komponenten der Benutzeroberfläche näher erklärt werden.

3.2.1 Videovorschau

Mithilfe des Vorschaubereichs „Preview“ (Abbildung 3.3) können die gesammelten Bilder durchlaufen und Bild für Bild angezeigt werden. Oben links innerhalb der Bildvoransicht wird angezeigt, welches Bild gerade betrachtet wird.
Über die Schaltflächen |< und < kann zum Anfang bzw. zum vorherigen Bild der Sequenz navigiert werden. Die Bedeutung der Knöpfe > und >| ist analog zu verstehen.

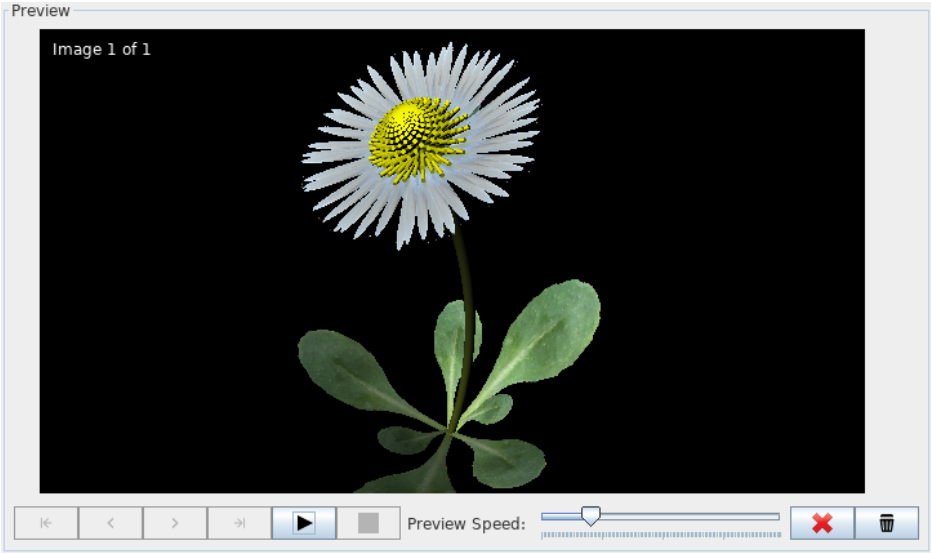

Abbildung 3.3: Das Vorschaufenster für das Beispielprojekt „Daisy Model“

Weiterhin kann die Voransicht auch interaktiv als Video abgespielt werden, sodass bereits vor der Videoerzeugung ein erster Eindruck entstehen kann. Das Video wird mit der Schaltfläche ▶ gestartet, kann auf der gleichen Taste (sobald das Video läuft) ❚❚ angehalten oder alternativ über ■ gestoppt werden. Die Wiedergabegeschwindigkeit kann mit einem Schieberegler auch während der Wiedergabe verändert werden, hier sind Einstellungen von einem bis hin zu 120 Bildern pro Sekunde möglich.
Zu guter Letzt befinden sich ganz rechts Schaltflächen zum Löschen des aktuellen Bildes ✖ sowie rechts daneben zum Löschen aller Bilder der Sequenz 🗑. Bei letzterem muss die Entscheidung in einem Dialogfenster bestätigt werden.

3.2.2 Bilderzeugung

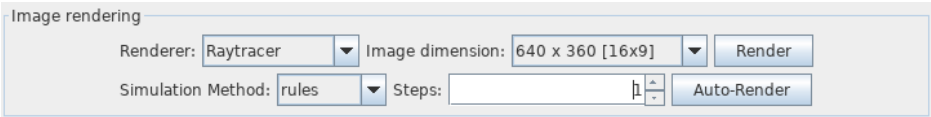

Abbildung 3.4: Die Bilderzeugungsfunktion der grafischen Benutzeroberfläche

Über den Bereich „Image rendering“ (Abbildung 3.4) wird das Erzeugen von Bildern gesteuert. Über die Auswahlbox „Renderer“ wird ausgewählt, aus welcher Quelle die Bilder gewonnen werden sollen. Weiterhin lässt sich die Auflösung des zu erzeugenden Bildes auswählen, diese kann aus einer Auswahlbox mit diversen Standardauflösungen gewählt werden. Ein Klick auf

den Button „Render“ startet die Bilderzeugung. Auf diese Weise kann der Benutzer manuell beliebige Bilder erzeugen und den gesammelten Bildern hinzufügen. Es gilt zu beachten, dass sich die Auflösung von Bildern eines Videos nicht mehr ändern darf. Aus diesem Grund ist das Auswahlmenü der Auflösungen nach dem ersten erzeugten Bild gesperrt und wird erst wieder freigegeben, wenn die Liste der Bilder wieder leer ist (zum Beispiel beim Erzeugen eines neuen Videos oder beim expliziten Leeren der Liste).

Für den Anwendungsfall, dass der Benutzer ein RGG Modell sukzessive simulieren und nach jedem Schritt ein Bild berechnen möchte, ist die „Auto-Render“ Funktion vorgesehen. Über das Auswahlmenü „Simulation Method“ kann die auszuführende Methode ausgewählt und im darauf folgenden Eingabefeld „Steps“ die Anzahl der Wiederholungen definiert werden. Ein Klick auf „Auto-Render“ startet das wiederholte Simulieren des RGG Modells mit anschließender Bildberechnung (die Einstellungen der Einzelbilder erfolgt wie auch beim manuellen Berechnen über die erste Zeile). Diese Funktion ist natürlich nur innerhalb von RGG Projekten verfügbar und wird andernfalls ausgeblendet.

3.2.3 Interpolation

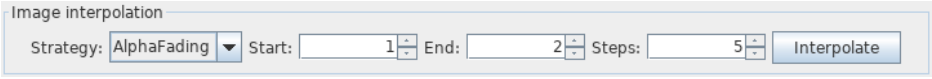

Abbildung 3.5: Die Interpolationsfunktion zur Erzeugung von interpolierten Zwischenbildern

Die Berechnung von interpolierten Zwischenbildern wird über den Bereich „Image interpolation“ (Abbildung 3.5) gesteuert. Dafür kann der Benutzer aus einer Liste eine Strategie („Strategy“) wählen und diese auf einem festgelegten Bereich der Liste der gesammelten Bilder (mit „Start“ und „End“ einstellbar) zwischen je zwei Bildern n mal anwenden, wobei n über das Eingabefeld „Steps“ festgelegt wird. Mit einem Klick auf „Interpolate“ werden die Berechnung gestartet und die Bilder nach erfolgreichem Abschluss an korrekter Stelle in der Liste der bereits vorhandenen Bilder eingefügt.

3.2.4 Videoexport

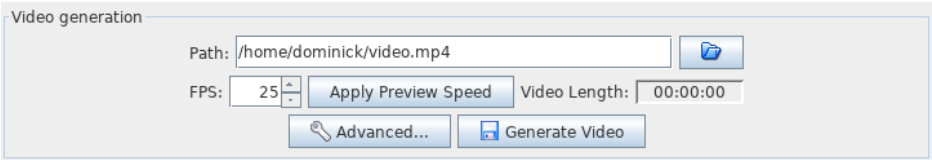

Abbildung 3.6: Der Bereich „Video generation“ zum Erzeugen des Videos

Die eigentliche Videoerzeugung wird im Bereich „Video generation“ (Abbildung 3.6) der Benutzeroberfläche gesteuert. Hierzu wird über das Eingabefeld „Path“ (oder alternativ über einen Dateispeicherdialog mit einem Klick auf) der Speicherort des zu erzeugenden Videos festgelegt. Wird der Speicherort über ein Dialogfenster festgelegt, stehen hier eine Menge von möglichen Dateiformaten zur Verfügung.

In der nächsten Zeile kann die Anzahl der Bilder pro Sekunde im Eingabefeld „FPS“ (Frames Per Second (FPS)) eingetragen werden. Um hier die Auswahl der korrekten Einstellung zu erleichtern ist es möglich, diese Einstellung direkt aus dem Vorschaufenster zu übernehmen, in welchem es wie bereits erwähnt möglich ist, vor der Videoerzeugung einen ersten Eindruck zu gewinnen. So lässt sich die Einstellung recht einfach „testen“ und kann dann mit einem Klick auf „Apply Preview Speed“ für das Endvideo übernommen werden. Es steht weiterhin ein Feld „Video Length“ zur Verfügung, welches stets die aktuelle Videolänge anzeigt und diese bei Änderung der Bildanzahl oder der Geschwindigkeit (in FPS) neu berechnet.

Zum Schluss können über einen Klick auf „Advanced...“ weitere Einstellungen in einem separaten sich öffnenden Fenster vorgenommen werden. Diese beinhalten zurzeit nur die Anzahl der Einzelbilder pro Sekunde des Ausgangsvideos (welche aber keinen Einfluss auf die Anzeigegeschwindigkeit hat). An dieser Stelle könnten weitere Einstellungsmöglichkeiten auf einfache Weise hinzugefügt werden. Ein finales Betätigen der „Generate Video“ Schaltfläche leitet die Videoerzeugung ein.

3.2.5 Fortschritt

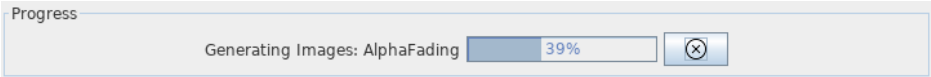

Abbildung 3.7: Der Fortschrittsbalken der grafischen Benutzeroberfläche

Der Bereich „Progress„ (Abbildung 3.7) zeigt den Fortschritt von Operationen an, welche durch die anderen Bedienelemente der Benutzeroberfläche initiiert wurden. Außerdem existiert eine Schaltfläche ⊗, mit der die aktuelle Operation abgebrochen werden kann. Die laufende Operation ist damit beendet, das Ergebnis wird verworfen und die Operation muss erneut gestartet werden, wenn dies erwünscht ist.

3.3 XL

Weiterhin soll die Videoerzeugung auch imperativ über XL-Code steuerbar sein. Hierfür werden einfache Methoden bereitgestellt, die innerhalb von XL-Code verwendet werden können und

eine programmgesteuerte Videoerzeugung ermöglichen. Über diese Methoden ist es möglich die Funktionen des Plugins ohne Verwendung einer grafischen Benutzeroberfläche anzusprechen und das Plugin dennoch in vollem Funktionsumfang nutzen zu können.

Die wenigen Methoden werden im Folgenden kurz erläutert:

clear Leert die Liste der gesammelten Bilder.

size Liefert die Anzahl der bisher gesammelten Bilder zurück (kann in Java Code innerhalb von XL weiterverwendet werden).

renderImage Erzeugt ein Bild unter Angabe der Bildquelle und einer Auflösung.

interpolate Erzeugt interpolierte Zwischenbilder unter Angabe einer Interpolationsstrategie und der Anzahl der zu berechnenden Zwischenbilder.

exportVideo Erzeugt ein Video unter Angabe von Zieldateinamen und der Anzahl der Bilder pro Sekunde.

printImageProviders Schreibt eine Liste aller verfügbaren Bildquellen auf die Kommandozeile. In GroIMP stehen derzeit folgende Bildquellen zur Verfügung: `Raytracer`, `POV-Ray`, `Flux`, `SceneSnapshot`.

printInterpolationStrategies Schreibt eine Liste aller verfügbaren Interpolationsstrategien auf die Kommandozeile. Es steht derzeit nur die Strategie `AlphaFading` zur Verfügung.

Kapitel 4

Implementierung

In diesem Kapitel wird die konkrete Implementierung des Plugins erklärt.

Bei dem Entwurf des Video-Export Plugins wurde versucht ein in sich geschlossenes System zu schaffen. Grund hierfür ist die Tatsache, dass die Erzeugung eines Videos aus Einzelbildern erst einmal nichts mit GroIMP im Speziellen zu tun hat und sich das Plugin zusätzlich auf diese Weise viel besser testen lässt. Natürlich muss das Plugin die Einzelbilder für das Video aus GroIMP beziehen. Für diesen Zweck wurde ein Connector-Interface definiert, welches das Plugin nach außen kapselt. Das Video-Export Plugin ist für zweierlei Nutzungen konzipiert worden. Zum einen soll es möglich sein über eine grafische Oberfläche interaktiv ein Video zu erstellen, zum anderen soll die Videoerzeugung automatisiert innerhalb von XL Code gesteuert werden können. Die eigentliche Erzeugung des Videos funktioniert mit Zuhilfenahme des externen Werkzeuges FFmpeg. Auch für die Videoerzeugung wurde ein separates Interface definiert, um den prinzipiellen Export testen und diesen in Zukunft möglicherweise durch ein anderes externes Tool ersetzen zu können.

4.1 VideoPlugin

Die Klasse `VideoPlugin` ist zwar nicht das Herzstück des Plugins, dafür aber die zentrale Anlaufstelle für korrekte Anbindungen des Plugins, komponentenübergreifende Fehlerbehandlung und die Integration in GroIMP.

Um ein Video-Export Panel (Abschnitt 4.4), das ist die grafische Oberfläche zur interaktiven Videoerzeugung, zu erzeugen, muss in GroIMP ein entsprechender Menüeintrag hinzugefügt werden. Dies geschieht über die `plugin.xml` Datei, welche die Anbindung eines Plugins an GroIMP definiert (detaillierte Informationen zum Aufbau eines Plugins sind dem entsprechenden Bericht zu entnehmen [20]). Die statische Methode zum Erzeugen dieses Panels ist in der zentralen

`VideoPlugin` Klasse zu finden. Weiterhin befinden sich in der Klasse `VideoPlugin` Methoden zum Erzeugen eines Standard-Connectors und eines Standard-Exporters. Zu Testzwecken ist also nur in dieser Klasse eine Änderung notwendig, um dem gesamten Plugin einen anderen Connector beziehungsweise Exporter zur Verfügung zu stellen. Für die Berechnung von Zwischenbildern (Abschnitt 4.7) muss es eine Möglichkeit geben alle vorhandenen Implementationen aufzulisten, um diese zum Beispiel auf der grafischen Oberfläche als Auswahlmenü anzeigen zu können. Hierfür stellt diese Klasse eine Methode bereit. Zu guter Letzt beinhaltet die Klasse die Möglichkeit zentral auf Fehler, genauer `Exception`s, innerhalb des Plugins zu reagieren. Dies ist in diesem Fall nötig, da die Fehler an den unterschiedlichsten Stellen auftreten können und eine annähernd gleiche Behandlung dieser mit dem javaeigenen Exception-Handling (Abschnitt 2.2) nicht möglich ist. Ursache hierfür ist, dass diese `Exception`s zum Teil in Listener-Methoden auftreten und eine Fehlerbehandlung an dieser Stelle nicht sinnvoll ist. Weiter geworfen werden können die Fehler mit dem Schlüsselwort `throw` allerdings auch nicht, da die Methodensignatur von definierten Listenern nicht geändert werden kann. Dennoch war es erwünscht, dass Fehler über die der Nutzer informiert werden soll an diesen weitergegeben werden.

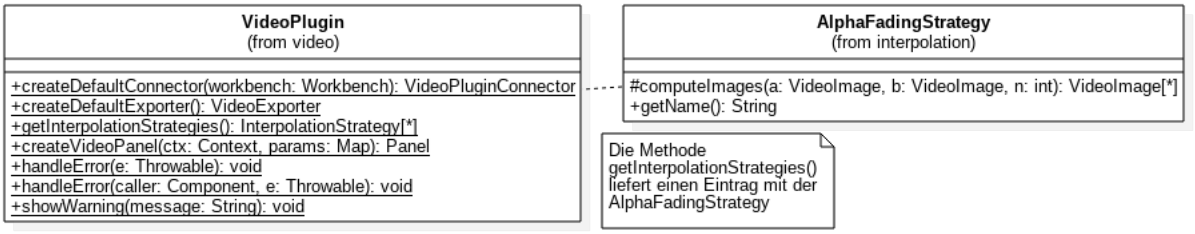

Abbildung 4.1: Die zentrale Klasse `VideoPlugin` [21]

Abbildung 4.1 zeigt die Klasse `VideoPlugin` und ihre Methoden. Die einzig verfügbare `InterpolationStrategy` ist die implementierte `AlphaFadingStrategy` und wird beim Aufruf der Methode `getInterpolationStrategies()` in einer Liste zurückgegeben.

4.2 Connector

Das `VideoPluginConnector` Interface dient der Trennung von Video-Erzeugungs- und Export-Logik und restlicher GroIMP Software. Die Trennung findet aber genauer gesagt gar nicht innerhalb des Interfaces statt.

Das Interface besitzt zwei Methoden, welche Listen von `ImageProvider`n und `SimulationMethod`s zurückliefern. Ein `ImageProvider` dient einzig und allein dazu, verschiedene Bildgenerierungsquellen einheitlich verwenden zu können. Jeder Provider hat

eine Methode zum Erzeugen des Bildes in einer wählbaren Auflösung und muss einen Namen definieren. Eine `SimulationMethod` ist die Abstraktion einer Änderungsmethode des RGG Modells. Sie wird verwendet, um die Funktionalität des automatischen Renderns (automatisches Ändern des Modells mit anschließendem Rendervorgang) von GroIMP-spezifischem Code zu trennen. Wie auch beim `ImageProvider` hat sie einen Namen und definiert zusätzlich dazu eine Methode `execute()`, in der die Modelländerung stattfinden soll. Diese zwei Komponenten genügen um die Kommunikation mit GroIMP zu regeln.

Zu Testzwecken gibt es neben der GroIMP Implementation dieses Interfaces (Abschnitt 4.2.2) auch noch einen `TestConnector` (Abschnitt 4.2.1). So kann die Funktionalität des Plugins auch ohne Kommunikation mit GroIMP unabhängig getestet werden. Abbildung 4.2 zeigt beide Varianten des `VideoPluginConnector`s.

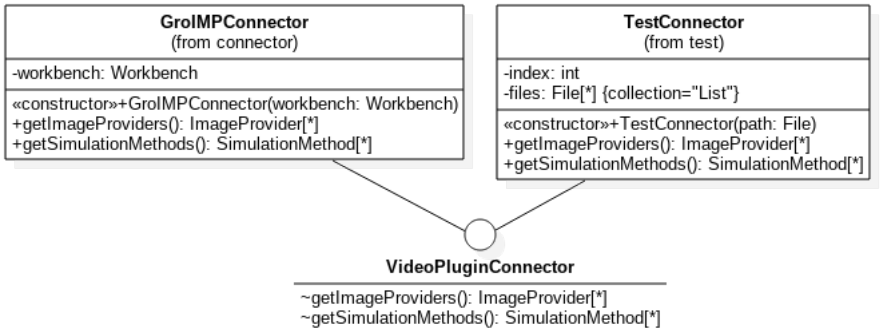

Abbildung 4.2: Die zwei Varianten des `VideoPluginConnector`s [21]

4.2.1 TestConnector

Zu Testzwecken wurde ein einfacher `TestConnector` für die `VideoPluginConnector` Schnittstelle entwickelt. Dieser bekommt beim Erzeugen einen Dateipfad übergeben und liefert Listen mit je einem `ImageProvider` und einer `SimulationMethod`. Die Klasse funktioniert so:

- Beim Erzeugen des `TestConnector`s wird das Verzeichnis gelesen und alle darin enthaltenen Dateien in einer Liste von `File`s gespeichert.
- Der einzige `ImageProvider`, der mit dem Aufruf `getImageProviders()` in einer Liste zurückgegeben wird, lädt das erste Element der zuvor angelegten Dateiliste von der Festplatte und gibt diese Datei als Bild zurück (schlägt fehl, falls die Datei kein Bild ist). Zuvor wird das Bild noch auf die übergebene Auflösung skaliert, was beim Hochskalieren keinen Vorteil bringt. Die Klasse `Image` von Java bietet hierfür die Methode `getScaledInstance`.

- Die einzige `SimulationMethod` (auch wieder in einer Liste) wechselt zum nächsten Eintrag der Liste. Auf diese Weise kann eine Datei nach der anderen aus dem Verzeichnis geladen werden, wenn die „Auto-Render“ Funktion verwendet wird. Zum Schluss beginnt die Liste wieder von vorn.

4.2.2 GroIMPConnector

Sowohl die `ImageProvider` als auch die `SimulationMethod`s benötigen an einigen Stellen Zugriff auf die GroIMP `Workbench`. Aus diesem Grund wird dem Connector beim Erzeugen eine Referenz auf diese mitgegeben.

Die Liste der verfügbaren `ImageProvider` wird aus zwei verschiedenen Implementationen zusammengestellt. GroIMP bietet zur Erzeugung von Bildern verschiedene `de.grogra.imp.Renderer` welche in je ein `RendererAdaptor` Objekt verpackt werden. Der `RendererAdaptor` adaptiert die GroIMP Renderer an die abstrakte `ImageProvider` Klasse. Die Liste der verfügbaren Renderer kann durch Abfragen der Registry (im Bericht näher erläutert [20]) erzeugt werden. In Listing 4.1 ist gezeigt, wie die Liste der GroIMP Renderer aus der `Workbench` extrahiert werden kann.

```
for (Item i = (Item) workbench.getRegistry().getRootRegistry().getItem("/renderers/3d").
    getBranch(); i != null; i = (Item) i.getSuccessor()) {
        // Check if item is an "Expression"
        if (i instanceof Expression) {
                Object r = ((Expression) i).evaluate(workbench, new StringMap());
                // Add renderers only
                if (r instanceof Renderer)
                        list.add(new RendererAdaptor(r));
        }
}
```

Listing 4.1: Erfragen aller `Renderer` mithilfe der GroIMP `Registry` [22]

Dann muss der Liste der verfügbaren `ImageProvider` noch ein weiterer hinzugefügt werden, damit auch die AWT- bzw. OpenGL-Darstellungen der Szene als Bild gespeichert werden können. Dafür wird der Liste ein `SceneSnapshotMaker` (4.6) angehängt.

Um die Liste der Simulationsmethoden zu bekommen, kann das aktuelle `RGG` Objekt abgefragt werden. Dieses kann über die `Workbench` erfragt werden. Wenn aber gerade kein RGG Projekt geöffnet ist, existiert auch das `RGG` Objekt nicht. In diesem Fall wird eine leere Liste zurückgegeben. Die Methode `isTransformationMethod` der Klasse `RGG` musste für diesen Zweck öffentlich

sichtbar gemacht werden. Listing 4.2 zeigt wie eine List der verfügbaren RGG-Methoden erzeugt werden kann.

```
RGG mainRGG = RGG.getMainRGG(workbench);

NType n = mainRGG.getNType();
int declaredMethodCount = n.getDeclaredMethodCount();

for (int i = 0; i < declaredMethodCount; i++) {
        Method m = n.getDeclaredMethod(i);
        if (RGG.isTransformationMethod(m))
                list.add(new RGGMethodAdaptor(workbench, mainRGG, m));
}
```

Listing 4.2: Abfragen der `RGG` Methoden zum Generieren der Liste der Simulationsmethoden [22]

4.3 ImageSequence

Um früher oder später ein Video erzeugen zu können ist es nötig erzeugte Bilder auf irgend eine Art und Weise zu sammeln. Java bietet dafür neben primitiven Array Typen auch diverse `Collection`s, die als generische Datencontainer für diesen Zweck verwendet werden könnten.

Einige Zusatzanforderungen an diese Liste von Bildern haben allerdings dazu geführt, dass hierfür eine eigene Klasse vorgesehen wurde:

- Teile des Plugins sollen einen einfachen Zugriff auf die `ImageSequence` ermöglichen. In der grafischen Benutzeroberfläche soll es hierfür eine grafische Vorschau der Liste geben, die beliebig durch navigiert werden kann und das Löschen einzelner oder aller Bilder gestattet. Es ist sinnvoll, dass diese Vorschauanzeige beim Ändern der Liste aktualisiert wird. Dafür wird konkret das `Observer` Designpattern verwendet (Abschnitt 2.3.2), sodass sich die Vorschau lediglich als Beobachter bei der Liste zu registrieren braucht und dann automatisch über Änderungen informiert wird.
- Bilder sollen nicht einfach blind in die Liste eingefügt werden, sondern es soll stets sichergestellt sein, dass die Liste auflösungstechnisch konsistent ist. Das heißt, dass zu jedem Zeitpunkt jedes Bild in der Liste die gleiche Auflösung hat, andernfalls wäre eine Videoerzeugung nicht möglich.
- Der Zugriff auf die Liste soll nach außen auf lesenden und schreibenden Zugriff aufgeteilt werden. Hierfür sind `View` und `Control` Interfaces definiert. Objekte von implemen-

tierenden Klassen dieser Interfaces werden über die Methoden `view()` und `control()` zurückgegeben.

4.3.1 VideoImage

Ein naiver Ansatz zum Videoexport ist die Bilder zunächst in der `ImageSequence` zu sammeln und beim konkreten Exportieren alle Bilder temporär auf die Festplatte zu schreiben und dann das externe FFmpeg Programm aufzurufen und zum Generieren eines Videos anzuweisen. Das hat zur Folge, dass insbesondere bei einer größeren Anzahl hochauflösender Bilder der Speichervorgang sehr viel Zeit in Anspruch nehmen kann. Diese Zeit ist relativ gesehen zur tatsächlichen Videoerzeugung von FFmpeg besonders lang.

Ein weiteres Problem besteht im „Sammeln“ der Bilder. Die Bilder werden zur Laufzeit in voller Auflösung in einer `ImageSequence` abgelegt, befinden sich also im Speicher, genauer Heap, der Anwendung. Ein Anwendungsfall eines FullHD Videos (Auflösung von 1920 x 1080 Pixel) mit einigen tausend Bildern scheint erst einmal nicht ausgeschlossen, ist auf diese Weise aber keineswegs möglich ohne große Mengen an Random Access Memory (RAM) zur Verfügung zu haben.

Aus diesem Grund wurde die Klasse `VideoImage` eingeführt, als Abstraktion für Bilder eines Videos. Das `VideoImage` bekommt beim Erzeugen ein Bild übergeben, speichert dieses automatisch eindeutig in einem temporären Verzeichnis auf der Festplatte ab und sorgt bei Programmbeendung für automatisches Löschen dieses Bildes. Weiterhin wird automatisch eine skalierte Version des Bildes generiert, welches dann in der grafischen Vorschau angezeigt werden kann. Falls der Zugriff auf das originale Bild gewünscht ist, wird es wieder von der Festplatte geladen. Die Referenz auf das originale Bild wird nach dem Erzeugen des `VideoImage`s sofort verworfen, sodass der Java eigene Garbage Collector (2.2) dieses automatisch entsorgt. Auf diese Weise stehen die Bilder zum Zeitpunkt des Videoexports bereits auf der Festplatte zur Verfügung, wodurch der zusätzliche Speichervorgang entfällt und der Export dadurch weniger Zeit in Anspruch nimmt. Weiterhin ist es so möglich deutlich mehr Bilder zu sammeln, da jetzt nicht mehr der RAM sondern die Festplatte das Limit setzt ,was in gewöhnlichen Desktop Computern weniger ein Problem darstellen sollte.

4.4 VideoPanel

Da die Benutzeroberfläche direkt über den Menüeintrag erzeugt wird und es keine andere darüberstehende Instanz des Plugins gibt verwaltet die Oberfläche, konkreter das darin enthaltene `VideoPanel`, die `ImageSequence`, die zugehörigen `View` und `Control` Objekte und einen `VideoPluginConnector` (4.2). Der Connector wird dem `VideoPanel` beim Erzeugen als Parameter übergeben.

Die Funktionen des Panels sind in einzelne Komponenten in Form von `JPanel`s aufgeteilt und liegen untereinander auf dem Haupt-`VideoPanel`, welches wiederum auch wieder ein Swing `JPanel` ist. Auf diese Weise kann das Video-Plugin auch ohne GroIMP verwendet werden. Für die isolierte Verwendung kann ein einfaches Swing `JFrame` verwendet werden (Listing 4.3).

```
package de.grogra.video.test;

import javax.swing.JFrame;
import de.grogra.video.ui.VideoPanel;

public class TestFrame extends JFrame {
        private static final long serialVersionUID = 1L;

        public TestFrame(VideoPanel videoPanel) {
                add(videoPanel);
                pack();
                setResizable(false);
                setVisible(true);
        }
}
```

Listing 4.3: Ein `JFrame` zum Verwenden des `VideoPanel`s ohne GroIMP

```
package de.grogra.video.ui;

import de.grogra.pf.ui.swing.*;

public class VideoPanelWrapper extends PanelSupport {
        private VideoPanel videoPanel;

        public VideoPanelWrapper(VideoPanel videoPanel) {
                super(new SwingPanel(null));
                this.videoPanel = videoPanel;
                ((SwingPanel) getComponent()).getContentPane().add(videoPanel);
        }

        public void dispose() {
                super.dispose();
                videoPanel.dispose();
        }
}
```

Listing 4.4: Die GroIMP Panel Wrapperklasse des `VideoPanel`s

Für die Verwendung mit GroIMP wird aber ein GroIMP eigenes Panel verwendet, damit dieses innerhalb von GroIMP zum Beispiel „andocken" kann. Dafür wird eine Hilfsklasse `VideoPanelWrapper` verwendet, welche von der Klasse `PanelSupport` erbt und das `VideoPanel` einfach enthält. Die Klasse `PanelSupport` gehört dem GroIMP Plugin `Platform-Swing` an und bildet eine Basisklasse für GroIMP Panel. Die Klasse `PanelSupport` enthält weiterhin eine Methode `dispose()`, welche beim Schließen des Panels aufgerufen wird. Dieser Methodenaufruf wird an das gekapselte `VideoPanel` weitergereicht, um notwendige Aufräumarbeiten erledigen zu können. In Listing 4.4 ist gezeigt, wie das `VideoPanel` in ein `PanelSupport` Panel verpackt wird.

Für diese Benutzeroberfläche soll im Menü von GroIMP ein Eintrag erstellt werden, sodass der Benutzer über diesen die Oberfläche erstellen kann. Dies wird durch einen entsprechenden Registry Eintrag in der `plugin.xml` Datei des Video-Plugins erreicht (wird im Bericht über die Erzeugung eines GroIMP Plugins näher erläutert [20]). Dieser Menüeintrag soll aber nicht immer in GroIMP angezeigt werden, sondern nur dann, wenn tatsächlich ein Projekt geöffnet wird. Diese Einschränkung wurde mithilfe der Existenzbedingung des `/project` Knotens der Registry erreicht (Listing 4.5). Hilfestellungen zur Erstellung der `plugin.xml` Datei bekam ich von einem ehemaligen Entwickler [22].

```
<registry>
  <ref name="ui">
    <ref name="panels">
      <panel name="video">
        <exists name=".available" ref="/project"/>
        <object expr="de.grogra.video.VideoPlugin.createVideoPanel">
          <var name="context"/>
          <vars/>
        </object>
      </panel>
    </ref>
  </ref>
</registry>
```

Listing 4.5: Registry Eintrag zum Erzeugen des VideoPlugin Panels

Dem `VideoPanel` muss beim Erzeugen ein Connector mitgegeben werden, welcher mithilfe der statischen Methode `createDefaultConnector(Workbench)` der zentralen `VideoPlugin` Klasse erzeugt wird. Die `Workbench` für diesen Methodenaufruf bekommt der `VideoPanelWrapper` beim Erzeugen übergeben, was über die statische Methode `createVideoPanel(Context, Map)` initiiert wird. Aus diesem `de.grogra.pf.ui.Context` Objekt kann die `Workbench` extrahiert werden.

Die Bildwiedergabe des Vorschaufensters wird dadurch erreicht, dass in einem eigenen Thread (dem `AnimationThread`) der Reihe nach Bilder weitergeschaltet werden bis das letzte Bild erreicht wurde. Zwischen den Bildern wartet der Thread eine festgelegte Zeitspanne, die durch die eingestellte Geschwindigkeit definiert ist.

4.5 Worker und Jobs

Das Berechnen von interpolierten Zwischenbildern sowie auch das Erzeugen des finalen Videos erfordern Zeit. Der `Worker` wurde konzipiert, um größere Aufgaben zu übernehmen und eine zentrale und insbesondere geordnete Zuständigkeit für Aufgabenabarbeitung zu schaffen. Diese Aufgaben sind in einer abstrakten Klasse `Job` definiert und werden im Worker in einer Warteschlange eingereiht. Jeder Job kann einen Fortschritt haben, der mithilfe des `Observer` Design-Patterns beobachtet werden kann. Um nicht die Rechenzeit anderer Komponenten mit der Abarbeitung der `Job`s zu belasten, hat der Worker einen eigenen `WorkerThread`, der solange Aufträge aus der Warteschlange bearbeitet bis keine mehr verfügbar sind und der sich danach schlafen legt (wartet bis neue Aufträge eintreffen). Hier gilt das First In First Out (FIFO) Prinzip, sodass die Jobs in der korrekten Reihenfolge bearbeitet werden. Wird ein neuer Job hinzugefügt, wird der WorkerThread geweckt, falls er nicht gerade schon am Arbeiten ist. Dieser zusätzliche `WorkerThread` ist notwendig, damit die grafische Oberfläche beispielsweise nicht blockiert, sobald innerhalb des zugehörigen Swing-Threads die Aufgabe ausgeführt wird (2.2). Es wurde entschieden, innerhalb eines Projektes nicht mehrere Worker gleichzeitig zu haben, die parallel Jobs ausführen. Hierfür wurde das Singleton Design-Pattern (2.3.3) verwendet.

Während das Hinzufügen eines neuen Jobs aus der grafischen Oberfläche so schnell wie möglich von statten gehen soll, darf der Methodenaufruf in XL nicht beendet sein, bevor die Aufgabe abgeschlossen ist. Anderenfalls würde XL Codestellen erreichen, obwohl zuvor angefangene Arbeiten nicht abgeschlossen sind, was unter Umständen zu Fehlverhalten aufgrund sogenannter *race conditions* führen würde. Um dies zu verhindern, steht die Methode `waitUntilFinished()` zur Verfügung, die den aufrufenden Prozess blockiert, bis alle Aufgaben abgeschlossen wurden.

Für alle größeren Aufgaben des Video-Plugins sind `Job`s vorgesehen, der Zusammenhang zwischen `Worker`, dem `WorkerThread` und den `Job`s ist in Abbildung 4.3 dargestellt:

1. Berechnen eines Bildes mit beliebigem `ImageProvider` und einer gewählten Auflösung
2. Automatisches Berechnen von Bildern und Änderung des Modells mithilfe einer `SimulationMethod`, einem beliebigen `ImageProvider` und einer gewählten Auflösung
3. Erzeugen von interpolierten Zwischenbildern mit einer bestimmten `InterpolationStrategy`
4. Erzeugen eines Videos mit einem bestimmten `VideoExporter`

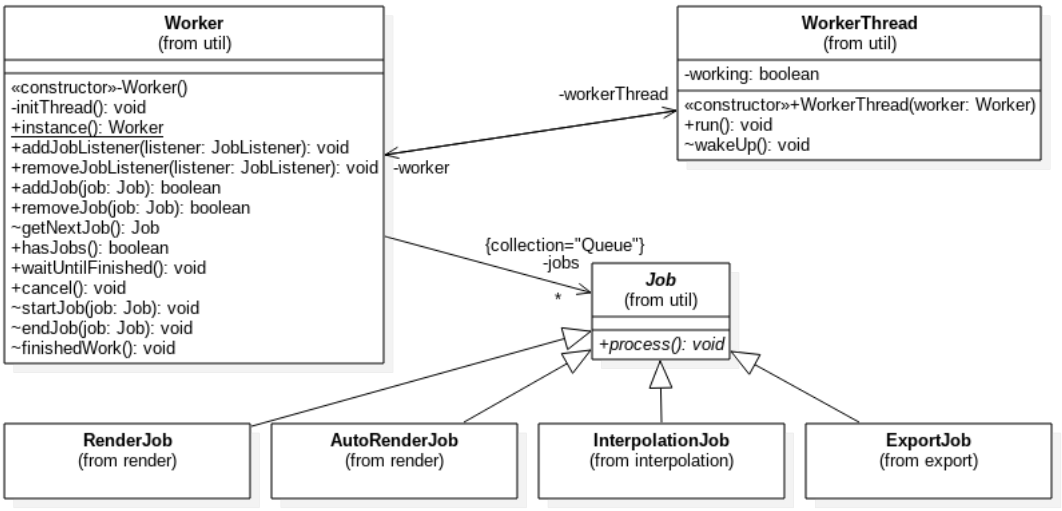

Abbildung 4.3: Der Worker und seine Jobs [21]

Um von Außen auf die Abarbeitung von Jobs reagieren zu können, wurde weiterhin ein `JobListener` definiert, der dem `Worker` hinzugefügt werden kann. In diesem kann auf drei Ereignisse reagiert werden: Ein neuer Job wird bearbeitet, die Bearbeitung des Jobs ist abgeschlossen und alle Aufgaben sind abgeschlossen.

Da über die grafische Oberfläche sowohl Einzelbilder als auch interpolierte Zwischenbilder berechnet werden können und es nicht sinnvoll erscheint beide Operationen vermischt durchzuführen (weil sonst die Reihenfolge der Bilder durcheinander geraten würde), wird die Benutzeroberfläche bei jeder Jobausführung vorübergehend gesperrt. Dies wird dadurch erreicht, dass dem Worker ein `JobListener` hinzugefügt wird, der bei jedem Jobstart das gesamte `VideoPanel` über die im `InteractionPanel` definierte Methode `setInteractionEnabled` sperrt und nach dem Fertigstellen aller Jobs wieder freigibt. Das Interface ist in allen Paneln implementiert und arbeitet dort auf ähnliche aber etwas unterschiedliche Weise, was an dieser Stelle aber nicht näher beleuchtet wird (der interessierte Leser sei hier auf den Quellcode und die dort vorhandene Java-Dokumentation verwiesen). Des Weiteren sollen Jobs während der Bearbeitung im `ProgressPanel` registriert werden, damit dort der Fortschritt mitverfolgt werden kann. Auch hier kommt ein anderer `JobListener` zum Einsatz, der das `ProgressPanel` bei jedem neu gestarteten Job als Beobachter (2.3.2) registriert und dieses nach Fertigstellung des Jobs wieder entfernt. So ist es ohne weitere Mühen möglich jeden Job den der Worker bearbeitet, über das `ProgressPanel` beim Fortschreiten beobachten zu können. Für die Arbeit mit XL wurde eine Klasse `CommandLineProgress` vorgesehen, welche die Anzeige des Fortschritts über die Standardausgabe ermöglicht.

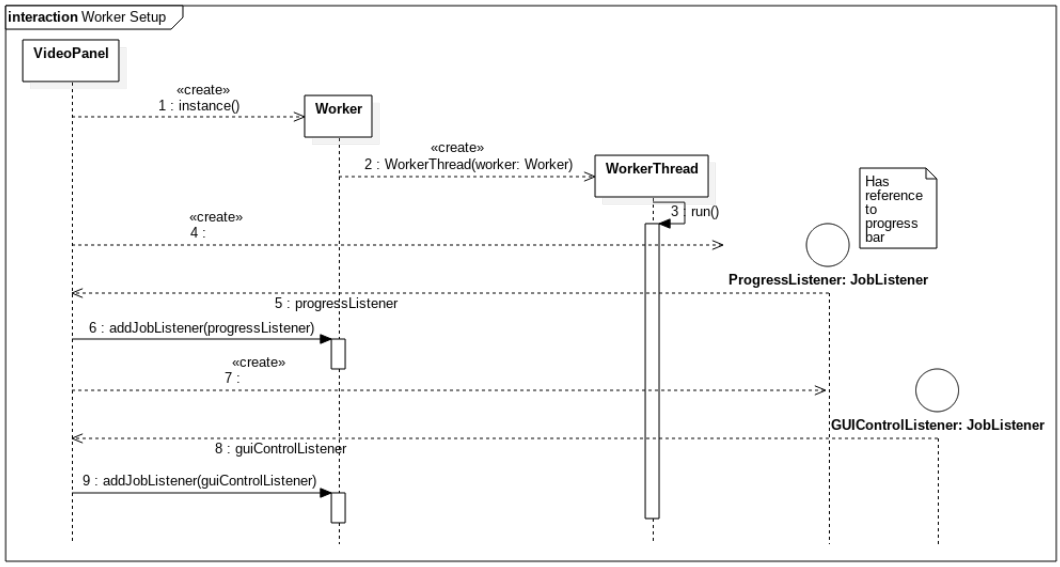

Abbildung 4.4: Initiale Konfiguration des Workers

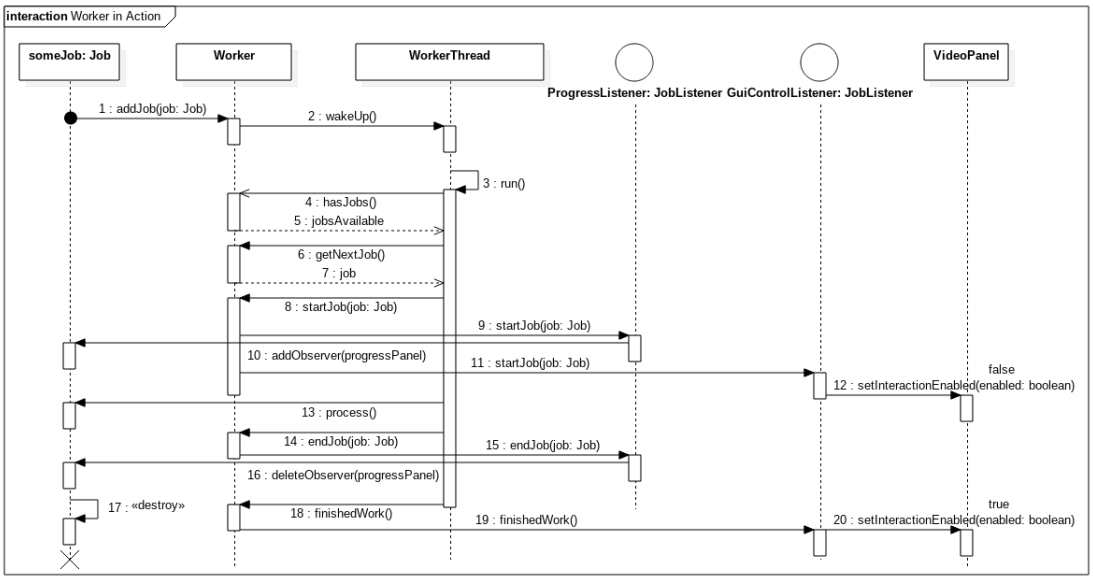

Abbildung 4.5: Kommunikation des Workers mit Benutzeroberfläche bei neuem Job

Die initiale Konfiguration des Workers ist in Abbildung 4.4 dargestellt. Das `VideoPanel` erzeugt eine Instanz des `Workers`, welcher wiederum einen zugehörigen `WorkerThread` erzeugt und startet. Danach erzeugt das `VideoPanel` der Reihe nach einen `ProgressListener` und einen

`GUIControlListener`, welche beim `Worker` registriert werden. Abbildung 4.5 zeigt die Interaktion des Workers mit den Listenern, sobald ein Job bearbeitet wird. Ein ankommender Job (`someJob`) wird dem Worker hinzugefügt. Darauf wird der zugehörige `WorkerThread`, falls notwendig, aufgeweckt. Der Thread bekommt über die Methode `getNextJob()` einen auszuführenden Job vom Worker. Vor der Ausführung des Jobs werden alle Listener im Worker über den Beginn der Aufgabe informiert. Das hat zur Folge, dass der Job bei der Fortschrittsanzeige registriert und damit angezeigt werden kann. Weiterhin wird die grafische Benutzeroberfläche für Eingabe deaktiviert. Sobald die Aufgabe abgeschlossen ist, wird der Listener wieder aus der Fortschrittsanzeige entfernt. Die GUI lässt Eingaben jedoch erst wieder zu, sobald alle Aufgaben abgeschlossen sind.

4.6 Bilder erzeugen

Für die Erzeugung von Bildern sollen zwei Quellen von GroIMP verwendet werden. Zum einen stellt GroIMP verschiedene Implementationen von „Renderern“ bereit, welche auf unterschiedliche Weise aus der aktuellen Szenendarstellung ein Bild in einer beliebigen Auflösung berechnen können. Weiterhin soll es möglich sein, einfach ein Bild der aktuell dargestellten Szene zu gewinnen, wie es im Anzeigebereich der Szene von GroIMP aussieht (AWT- oder OpenGL-Darstellung der Szene). Das erspart zusätzliche Renderarbeit und kann aus Performancegründen in einigen Situationen von Vorteil sein.

Das Erzeugen von Bildern ist in einen `RenderJob` ausgelagert, damit die Benutzeroberfläche bei längeren Berechnungsarbeiten gesperrt bleibt. Da die Berechnung über einen einzigen Methodenaufruf abläuft, kann hier ein Fortschritt leider nicht einfach mitverfolgt werden. GroIMP bearbeitet das Rendern allem Anschein nach in einem eigenen „JobManager“, welcher aus zeitlichen Gründen und dem Mangel an Dokumentation nicht weiter studiert werden konnte.

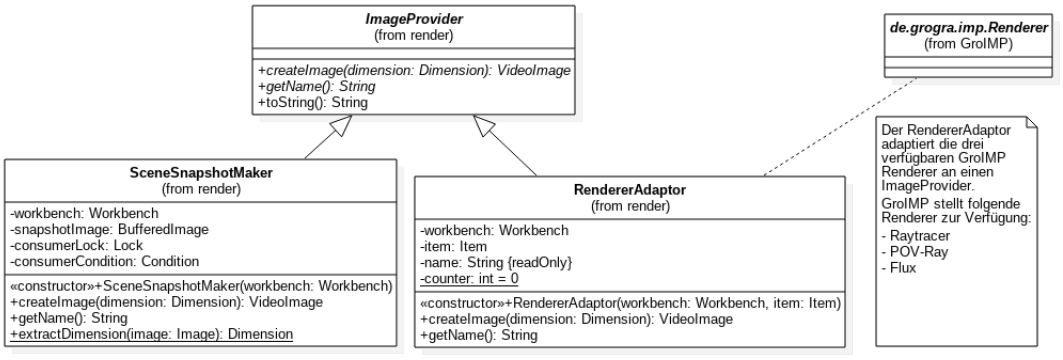

Abbildung 4.6: Die verschiedenen `ImageProvider` des Plugins [21]

Die für das Berechnen der Bilder zuständige Klasse `ImageProvider` hat zwei Implementationen für GroIMP: Den `RendererAdaptor` und den `SceneSnapshotMaker` (Abbildung 4.6).

Auf der einen Seite gibt es die Renderer in GroIMP. Diese Renderer werden von GroIMP zur Verfügung gestellt und erweitern die abstrakte Klasse `de.grogra.imp.Renderer` aus dem GroIMP Plugin `IMP`. Interessant sind hier die Methoden `initialize(View view, int width, int height)` und `computeImage()`. Die Methode `initialize` bekommt ein `de.grogra.imp.View` Objekt (welches die grafische Darstellung der Szene in GroIMP zu repräsentieren scheint) und die Dimensionen in der das Bild erzeugt werden soll. Die Erzeugung wird mithilfe der zweiten Methode gestartet. Für diese GroIMP Renderer gibt es eine `RendererAdaptor` Klasse, die den `ImageProvider` erweitert und die Renderer an den `ImageProvider` adaptiert (2.3.1). Die Dimensionen bekommt der Renderer direkt über den Parameter `Dimension` der Methode `computeImage` übergeben. Um an das korrekte `View` Objekt für die Methode `initialize` heranzukommen, gibt es in der GroIMP Klasse `de.grogra.imp3d.View3D` eine statische Methode `getDefaultView(Workbench)`, welche mithilfe einer `Workbench` Referenz das passende `View` Objekt zurückliefern kann. Die `Workbench` bekommt der `RendererAdaptor` beim Erzeugen, die wiederum vom Connector (4.2) geliefert wird. Dabei handelt es sich um eine Klasse von GroIMP, die das aktuelle Projekt zu repräsentieren scheint.

Auf der anderen Seite sollen Bildschirmaufnahmen der Szene als Bild zurückgeliefert werden. Dafür ist die `SceneSnapshotMaker` Implementation des `ImageProvider`s verantwortlich. Hierfür gibt es ein `ViewComponent` Objekt, welches über das zuvor erwähnte `View` Objekt mithilfe der Methode `getViewComponent` geholt werden kann. `ViewComponent` bietet eine Methode `makeSnapshot`, mit deren Hilfe diese Bildschirmaufnahme erzeugt werden kann. Es ist leider nicht möglich eine Auflösung festzulegen, da das Bild immer genau die Auflösung hat, in der die Szenendarstellung in GroIMP gerade angezeigt wird. Eine Größenänderung der Szenendarstellung hat demnach zur Folge, dass `makeSnapshot` ein Bild in einer anderen Auflösung zurückliefern würde. Damit der `SceneSnapshotMaker` dennoch in verschiedenen Auflösungen Bilder erzeugen kann wird etwas getrickst und das durch `makeSnapshot` gewonnene Bild mit Beibehalten des Seitenverhältnisses auf die gewünschte Auflösung hoch- oder runterskaliert und an den Rändern bei Bedarf leer aufgefüllt.
Das führt leider dazu, dass hochauflösende Bilder nicht wirklich erzeugt werden können und die Qualität sehr niedrig ist, wenn die Szenenansicht von GroIMP klein ist. Beide Implementationen des `ImageProvider`s geben ein in ein `VideoImage` verpacktes Bild zurück.

Um mehrere Bilder automatisiert zu erzeugen und das RGG Modell nach jedem Bild weiter zu simulieren, steht die Funktion des automatischen Renderns zur Verfügung. Wie schon zuvor in Abschnitt 4.5 erwähnt, ist hierfür ein eigener Job vorgesehen, der einfach für eine festgelegte Anzahl an Wiederholungen die Schritte „Simulieren“ und „Rendern des Bildes“ wiederholt. Der Fortschritt aktualisiert sich nach jedem Simulations- bzw. Bildberechnungsschritt, wobei jeder

dieser Schritte als Einheit im Verhältnis zur Gesamtzahl der Schritte betrachtet wird.

Das eigentliche Simulieren wird über `SimulationMethod` Objekte geregelt, wobei jeder „Auto-Render-Job“ genau eines dieser Objekte zum Arbeiten erhalten muss. Die GroIMP Implementation dieses Objektes ist die Klasse `RGGMethodAdapter`, welche die im RGG Modell definierten Methoden an die abstrakte `SimulationMethod` Klasse adaptiert. Die Methoden des RGG Modells sind Instanzen der Klasse `de.grogra.reflect.Method` und bilden eine Art Alternative zum javaeigenen Reflection-Framework. Diese Methoden können jedoch nicht ohne weiteres ausgeführt werden, hierzu ist ein wenig Vorarbeit zu leisten.

Die Klasse `de.grogra.rgg.RGG` ist die Basisklasse aller relationalen Wachstumsgrammatiken innerhalb von GroIMP. Alle Wachstumsprojekte („RGG Projekte“) für GroIMP werden als Objekte dieser Vaterklasse instanziiert. Die Klasse bietet unter anderem eine Unterklasse `Apply`, welche von GroIMP selbst zum Ausführen dieser Methoden verwendet wird. Über die Referenz auf das aktuelle `RGG` Objekt kann eine solche `Apply` Instanz unter Angabe der Methode erzeugt und anschließend mit dem Aufruf `run(workbench, workbench)` gestartet werden [22].

4.7 Berechnung von interpolierten Zwischenbildern

Für ein weicheres Videoerlebnis kann es bei sich stark ändernden Einzelbildern sinnvoll sein, interpolierende Zwischenbilder zu berechnen. Um Algorithmen für diesen Zweck zu kapseln wurde die abstrakte Klasse `InterpolationStrategy` eingeführt.

Innerhalb des Plugins wird nur mit `InterpolationStrategy`-Referenzen gearbeitet. Die nach außen sichtbare Methode heißt `generateImage` und leitet die Berechnung von `n` Zwischenbildern zwischen allen Bildern der Liste `images` ein. Diese Methode kann nicht überschrieben werden, da sie eine relevante Prüfung der Bilddimensionen durchführt. Die tatsächliche Berechnung findet in einer abstrakten Methode `computeImages` statt. Zur Vereinfachung wurde eine weitere abstrakte Klasse `SimpleInterpolationStrategy` eingeführt, welche es ermöglicht eine Strategie zu definieren, für die der Algorithmus nur zwischen je zwei Bildern Zwischenbilder berechnen muss. Der Programmierer der Strategie kann sich somit aussuchen, ob die Berechnung der Zwischenbilder einfach nur zwischen zwei Bildern stattfindet oder die gesamte Bildsequenz in die Berechnung einbezogen werden soll.

Für die Berechnung von Zwischenbildern wird ein einfacher Alpha-Fading Ansatz verwendet. Dafür ist die Klasse `AlphaFadingStrategy` zuständig, welche von der `SimpleInterpolationStrategy` erbt, da es für diesen naiven Ansatz genügt sich nur je zwei aufeinanderfolgende Bilder anzuschauen. Die Methode `computeImages(VideoImage, VideoImage, n)` berechnet für die beiden übergebenen Bilder n Zwischenbilder gemäß der nun definierten „Alpha-Fading“ Strategie. In Abbildung 4.7 ist die Beziehung der Strategieklassen untereinander dargestellt.

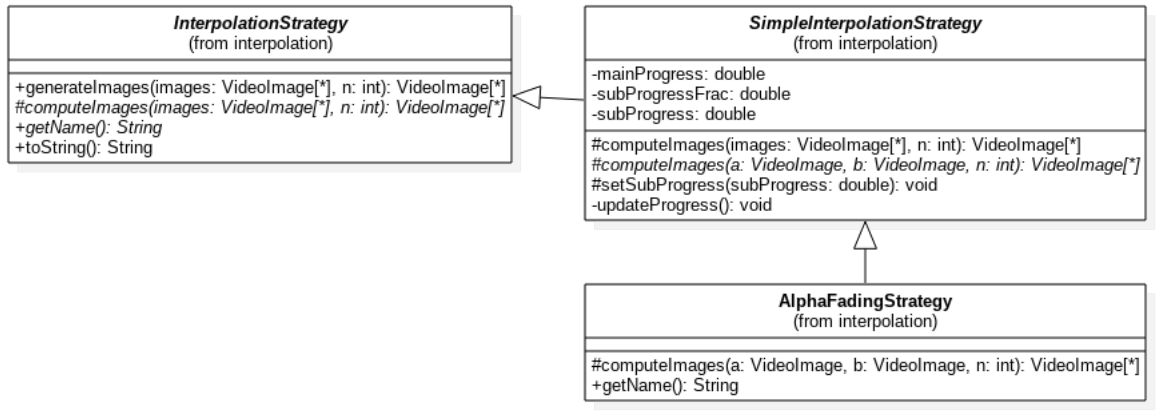

Abbildung 4.7: Klassenstruktur der Interpolationsstrategien [21]

Die Strategie betrachtet für je zwei Bilder Img_A und Img_B jedes Pixel $P_{x,y}$ einzeln und berechnet für jedes Zwischenbild $\text{Img}_i, i \in \{1, \ldots, n\}$ die Übergangspixel $P^i_{x,y}$, wobei n die Anzahl der zu interpolierenden Bilder ist und $P_{x,y} = (r, g, b)$ ein Tupel der Farbwerte rot, grün und blau mit $r, g, b \in [0, 255]$ (0 ist hierbei keine und 255 volle Farbe). Um Zugriff auf die einzelnen Farbwerte zu bekommen seien hierfür Zugriffsfunktionen $P_{x,y}(r)$, $P_{x,y}(g)$ und $P_{x,y}(b)$ definiert. Jedes Img_i der n Bilder wird komplett pixelweise nach der folgenden Formel berechnet:

$$P^i_{x,y} = \big((1-\alpha) \cdot P^A_{x,y}(r) + \alpha \cdot P^B_{x,y}(r), (1-\alpha) \cdot P^A_{x,y}(g) + \alpha \cdot P^B_{x,y}(g), (1-\alpha) \cdot P^A_{x,y}(b) + \alpha \cdot P^B_{x,y}(b)\big)$$

mit

$$\alpha = \frac{i}{n+2} \in (0,1), \quad P^A_{x,y} \text{ und } P^B_{x,y} \text{ Pixel der Bilder } \text{Img}_A \text{ und } \text{Img}_B$$

Hier sei ganz deutlich darauf hingewiesen, dass die Idee dieser Zwischenbildberechnung mit Alpha Fading nicht neu und nicht von mir erfunden wurde. Die Formel jedoch ist selbst aufgestellt nach dem simplen zugrunde liegenden Prinzip.

4.8 Exportieren des Videos

Der Exporter ist eine Abstraktion eines Objektes, welches für das eigentliche Erzeugen eines Videos verantwortlich ist. Durch die Abstraktion ist es möglich das Plugin zu testen, auch ohne wirklich ein Video generieren zu müssen und es bietet für die Zukunft die Möglichkeit Videos auf andere Weise (zum Beispiel mit anderen externen Tools) zu erzeugen.

Ein Exporter hat eine Methode `createVideo(ImageSequenceView, VideoSettings, File)`, mit der über ein `View` Objekt einer zugehörigen `ImageSequence`, festgelegten Einstellungen und einer Zieldatei ein Video erzeugt werden kann. Weiterhin können mögliche Zieldateiformate mit der Methode `getFileFormats()` erfragt werden. Videoeinstellungen werden in einem `VideoSettings` Objekt gespeichert. Vorgesehen sind Einstellungen wie die angezeigten Bilder pro Sekunde sowie auch die Bilder pro Sekunde im Endvideo. Bei erstem Hören scheinen diese zwei Einstellungen identisch zu sein, daher soll dies in einem kurzen Beispiel erläutert werden.

Existieren beispielsweise 40 Bilder und das Video soll eine Länge von 10 Sekunden haben, müssen die Bilder mit 4 Bildern pro Sekunde abgespielt werden. Das wird mit der Einstellung `inputFps` festgelegt. Weiterhin hätte dies zur Folge, dass das Video nur 4 Bilder pro Sekunde anzeigen würde, was in einigen Abspielprogrammen zu Problemen führen kann, da die FPS je nach Videoformat standardisierte Werte haben. Deshalb kann zusätzlich über die Einstellung `outputFps` die Anzahl an Bildern pro Sekunde des resultierenden Videos eingestellt werden. Ist dieser Wert höher als die `inputFps`, werden einige Bilder einfach wiederholt im Video gespeichert. Anderenfalls werden einige Bilder verworfen. Bei gleichen Werten würde sich am Video gar nichts ändern.

Die Idee aus Einzelbildern ein Video zu erzeugen ist keineswegs neu, daher ist es auch wenig überraschend, dass im Internet viele Lösungen für dieses Problem zu finden sind. Native Java Lösungen haben leider keine große Anpassbarkeit geboten. Deshalb fiel die Wahl auf FFmpeg. Hierfür gibt es zwar auch bereits existierende Java Anbindungen, wobei auch hier leider viele veraltet oder fehlerbehaftet sind. Da FFmpeg zu sehr viel mehr in der Lage ist, als dieses Video-Plugin dem Programm abverlangt, fiel die Entscheidung auf eine eigene kleine Anbindung, die das Programm mit entsprechenden Parametern startet. Um dennoch eine Plattformunabhängigkeit zu erreichen werden dem Video-Plugin drei Versionen von FFmpeg für die Betriebssysteme Linux, Windows und Mac mitgeliefert.

Zum Exportieren eines Videos mit FFmpeg gibt es eine Implementation der abstrakten `VideoExporter` Klasse, den `FFmpegWrapper`. Es existiert eine statische Methode, die abhängig vom laufenden Betriebssystem den Dateipfad zu einer der drei beiliegenden kompilierten Versionen von `FFmpeg` zurückliefert. Die Bilder sind in der `ImageSequence` gespeichert, auf welche der Exporter über ein `ImageSequenceView` Objekt ausschließlich lesenden Zugriff erhält. Die Videoerzeugung wird durchgeführt, indem ein `Process` von `FFmpeg` erzeugt und ausgeführt wird. Hierbei wird das Programm mit Parametern gestartet, die abhängig von den übergebenen `VideoSettings` sind. Die Namen der Einzelbilder, die dem Video hinzugefügt werden sollen, können jedoch nicht einfach als Parameter übergeben werden. Hier besteht lediglich die Möglichkeit einen regulären Ausdruck zu definieren, der alle Bilder beschreibt. Dieser Ansatz birgt jedoch Probleme, wenn aus einer Liste von Bildern wieder Einträge gelöscht werden oder nicht alle Dateien in dem Verzeichnis zum Video hinzugefügt werden sollen. Durch die Kapselung der Bilder in `VideoImage`s und die damit verbundene Speicherung aller Bilder einer laufenden GroIMP Applikation in einem einzigen Verzeichnis wurde dieser Ansatz verworfen.

Es gibt jedoch weiterhin die Möglichkeit, die Bilder als Dateistrom per Standardeingabe an den FFmpeg-Prozess zu übergeben. Dieser Ansatz ist unabhängig von den Namen der einzelnen Bilder und daher an dieser Stelle gewählt worden. Hierfür wird der Eingabestrom des Prozesses mit `getOutputStream` geöffnet (es ist der Ausgabestrom in den Prozess!) und jedes Bild nacheinander als `Array` von Bytes in diesen geschrieben. Da dies bei vielen Bildern durchaus wahrnehmbare Zeit in Anspruch nehmen kann, wird der Fortschritt mitverfolgt. Die eigentliche Erzeugung des Videos wird gestartet, sobald der Eingabestrom wieder geschlossen wird, was FFmpeg zum Beginn seiner Arbeit auffordert. Um auch von FFmpeg den Fortschritt mitverfolgen zu können, hängt sich der `FFmpegWrapper` an den Fehlerausgabestrom des Prozesses (hier landen Informationen während der Videoerzeugung) und analysiert die Ausgabe zeilenweise. FFmpeg schreibt während der Bearbeitung an einigen Stellen, welches Einzelbild gerade in die Videodatei geschrieben wird. Daraus lässt sich recht einfach ein prozentualer Fortschritt berechnen. Der Eingabestrom (und Ausgabestrom von FFmpeg) wird automatisch geschlossen, sobald der Prozess beendet ist und damit endet auch die Methode `createVideo`. Das Video ist jetzt vollständig erzeugt.

Neben der FFmpeg-Implementation des Exporters gibt es weiterhin einen `TestExporter`, der sowohl alle Einstellungen als auch die Dateipfade der Einzelbilder des zu erzeugenden Videos auf der Standardausgabe ausgibt. So ist ein Test der korrekten Einbettung des Exporters in das restliche Plugin einfach möglich, ohne die gesamte Logik des `FFmpegWrapper`s korrekt implementiert haben zu müssen.

4.9 Anbindung an XL Code

Die Klasse `XLBinding`, welche zur Steuerung des Plugins aus XL Code heraus eingeführt wurde, besitzt eine statische `ImageSequence` und steuert die Videoerzeugung über statische Methoden. Es ist nicht möglich die Anbindung zeitgleich in zwei Projekten zu verwenden, weil die darin enthaltene `ImageSequenz` nur ein einziges Mal zur gesamten Programmlaufzeit existiert und die Projekte sonst in die gleiche Liste schreiben würden. Jede Methode der `XLBinding` kann etwas am Zustand der Liste ändern oder abfragen oder das Video erzeugen. Außerdem gibt es noch die Methoden `printImageProviders` und `printInterpolationStrategies` welche auf der Standardausgabe verfügbare `ImageProvider` bzw. `InterpolationStrategy`s auflisten. Die Methoden `renderImage` und `interpolate` arbeiten mit dem Namen der Provider oder der Strategien, um die Verwendung aus XL Code heraus einfacher zu gestalten. Hierfür erzeugt die Klasse `XLBinding` bei erstmaliger Verwendung zwei `Map`s, welche zu jedem Namen das passende Objekt speichern. Innerhalb der Anbindung wird ein `Worker` verwendet, um die verschiedenen Aufgaben durchzuführen. Bei jedem Methodenaufruf wird aber bis zur vollständigen Abarbeitung des Workers gewartet, damit die Ausführungsreihenfolge der XL Befehle nicht beeinflusst wird. Da XL nicht über eine plugineigene Oberfläche gestartet wird, kann der Fortschritt von Operationen nicht visualisiert werden. Deshalb wird der Fortschritt über die Kommandozeile ausgegeben.

Kapitel 5

Fallstudie

In diesem Kapitel soll das fertige Plugin in einer Reihe von unterschiedlichen Testszenarien auf Funktionalität geprüft werden. Das Plugin wird zunächst komplett losgelöst von der restlichen GroIMP Software mithilfe des `TestConnector`s und eines `TestFrame`s getestet, um die Funktionalität des Plugins im Allgemeinen beurteilen zu können. Auch wird zu Beginn statt dem echten `FFmpegWrapper` zum Video-Export ein `TestExporter` verwendet.

Nach diesen Tests wird die Erzeugung eines Videos mithilfe von FFmpeg getestet. Zu guter Letzt wird die Zusammenarbeit des Plugins mit GroIMP untersucht. Hier wird zwischen der Nutzung der Benutzeroberfläche und der Nutzung aus XL Code heraus unterschieden.

Getestet wird mit zwei Konfigurationen unter Linux und Windows (Tabellen A.1 und A.2 im Anhang). Mac OS X konnte leider aufgrund fehlender Geräte nicht getestet werden, es sind jedoch keine größeren Hürden zu erwarten.

5.1 Test mit TestFrame, TestConnector und TestExporter

Zuallererst soll das Plugin komplett losgelöst von GroIMP getestet werden. Hierzu wurde ein Verzeichnis `test` innerhalb des Video-Plugin Verzeichnisses erstellt und mit 15 fertig gerenderten Bildern des „Sympodial“ Beispielprojektes befüllt. Der `TestConnector` lädt beim Rendern stets Bilder aus diesem Verzeichnis, beim Simulieren wird zur nächsten Datei übergegangen, wobei zum Schluss wieder von vorne begonnen wird.

Es wird ein `TestFrame` verwendet, um das VideoPlugin in ein Swing `JFrame` einzubetten und die folgende kleine `VideoPanelTest` Klasse, um den Test ohne GroIMP ausführen zu können. Die Testklasse, die für diesen Abschnitt verwendet wurde, `TestFrame`s ist in Listing 5.1 gezeigt.

```
package de.grogra.video.test;

import java.io.File;
import javax.swing.JFrame;
import de.grogra.video.ui.VideoPanel;

public class VideoPanelTest {
        public static void main(String[] args) {
                TestFrame tf = new TestFrame(new VideoPanel(new TestConnector(new File("
                    ../Video/test"))));
                tf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        }
}
```

Listing 5.1: Testprogramm für das VideoPanel ohne GroIMP

5.1.1 Rendern

Zu Beginn ist nur die „Image rendering“ Funktion des Plugins benutzbar, alle anderen Funktionen sind nicht anwählbar. Es stehen nur der Renderer „ImageFromFileLoader“ und die Simulationsmethode „GoToNextFile“ zur Verfügung. Die Dimension kann über das Auswahlmenü „Image dimension“ ausgewählt werden. Das Auswählen der Auflösung steht nur bei leerer Liste zur Verfügung und graut aus, sobald das erste Bild gerendert wurde. Wird die Liste wieder geleert (über die Bildanzeige), steht diese Funktion wieder zur Verfügung. Sowohl das Rendern als auch die Funktion „Auto-Render“ funktionieren problemlos, wenn in dem Verzeichnis nur Bilddateien liegen. Falls sich dort eine Datei anderen Formats befindet, kommt es zu einem Fehler beim Lesen des Bildes. Dieser Fehler kann allerdings vernachlässigt werden, da der `TestConnector` nur zu Demonstrationszwecken entwickelt wurde und nicht für den produktiven Einsatz gedacht ist. Sobald mindestens ein Bild gerendert wurde ist weiterhin die Funktion „Video generation“ verfügbar, ab zwei Bildern kommt die Funktion „Image interpolation“ hinzu. Während des Rendervorgangs kann der Status im unteren „Progress“ Bereich verfolgt werden, alle anderen Komponenten sind währenddessen deaktiviert.

5.1.2 VideoImage

Jedes erzeugte Bild (ob gerendert oder durch Interpolation) wird durch die Klasse `VideoImage` automatisch in einem temporären Verzeichnis abgelegt. Dieses wird beim Schließen des Programms mitsamt aller darin enthaltenen Dateien gelöscht. Erreicht wird dies durch die Methode `deleteOnExit()` der Klasse `java.io.File`. Dies funktioniert sogar, wenn das Programm von

außen unvorhergesehen (beispielsweise mit `kill <PID>` über die Kommandozeile) beendet wird, da hier die virtuelle Maschine vor Beendigung noch Aufräumarbeiten (wie das vorgesehene Löschen dieser Dateien) vornimmt. Lediglich ein Beenden mit `kill -9 <PID>` von außen sorgt dafür, dass die Java Applikation sofort beendet wird und die temporären Daten auf der Festplatte verbleiben. Hierfür gibt es keine Lösung, was aber auch auf die meisten anderen Programme zutrifft, die temporäre Dateien auf der Festplatte ablegen. Außerdem sorgen die meisten Betriebssysteme selbst dafür, unnötige temporäre Daten wieder zu entfernen. Aus diesem Grund ist dieses Problem nicht weiter von Belang.

Werden Bilder gelöscht während das Programm noch läuft und diese im weiteren Verlauf noch verwendet (zum Interpolieren oder zum Export), wird der Benutzer über dieses Problem mit einer Fehlermeldung in Kenntnis gesetzt. Lösen lässt sich dieses Problem dann aber nicht mehr, da das originale Bild nicht mehr im Speicher referenziert ist (und potenziell sogar schon vom Garbage Collector 2.2 entfernt wurde). Würden alle Bilder im Speicher gehalten werden, gäbe es viel zu früh Probleme mit unzureichendem Speicher.

Aus Speichergründen werden die erzeugten Bilder auf der Festplatte abgelegt und lediglich die skalierten Vorschaubilder im Heap aufbewahrt. Beim Testen ist jedoch aufgefallen, dass auch hier die Speicherauslastung höher ausfällt als ursprünglich geplant. Aus dieser Problematik heraus wurde eine Klasse `HeapMonitor` (Anhang Listing B.1) geschrieben, mithilfe derer die Heap Auslastung gemessen werden kann. Die Java Klasse `Runtime` bietet hierzu als nützliche Methoden `totalMemory()` und `freeMemory()` an.

Um das Speicherproblem zu lösen wurden nun auch die Vorschaubilder nicht mehr im Heap gespeichert, sondern bei Anfrage mittels `getPreviewImage()` aus dem Originalbild berechnet, welches dafür von der Festplatte geladen werden muss. Diese beiden Ansätze der Speicherung der Vorschaubilder im Heap und der Berechnung derselbigen aus den Originalbildern wurden einer Messung der Speicherauslastung unterzogen. Getestet wurde mit drei verschiedenen Auflösungen und unterschiedlichen oberen Grenzen für den Heap, welche die virtuelle Maschine über den Parameter `-Xmx` festlegen kann.

In den Abbildungen 5.1 und 5.2 sind die Speicherauslastungen für eine Heapgrenze von 100 Megabyte und die Auflösungen 640×360 und 1920×1080 Pixeln dargestellt. Es sind die Anzahl der erzeugten Bilder gegen die Speicherauslastung in Bytes aufgetragen. In allen Messungen stellen die blauen Balken die tatsächliche Auslastung dar, die rote Linie zeigt `totalSpace()` und die gelbe Linie `maxSpace()`. Beide Messungen brachen nach wenigen Bildern (im Diagramm nachvollziehbar dargestellt) mit einem Speicherfehler ab. In der Auflösung 3840×2160 konnte kein einziges Bild mit der Speichergrenze von 100 MB gerendert werden.

Die Abbildungen 5.3 und 5.4 zeigen, wie durch die Verbesserung der Vorschaubildberechnung sehr viel mehr Bilder erzeugt werden können. Diese Ergebnisse wurden bei Auflösungen von 640×360 und 3840×2160 Pixeln mit nur 100 bzw. 300 MB erzielt. Die Messungen wurden nach

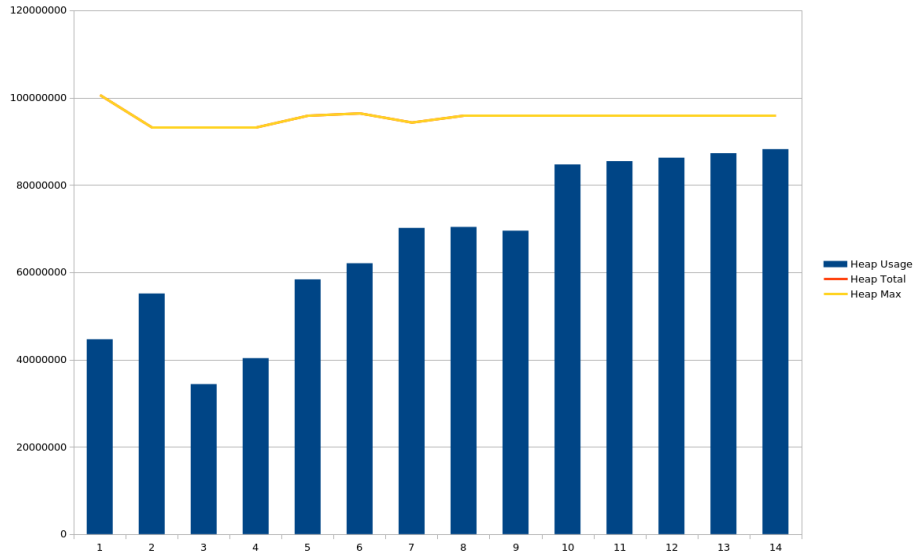

Abbildung 5.1: Heapauslastung bei 640×360 Pixeln und 100 MB Heaplimit (Heap)

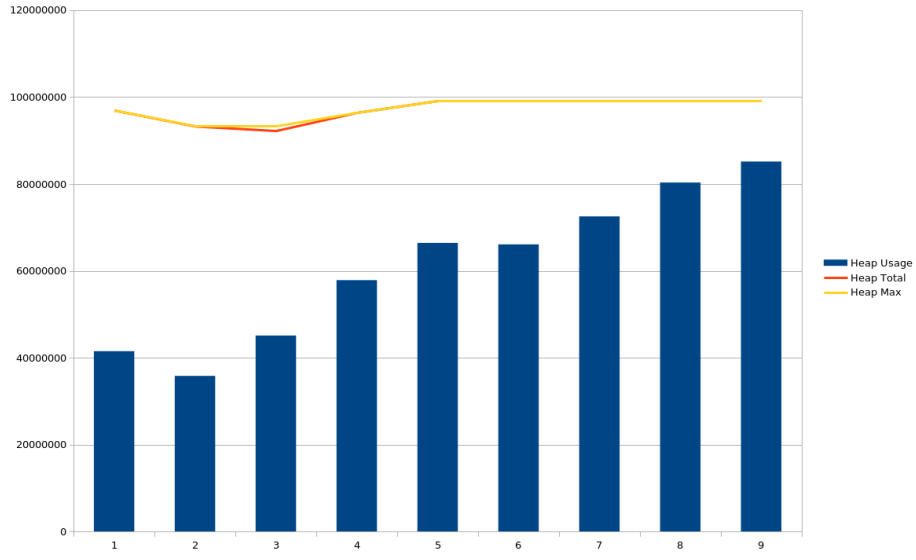

Abbildung 5.2: Heapauslastung bei 1920×1080 Pixeln und 100 MB Heaplimit (Heap)

einiger Zeit (1302 und 341 Bildern) abgebrochen, verursachten aber keine Speicherfehler. Diese Lösung scheint praktikabel zu sein, solange der Speicherplatz des temporären Verzeichnisses nicht zur Neige geht.

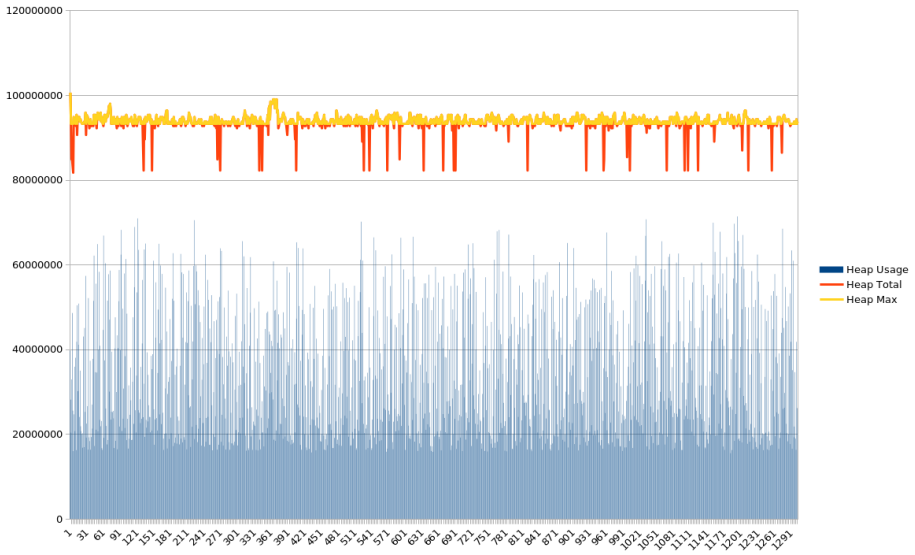

Abbildung 5.3: Heapauslastung bei 640×360 Pixeln und 100 MB Heaplimit (Berechnet)

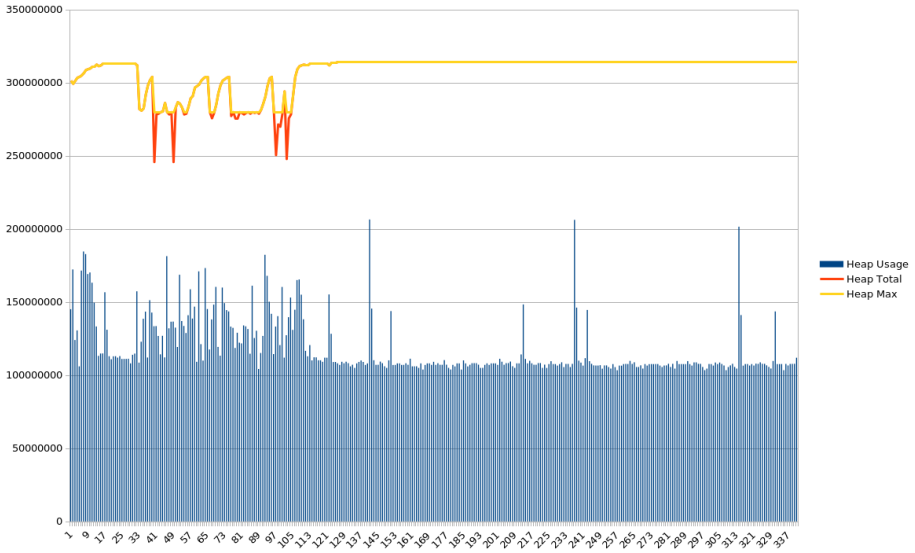

Abbildung 5.4: Heapauslastung bei 3840×2160 Pixeln und 300 MB Heaplimit (Berechnet)

Mithilfe des externen Programms `jvmtop` war insbesondere bei der Vorschaubildberechnung eine Aktivität des Garbage Collectors im Bereich von 5% wahrzunehmen. Weitere Messungen sind im Anhang zu finden.

5.1.3 Bildanzeige mit Videovorschau

Die Vorschaufunktion im oberen „Preview“ Bereich zeigt sowohl zu Beginn als auch beim Leeren der Liste (über das Papierkorbsymbol) `--- no image ---` an. Über die Navigationstasten kann (von links beginnend) zum ersten, vorherigen, nächsten oder letzten Bild navigiert werden. Diese stehen erst ab zwei Bildern zur Verfügung und werden nur eingeblendet, wenn die jeweilige Navigation sinnvoll ist. Die Vorschaufunktion steht bereits ab einem Bild zur Verfügung und kann über die „Play“-Taste gestartet und auf gleichem Knopf wieder pausiert werden. Alternativ steht die Schaltfläche zum Beenden (rechts davon) zur Verfügung (Abbildung 3.3). Wird während einer Vorschau eine Operation gestartet, wird die Vorschau automatisch pausiert. Über die beiden rechten Schaltflächen können das aktuelle oder alle Bilder entfernt werden. Diese Funktionen stehen während einer Vorschau nicht zur Verfügung. Neu ankommende Bilder werden angezeigt, wenn die Liste auch zuvor das neueste (letzte) Bild angezeigt hat.

Durch die Änderung am `VideoImage`, dass Vorschaubilder nun durch Laden des Originalbildes von der Festplatte und darauf folgende Skalierung gewonnen werden (Abschnitt 5.1.2), ist eine deutliche Änderung der Abspielzeit der Vorschau aufgetreten. Aus diesem Grund wurde die Wiedergabegeschwindigkeit mit variablen Auflösungen und FPS Einstellungen genau gemessen und mit der eigentlich korrekten Zeit verglichen, die Tabellen C.1 bis C.8 im Anhang bilden die Messergebnisse ab. Jede Messung wurde fünfmal wiederholt. Zur finalen Auswertung wurde das Minimum dieser Werte verwendet, um den Einfluss systembedingter Verzögerungen zu minimieren. Die Abbildungen 5.5 und 5.6 zeigen den relativen zeitlichen Unterschied zwischen korrekter Abspielzeit und dem Minimum der gemessenen Werte sowohl für die im Heap gespeicherten Vorschaubilder als auch für die berechneten (Verbesserter Ansatz).

In Abbildung 5.5 kann man sehen, dass keine relative Verzögerung über $3,5\%$ eintritt. Grund für die stärkere Abweichung bei 60 bzw. 120 FPS könnte die Tatsache sein, dass die Animation zwischen je zwei Bildern stets $\left\lfloor \frac{1000}{\text{FPS}} \right\rfloor$ wartet, was bei diesen beiden Werten die größten Rundungsfehler aufgrund ganzzahliger Division zur Folge hat. Für die Auflösung 3840×2160 Pixel wurden aufgrund des zuvor erwähnten Speicherproblems keine Messungen durchgeführt.

Nach der Speicherverbesserung durch Berechnung der Vorschaubilder sind in Abbildung 5.6 die daraus resultierenden relativen Verzögerungen zu sehen. Diese wachsen annähernd linear mit der Gesamtzahl der Pixel und ziemlich linear mit der Wiedergabegeschwindigkeit in FPS. Das lässt sich dadurch erklären, dass jedes Mal das gesamte Originalbild von der Festplatte geladen werden muss (was eine Laufzeit abhängig von der Dateigröße / der Pixelmenge hat), welches dann anschließend skaliert wird. Auch das Skalieren kann bei größeren Bilder eine längere

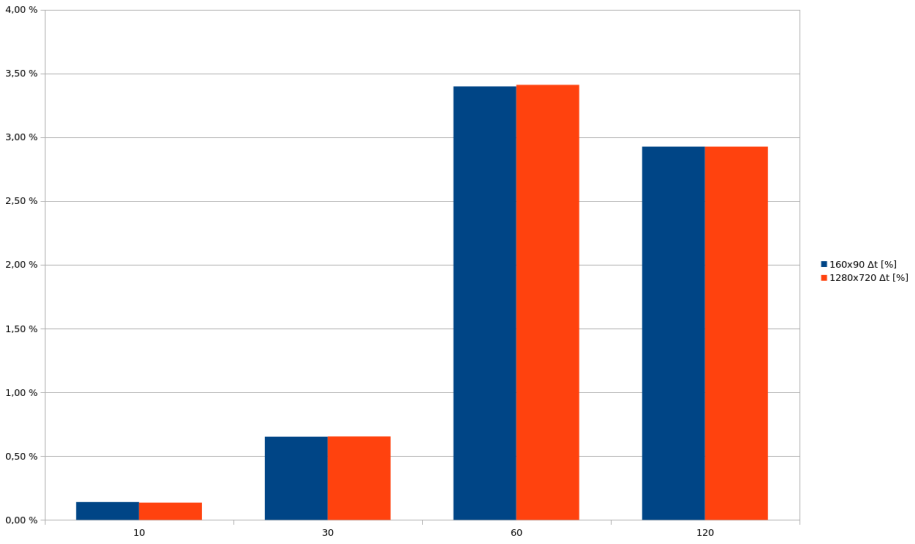

Abbildung 5.5: Relativer Fehler in der Vorschauwiedergabenlänge (Heap)

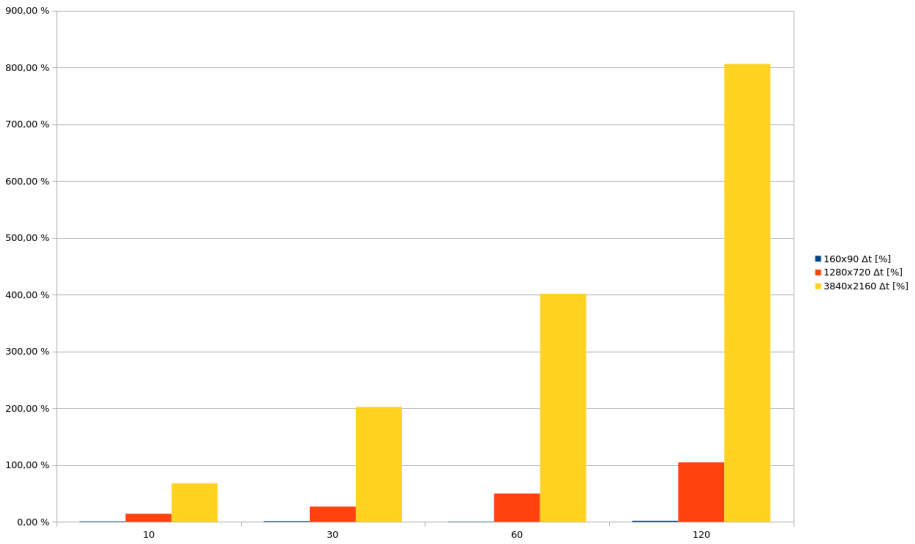

Abbildung 5.6: Relativer Fehler in der Vorschauwiedergabenlänge (Berechnet)

Zeit in Anspruch nehmen. Es wurde jedoch das schnellste zur Verfügung stehende Verfahren `Image.SCALE_FAST` verwendet.

Um dieses Problem zu reduzieren wurde das `VideoImage` soweit geändert, dass nun auch skalierte Vorschaubilder auf der Festplatte abgelegt werden. Hierdurch kann die Ladezeit, insbesondere bei großen Bildern, erheblich reduziert werden. Des Weiteren entfällt der Skalierungsvorgang, da dieser schon beim Erzeugen einmalig abgearbeitet wurde. Die Messungen C.5 und C.6 im Anhang zeigen, dass dieser Ansatz das Speicherproblem nicht nennenswert beeinflusst hat. Die Abbildung 5.7 zeigt, dass die relative Abweichung sichtlich niedriger ist. Sie liegt nun bei einer Auflösung von 3840×2160 Pixeln und 120 FPS bei einem Maximum von $43,58\%$. Dass auch dieser Ansatz sichtlich mehr Verzögerung als die Speicherung im Heap mit sich bringt, ist den Festplattenzugriffen geschuldet.

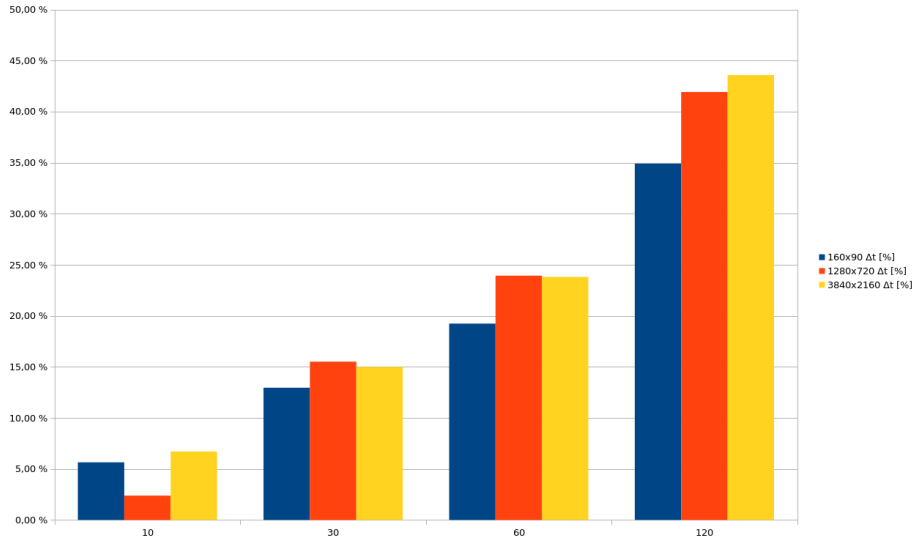

Abbildung 5.7: Relativer Fehler in der Vorschauwiedergabenlänge (Festplatte)

Die von der Festplatte geladenen Bilder lagen im temporären Verzeichnis, welche sich auf einer Solid State Drive (SSD) befanden. Würde hier eine magnetische Festplatte mit damit langsamerer Lesegeschwindigkeit zum Einsatz kommen, wären die erzielten Ergebnisse um einiges schlechter.

5.1.4 Interpolation

Bilder können mit der einzig verfügbaren Strategie „AlphaFading“ interpoliert werden. Hier können Start- und Endbild (1-indiziert) angegeben werden sowie die zu berechnende Anzahl von Bildern zwischen je zwei Bildern. Mit „Start“ und „End“ kann der Bereich der Liste angepasst werden, in dem Bilder berechnet werden sollen. Dieser Bereich umfasst mindestens zwei

Bilder, was beim Ändern der Werte über die Regler sichergestellt wird. Das „End“ Eingabefeld ändert seine obere Grenze automatisch, wenn neue Bilder hinzukommen und die Grenzen zuvor die gesamte Liste umfasst haben. Diese Funktion ist erst ab zwei Bildern verfügbar. Wenn sich im Laufe der Interpolation viele Bilder in der Liste befinden und bearbeitet werden, kann es zu Problemen mit mangelndem Speicher kommen. Die Berechnung der Bilder wird vom WorkerThread direkt übernommen und läuft in einem einzigen Thread ab. Hierdurch können keine Geschwindigkeitsvorteile durch Mehrkernprozessoren ausgeschöpft werden.

5.1.5 Video Export

Der Pfad ist automatisch das Benutzerhomeverzeichnis mit der Datei `video.mp4`. Dieser kann über das Ordnersymbol geändert werden. Mit „FPS“ lassen sich die Einzelbilder der Liste einstellen, die pro Sekunde des Endvideos angezeigt werden sollen (dadurch ist eine Animationsgeschwindigkeit festlegbar). Ein Klick auf „Apply Preview Speed“ übernimmt die Geschwindigkeit des Vorschau-Fensters. „Video Length“ zeigt die voraussichtliche Videolänge an, die beim Export mit den eingestellten FPS resultieren würde. „Advanced...“ öffnet ein kleines Fenster mit weiteren Einstellungen, welche momentan nur die Output-FPS umfassen. Diese steuern, wie viele echte Bilder pro Sekunde im Video existieren sollen. Die FPS sind auf den Bereich von 1 bis 120, die Output-FPS auf den Bereich von 10 bis 120 eingeschränkt (Videos mit weniger als 10 Bildern pro Sekunde zeigten Fehler beim Abspielen). Ein Klick auf „Generate Video“ startet den `TestExporter`, welcher alle Dateien auflistet. Der Exporter ist korrekt an das Plugin angebunden. Diese Funktion ist erst ab einem Bild verfügbar.

5.1.6 Progress

Jede Operation kann über den „Progress“ Bereich mitverfolgt werden. Währenddessen wird die grafische Benutzeroberfläche für Eingaben gesperrt. Während der Bearbeitung eines `Job`s kann dieser über die Schaltfläche mit dem Kreuz rechts neben dem Fortschrittsbalken (Abbildung 3.7) vorzeitig abgebrochen werden.

Sowohl beim Abbrechen als auch beim erfolgreichen Abschluss einer Operation verbleibt der Fortschrittsbalken bei seinem letzten Zustand.

5.2 Test mit TestFrame, TestConnector und FFmpeg

Der Exporter hat per Entwurf keinen Einfluss auf andere grafische Nutzerkomponenten, diese müssen deshalb nicht erneut im Detail getestet werden. Selbst die Exportoperation `ExportJob` ist die gleiche, lediglich mit einem anderen Exporter.

Geändert hat sich aber die Liste der gültigen Dateiformate (zu sehen, wenn man den Zielpfad mit einem Klick auf das Ordnersymbol in einem grafischen Dialog auswählen möchte). Hier gibt es jetzt die Einträge `avi` und `mp4`, welche vom `FFmpegWrapper` geliefert werden. Die Exportfunktion des Plugins funktioniert. Das entstandene Video kann zum Beispiel mit dem VLC Player [23] abgespielt werden. Da sich die Bilder schon auf der Festplatte befinden, entfällt der Speichervorgang. Das Übertragen der Bilder an den FFmpeg Prozess dauert bei größeren Bildern oder größerer Anzahl aber merklich länger. Der eigentliche Export geht hingegen ziemlich schnell. Beim Erzeugen eines Videos mit der Auflösung 3840×2160 trat der Fehler `Broken pipe` mit einem Bild pro Sekunde und Output-FPS von 30 auf. Dieses Problem wurde dadurch gelöst, dass Bilder nun doch nicht mehr an den `FFmpeg` Prozess mittels Pipe übertragen werden, sondern dem externen Programm über einen regulären Ausdruck mitgeteilt werden, der alle Bilder der Sequenz beschreibt. Hier gibt es das Problem, dass zur Laufzeit der Applikationen alle `VideoImage`s in das gleiche temporäre Verzeichnis geschrieben werden, ganz gleich aus welcher `ImageSequence` sie stammen. Auch werden Bilder erst bei Terminierung gelöscht und so könnten zur Laufzeit entfernte Bilder dennoch ihren Weg ins Video finden. Dafür wurde die `ImageSequence` um Methoden `reorder()` und `getRegex()` erweitert. Erstere generiert eine zufällige und kollisionsfreie Zeichenkette und benennt alle Bilder der Sequenz mit einer fortlaufenden neuen Nummer um, wobei die Zeichenkette dem Dateinamen vorangestellt wird. Auf diese Weise ist jeder Dateiname weiterhin eindeutig und dem FFmpeg Prozess können mithilfe eines regulären Ausdrucks (der mit der zweiten Methode generiert wird) alle Bilder der Sequenz mitgeteilt werden. Neben dem gelösten Pipe Problem hat dieser Ansatz auch die beste Laufzeit, da das Umbenennen keine nennbare Verzögerung hat.

5.3 Test mit GroIMP Panel, GroIMPConnector und FFmpeg

Ein weiterer Test des Plugins mit dem `TestConnector` aber über die GroIMP Software wurde hier weggelassen, da keine Änderungen aufgetreten sind.

In diesem Abschnitt sollen nur noch die Probleme und wichtigen Punkte besprochen werden, die sich von den bisherigen Ergebnissen unterscheiden. Zu diesem Zweck wird das Plugin anhand einiger GroIMP Beispielprojekte sowie eines neuen Projektes getestet. Zu finden sind diese über das Menü „File“ → „Show Examples“. Innerhalb eines Projektes wird die Benutzeroberfläche über den Menüeintrag „Panels“ → „Video Generator“ geöffnet.

5.3.1 Beispielprojekt „Koch“

Es stehen die Renderer „Raytracer“, „POV-Ray“, „Flux“ und „SceneSnapshot“ zur Verfügung. Die Auflösung ist wie gewohnt problemlos auswählbar.

Das Rendern jedoch funktioniert nicht und verursacht einen Fehler. Dieser Fehler wird durch

den Methodenaufruf `computeImage` in der eigenen `RendererAdaptor` Klasse hervorgerufen und hat seinen Ursprung in der Klasse `de.grogra.graph.GraphState`. Hier befindet sich eine Methode `get(Graph, ThreadContext)`, welche zu passenden Parametern den korrekten `GraphState` aus einer `java.util.Map` zurückliefern soll. In dem Testfall lieferte diese Methode jedoch `null`. GroIMP scheint an dieser Stelle mit der Information, aus welchem Thread heraus diese Methode aufgerufen wird, zu arbeiten. Die Benutzeroberfläche des Plugins läuft in einem eigenen Swing Thread. Dies könnte die Ursache dafür sein, dass diese Methode nicht den gewünschten `GraphState` liefert.

```
public static GraphState get (Graph graph, ThreadContext tc) {
        WeakReference ref = (WeakReference) graph.getStateMap ().get (tc);

        if (fixFlag) {
                if (ref == null) {
                        for (Object o : graph.getStateMap().values()) {
                                if (o instanceof WeakReference) {
                                        ref = (WeakReference) o;
                                        break;
                                }
                        }
                }
        }

        return (ref != null) ? (GraphState) ref.get () : null;
}

// ************************************************************
// Fix Flag
public static boolean fixFlag = false;
// ************************************************************
```

Listing 5.2: Notlösung mit Schalter in der Klasse `GraphState`

Leider wurde hierfür keine elegante Lösung gefunden und eine Notlösung implementiert. Die Klasse `GraphState` wurde um einen Schalter ergänzt, welcher vor dem Rendern mit dem Plugin ein- und danach wieder ausgeschaltet wird. Wenn der Schalter gesetzt ist, läuft die Methode `get` wie gehabt. Wenn jedoch kein Ergebnis in der `Map` gefunden wurde, wird die `Map` durchlaufen und der erste gefundene `GraphState` zurückgegeben (Listing 5.2). Komplett gelöst ist das Problem jedoch immer noch nicht, da der `RendererAdaptor` nicht die korrekten Bilder zurückliefert. Wird die Szene geändert und danach gerendert, wird noch das alte Bild zurückgegeben. Erst ein wiederholtes Rendern (ohne Änderung der Szene) liefert das korrekte Bild. Die Ursache dieses Problems ist nicht bekannt, jedoch konnte das Problem umgangen werden, indem der

`de.grogra.imp.Renderer` jedes Mal neu aus der GroIMP Registry (genauer aus dem dort liegenden `Item` des Renderers) mithilfe der Funktion `evaluate` bestimmt wird. Dann tritt dieses Problem nicht mehr auf. Es kann jedoch leider nicht erklärt werden, warum dieser Ansatz das Problem löst.

Jetzt ist es möglich, die Render Funktion mit dem „Raytracer“ und beliebiger Auflösung zu nutzen. Der Renderer „POV-Ray“ funktioniert auch weitestgehend, gibt aber in einigen Situationen Warnungen auf der Konsole aus (Listing 5.3). Eine Gesetzmäßigkeit ist hier nicht bekannt, die Warnungen sind auch ohne Verwendung des Plugins aufgetreten.

```
File '/tmp/renderer8156373680209585391.tmp' line 93846: Parse Warning: Possible
 '}' brace missmatch.
File '/tmp/renderer8156373680209585391.tmp' line 93848: Parse Warning: Possible
 '}' brace missmatch.
File '/tmp/renderer8156373680209585391.tmp' line 93850: Parse Warning: Possible
 '}' brace missmatch.
 ...
```

Listing 5.3: Warnungen beim Rendern mit „POV-Ray“

Der Renderer „Flux“ war unter Linux (Anhang Tabelle A.1) nicht funktionsfähig. Dies ist aber auch bei Verwendung ohne das Plugin der Fall gewesen und kann ein treiberseitiges Systemproblem sein.

Der `SceneSnapshotMaker` als letzter Eintrag der Liste hat verschiedene Darstellungsmöglichkeiten. Die Art der Szenedarstellung kann nicht über das Plugin, sondern nur über das Szenefenster von GroIMP im Menü „View“ → „Display“ geändert werden. Beim Erzeugen eines Bildes mit dem `SceneSnapshotMaker` war noch etwas Nacharbeit notwendig, da die Szenedarstellung über „OpenGL“ beim ersten Bild kein Bild lieferte. Das liegt daran, dass die Methode `makeSnapshot` der `ViewComponent` bei einer „AWT“ und „OpenGL“ Darstellung der Szene unterschiedliches Verhalten aufweist. Bei einer „AWT“ Szene wird das übergebene `SceneConsumer` Objekt synchron, bei „OpenGL“ jedoch asynchron abgearbeitet. Das führt dazu, dass zum Zeitpunkt der Bildrückgabe das Bild noch gar nicht vorhanden ist. Dieses Problem konnte mit einer Synchronisierung mit Java Mitteln zunächst gelöst werden. Beim SnapshotMaker gibt es zusätzlich noch die Besonderheit, dass der Benutzer über eine Warnmeldung darauf hingewiesen wird, wenn sich die Größe der Szeneansicht von GroIMP nach erstmaliger Bilderzeugung ändert. Das ist ein gewünschtes Verhalten, da dadurch auch die Skalierung der Szene im Endbild verändert wird und dies unerwünschtes Verhalten im Video zur Folge haben kann. Diese Warnmeldung ist aber lediglich ein Hinweis und hat keine Auswirkung auf das Erzeugen des Bildes. Zusätzlich zu „AWT“ und „OpenGL“ ist noch eine „OpenGL (Proteus)“ Einstellung verfügbar. Diese Darstellung

hat sich jedoch als sehr fehleranfällig auf dem verwendeten Testsystem herausgestellt (sowohl mit als auch ohne Video-Plugin).

Die Funktion „Auto-Render“ ist zwar prinzipiell funktionsfähig, hat aber Synchronisierungsprobleme. Diese zeigen sich dadurch, dass bei einer „Auto-Render-Operation“ mit mehreren Schritten einige Einzelbilder doppelt, andere gar nicht und wieder andere teils nicht komplett auftauchen können. Das liegt wahrscheinlich daran, dass das Ausführen der Simulationsmethode nicht wie gewünscht synchron, sondern GroIMP-intern asynchron abläuft. Ein Weg diesen Vorgang korrekt zu synchronisieren wurde leider nicht gefunden. Der GroIMP `IMPJobManager` könnte hierfür aber eine mögliche Anlaufstelle sein. Besonders auffällig ist dieses Phänomen beim SnapshotMaker, da dieser eine extrem kurze Zeit für die Bilderzeugung beansprucht. Bildfehler einer automatischen Simulation des „Koch“ Projektes mit fünf Schritten werden in Abbildung 5.8 gezeigt. Es ist deutlich erkennbar, dass Teile oder gar ganze Bilder fehlen. Zum Vergleich ist in Abbildung 5.9 eine korrekte Bildabfolge gezeigt.

Abbildung 5.8: Bildfehler von 5-fachem „Auto-Render“ der Koch-Kurve

Abbildung 5.9: Korrekte Bildfolge der Koch-Kurve nach fünf Schritten

Es bleibt noch zu erwähnen, dass die Auflistung der Simulationsmethoden beim Erzeugen eines `VideoPluginConnector`s stattfindet. Das bedeutet in diesem Fall, dass ein nachträgliches Ändern des RGG Projektes (wie zum Beispiel das Hinzufügen einer neuer Methode) keine Auswirkung auf die Pluginoberfläche hat. Um auch diese Methode automatisch ausführbar zu machen, muss das Plugin-Panel geschlossen und neu gestartet werden.

5.3.2 Beispielprojekt „NURBSTree“

Der „POV-Ray“ Renderer liefert in diesem Projekt kein sinnvolles Bild, jedoch auch nicht ohne Verwendung des Plugins. Es ist lediglich ein wenig Himmel mit dortigem Schatten der Pflanze erkennbar (Abbildung 5.10). Die Ursache dafür ist nicht bekannt.

Der Raytracer leistet hier bessere Arbeit, jedoch ist hier nach 11 Bildern ein unnatürliches Wachstum aufgetreten (Abbildung 5.11). Der Fehler ist aber nicht systematisch reproduzierbar und sehr

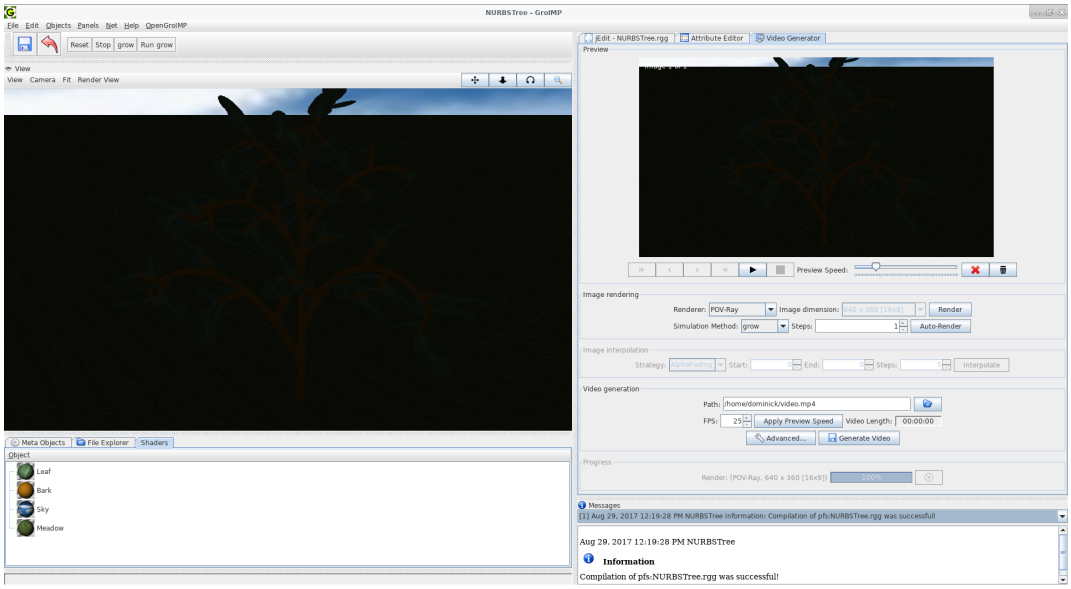

Abbildung 5.10: Bildfehler im Projekt „NURBSTree“ bei Verwendung des „POV-Ray“ Renderers

unwahrscheinlich auf das Video-Plugin zurückzuführen.

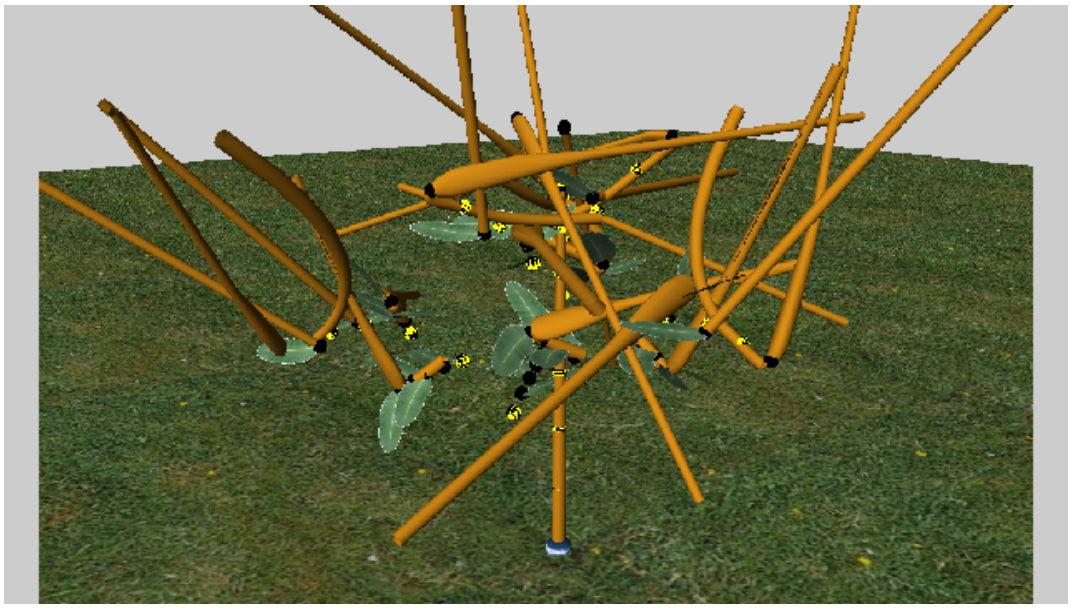

Abbildung 5.11: Bildfehler im Projekt „NURBSTree“ bei Verwendung des „Raytracer“ Renderers

Der SnapshotMaker lieferte als Einziger im ersten Anlauf korrekte Bilder (Abbildung 5.12). Es wurde die „OpenGL“ Darstellung gewählt.

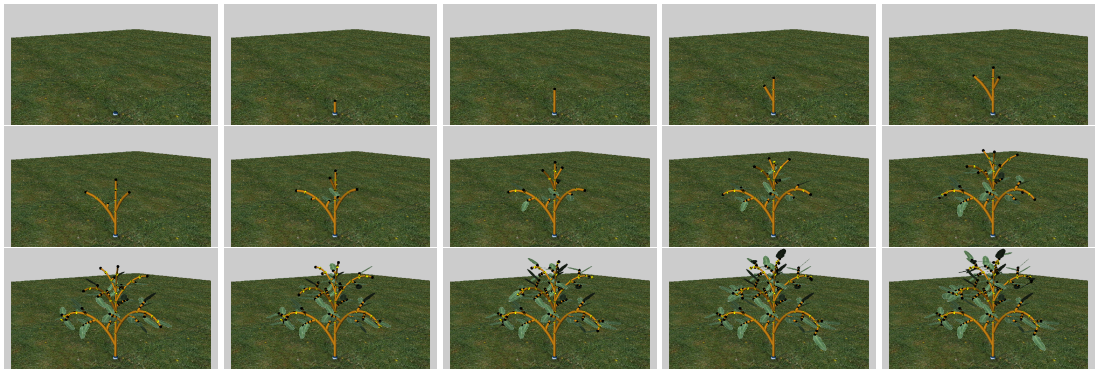

Abbildung 5.12: Wachstum des „NURBSTree“ mit dem Snapshotmaker „OpenGL“

5.3.3 Neues Projekt (nicht RGG)

Zu guter Letzt soll das Video-Plugin noch in einem neu erstellen Nicht-RGG Projekt getestet werden. Hierzu wird über „File“ → „New“ → „Project“ ein neues Projekt gestartet und in diesem das Video-Panel über den entsprechenden Menüeintrag erzeugt. Test halber wurde der Szene ein Ball hinzugefügt.

Die Oberfläche des `VideoPanels` ist bis auf die fehlende „Auto-Render“ Funktion identisch. Diese Funktion fehlt, da in einem normalen Projekt keine RGG-Methoden verfügbar sind. Ansonsten ist die Funktionalität aller anderen Funktionen voll gewährleistet. In Abbildung 5.13 ist ein solches neues Projekt zu sehen.

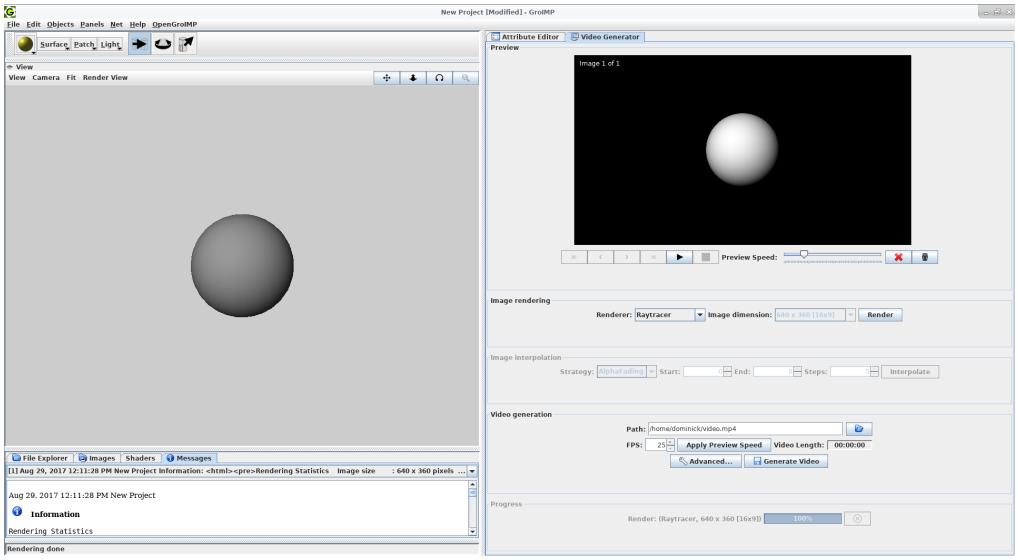

Abbildung 5.13: Das Video-Plugin in Verbindung mit einem neuen GroIMP Projekt

5.4 Test der XL Anbindung

Der Test der XL Anbindung kann nicht ohne GroIMP getestet werden, da XL ein Bestandteil dieser Software ist. Ein Test mit dem `TestConnector` wäre wenig sinnvoll, weil dessen Funktionalität bereits gezeigt wurde.

Es wird an ein paar Beispielprojekten gezeigt, wie das Plugin aus XL Code heraus verwendet werden kann.

5.4.1 Beispielprojekt „Koch“

Zuallererst wird wieder das Projekt „Koch“ betrachtet, da es sich aufgrund der geringen Komplexität gut eignet.

```
import static de.grogra.video.XLBinding.*;

public static final int RENDER_WIDTH = 1280;
public static final int RENDER_HEIGHT = 720;
public static final String RENDERER = "Raytracer";
public static final String INTERPOLATOR = "AlphaFading";
//public static final int INTERPOLATION_BEGIN = 1;
//public static final int INTERPOLATION_END = 5;
public static final int INTERPOLATION_STEPS = 10;
public static final String FILE_NAME = "/home/dominick/xl_test.mp4";
public static final int INPUT_FPS = 10;
public static final int OUTPUT_FPS = 25;

public void rules() {
        [
                Axiom ==> F(10) RU(120) F(10) RU(120) F(10);
                F(x) ==> F(x/3) RU(-60) F(x/3) RU(120) F(x/3) RU(-60) F(x/3);
        ]
        renderImage(RENDERER, RENDER_WIDTH, RENDER_HEIGHT);
}

protected void init() {
        super.init();
        clear();
}

public void makeVideo() {
        // interpolate(INTERPOLATOR, INTERPOLATION_STEPS, INTERPOLATION_BEGIN,
            INTERPOLATION_END);
        interpolate(INTERPOLATOR, INTERPOLATION_STEPS);
```

```
        exportVideo(FILE_NAME, INPUT_FPS, OUTPUT_FPS);
}
```

Listing 5.4: Videoerzeugung über XL Code im Projekt „Koch“

Ganz oben sollte ein statischer Import der Video-Plugin Methoden der XL Anbindung erfolgen, um deren Verwendung zu erleichtern. Alternativ kann dies auch weggelassen werden, dann müssen jedoch die Methoden der Anbindung (`clear`, `renderImage`, `interpolate` und `exportVideo`) stets mit der vollen Paketbezeichnung angegeben werden.

Zur Übersicht wurden alle Einstellungen in konstanten Variablen oberhalb der Methoden definiert. Auf die Bedeutung der einzelnen Variablen wird trivialerweise nicht weiter eingegangen.

Die Methode `rules()` gehört zum „Koch“-Projekt und beinhaltet die Wachstumsregel. XL-Methoden öffnen und schließen ihre Regelblöcke mit eckigen (`[` und `]`) statt geschweiften Klammern. Soll jedoch zusätzlich echter Java-Code ausgeführt werden, müssen imperative Blöcke (mit Java-Syntax) und Regelblöcke geschachtelt werden. In diesem Fall soll nach jedem Wachstumsschritt zusätzlich ein Bild mit den oben definierten Einstellungen gerendert werden.

Die Methode `init()` wird automatisch aufgerufen, wenn das Projekt initialisiert wird (manuell über einen Klick auf „reset“ oder automatisch beim Speichern des XL Codes). In diesem Projekt war die Methode nicht explizit hingeschrieben, weil keine weitere Handlung erforderlich war. In unserem Fall soll bei einem Zurücksetzen des Projektes die Liste (falls zuvor eine existierte) geleert werden.

Zum Schluss wird eine öffentliche Methode `makeVideo()` angehängt, welche zwischen den existierenden Bildern interpoliert und anschließend das Video erzeugt. Die Einstellungen hierfür finden sich auch oberhalb der Methoden. Bei der Interpolation gibt es zusätzlich die Möglichkeit zu definieren, bei welchen Bilder interpoliert werden soll. Bei Angabe keiner Parameter wird die Interpolationsstrategie automatisch auf die gesamte Bildsequenz angewendet. Ein finaler Aufruf der `exportVideo` Methode der XL Anbindung leitet die Videoerzeugung ein. Wird die Methode mit ungültigen Parametern (wie einer zu geringen Output-FPS Zahl) aufgerufen, schlägt der Exportvorgang fehl und der Benutzer wird über diesen Fehler informiert (Abbildung 5.14). Öffentliche Methoden werden automatisch oberhalb der Szenedarstellung als Schaltflächen eingeblendet.

Ein möglicher Arbeitsablauf nach Speichern des XL Codes wäre jetzt:

- Mehrfaches Betätigen der Schaltfläche „rules“, je nachdem wie viele Wachstumsschritte das Video enthalten soll
- Anschließendes Betätigen der Schaltfläche „makeVideo“

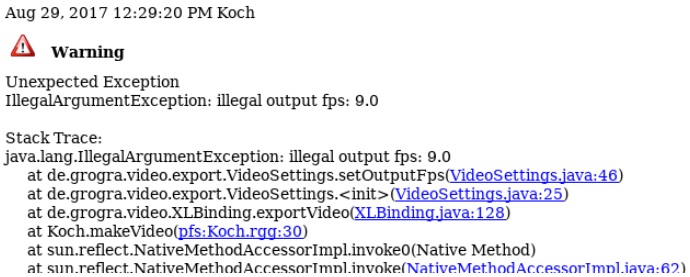

Abbildung 5.14: Fehler beim Exportieren über XL aufgrund falscher Parameter

Falls die leere Szene auch als Bild im Video vorhanden sein soll, kann dies mit einem Aufruf von `renderImage` am Ende der `init()` Methode erreicht werden.

Die vollständige Automatisierung der Videoerzeugung über XL ist leider nicht gelungen. Ein Ansatz war es innerhalb der `makeVideo` Methode die Wachstumsmethode `rules()` wiederholt aufzurufen und anschließend das Video zu erzeugen (Listing 5.5). Der wiederholte Aufruf von `rules()` bewirkt zwar, dass `loop`-viele Bilder gerendert werden, aber das eigentliche Wachstum scheint von GroIMP erst nachträglich durchgeführt zu werden. Das erzeugte Video enthält nämlich nur Bilder der initialen Szene, was sich beim „Koch“ Projekt durch einen schwarzen Bildschirm zeigt.

```
public void makeVideo() {
        int loops = 5;
        while (loops-- > 0)
                rules();

        // interpolate(INTERPOLATOR, INTERPOLATION_STEPS, INTERPOLATION_BEGIN,
            INTERPOLATION_END);
        interpolate(INTERPOLATOR, INTERPOLATION_STEPS);
        exportVideo(FILE_NAME, INPUT_FPS, OUTPUT_FPS);
}
```

Listing 5.5: Automatische Videoerzeugung in der `makeVideo` Methode

Bei Auswahl des `SceneSnapshot` Renderers in XL kommt es leider zu einem Deadlock, wenn die Szenedarstellung im „OpenGL“ Modus läuft. Das Problem ist durch die Synchronisierung des `SceneSnapshotMaker`s entstanden und konnte leider nicht gelöst werden. Würde auf diese Synchronisierung verzichtet werden, ist der Renderer `SceneSnapshot` aus XL heraus lauffähig, schlägt aber dafür über die Nutzeroberfläche fehl. Hier wurde die Entscheidung getroffen vorzugsweise die Benutzeroberfläche nutzbar zu machen.

5.4.2 Beispielprojekte „Game of Life“ / „Ludo Game“

In Projekten wie „Game of Life“ oder „Ludo Game“ (dem klassischen Mensch-Ärger-Dich-Nicht) kann es interessant sein sehr viele Bilder in ein Video zu exportieren. Dies kann einfach dadurch erreicht werden, dass die Schaltfläche „Run run“ bzw. „Run bremen“ / „Run karlsruhe“ in den Projekten betätigt wird. GroIMP wiederholt dann die Simulation in einer Endlosschleife und rendert in jedem Schritt ein Bild, dank der XL Anbindung des Video-Plugins. Sind genug Bilder gesammelt worden, beendet ein Klick auf „Stop“ die Schleife und die Videoerzeugung kann gestartet werden. So ist zwar eine exakte Anzahl von Zielbildern nicht einstellbar, aber die Erzeugung einer großen Zahl von Simulationen deutlich angenehmer. Im Anhang sind die Anpassungen der beiden Projekte in XL Code unter Listing D.1 und D.2 zu finden.

5.5 Test unter Windows

Nach Abschluss aller vorherigen Tests wurde das Plugin noch einmal unter Windows 10 getestet. Sowohl die grafische Benutzeroberfläche als auch die XL Anbindung funktionieren im Wesentlichen gleich und verursachten keine neuen Fehler. Lediglich die Vorschauansicht zeigt unter Umständen Bildfragmente rechts neben der Vorschau, wenn das Panel in der Größe verändert wird. Die Funktionalität des Bildersammelns oder des anschließenden Exports sind davon nicht betroffen.

Abbildung 5.15: Fehlerhafte Darstellung des Vorschaubildes unter Windows 10

Auch der Flux Renderer wurde unter Windows getestet, um ein Linux plattformspezifisches Problem ausschließen zu können. Leider war dieser Renderer auch unter Windows 10 auf dem Testsystem nicht funktionstüchtig (weder mit noch ohne Plugin). Die Ursache ist nicht bekannt.

5.6 Optimierungen

Nachdem alle Testergebnisse vorliegen, soll noch einmal separat über Verbesserungen gesprochen werden, die nach Analyse der Probleme vorgenommen worden sind oder die noch möglich wären. Wie bereits in Abschnitt 5.1.2 und 5.1.3 erwähnt, wurde die Klasse `VideoImage` mehrfach überarbeitet, um Probleme mit mangelndem Heap Speicherplatz zu lösen. Nachdem sich das Speichern der skalierten Vorschaubilder im Heap als zu speicher aufwendig herausgestellt hat, wurden die Vorschaubilder zunächst aus den von der Festplatte geladenen Originalbildern berechnet und anschließend direkt in skalierter Form zusätzlich gespeichert, um das Laden großer Datenmengen zu vermeiden.

Aus diesen Modifikationen hat sich ergeben, dass die Wiedergabegeschwindigkeit der Videovorschau-Funktion erheblich von der finalen Videolänge abgewichen hat. Der für die Animation zuständige `AnimationThread` innerhalb des `PreviewPanel`s hat anfangs vor jedem Weiterschalten des Bildes stets eine fixe Zeitlänge gewartet, die sich aus der Wiedergabegeschwindigkeit in FPS berechnen ließ. Dieser Ansatz ignoriert jedoch das Problem, dass das Laden des Bildes keine sofortige Aktion ist und in diesem Fall sogar einen erheblichen Zeitanspruch mit sich bringt. Obwohl die relative zeitliche Differenz mit dem Speichern der skalierten Bilder auf knapp 44% im Vergleich zu über 800% (bei berechneten Vorschaubildern) reduziert werden konnte, gab es meinerseits trotzdem den Anspruch das Video in korrekter Länge anzeigen zu können.
Dafür wurde der `AnimationThread` insoweit geändert, dass er nun in deutlich schnellerer Geschwindigkeit prüft wie viel Zeit seit Beginn der Vorschau vergangen ist und welches Bild gerade angezeigt werden sollte. Auf diese Weise wird die Animation auf jeden Fall zur korrekten Zeit beendet (mit geringfügiger Abweichung, falls das letzte Bild geladen werden muss). Das hat natürlich zur Folge, dass bei einer sehr schnellen Animation unter Umständen nicht alle Bilder zwischendurch angezeigt werden. Dieses Problem sollte aber nicht weiter negativ ins Gewicht fallen. Hier wurde entschieden, dass die Wiedergabe der wirklichen Videolänge von sehr viel größerer Bedeutung ist.

Eine weitere mögliche Modifikation bestünde darin, das `BufferedImage`, welches zum Zwecke der Speicherung in der Klasse `VideoImage` erstellt wird, nicht jedes Mal neu zu erzeugen, sondern wiederzuverwenden. Diese Idee wurde allerdings nicht mehr umgesetzt, da die letzten Verbesserungen der Klasse bereits alle Speicherprobleme gelöst haben. Die Berechnung der Interpolationsbilder könnte außerdem überarbeitet werden, sodass diese Operation parallelisiert ausgeführt werden kann. Hierdurch würden sich bei Mehrkernsystemen deutliche Vorteile zeigen.

Kapitel 6

Zusammenfassung und Ausblick

6.1 Zusammenfassung

Dieses Kapitel fasst das Ergebnis der vorliegenden Bachelorarbeit zusammen. Die Zielsetzung dieser Arbeit war die Erweiterung der Software GroIMP um die Möglichkeit der Videoerzeugung. Dieses Ziel wurde erreicht. Es gibt jedoch noch ungeklärte Probleme beim Verwenden des Plugins, weshalb eine uneingeschränkte und fehlerfreie Nutzung derzeit leider noch nicht möglich ist. Das liegt zum einen daran, dass es allem Anschein nach Synchronisierungsprobleme bei der Kommunikation mit GroIMP gibt, welche zur Folge haben, dass teils nicht fertig gerenderte Bilder dennoch in die Videodatei geschrieben werden. Zum anderen gibt es eine Reihe gänzlich ungeklärter Probleme, bei denen einige Renderer von GroIMP keine sinnvollen Bilder erzeugen oder das Programm mit Fehlermeldungen abstürzt. Dieses Verhalten konnte wiederholt beobachtet werden, ist jedoch nicht systematisch reproduzierbar.
Das Erzeugen der Bilder funktioniert aber nur dank einer umgesetzten Notlösung in einer existierenden GroIMP Klasse, welche in einem produktiven System so nicht existieren darf. Dieser Versuch einer Lösung kann in einigen Situationen vermutlich dennoch fehlerbehaftet sein und sogar unerwünschte Nebeneffekte mit sich bringen (welche bei den Tests jedoch nicht beobachtet wurden).

Die Videoerzeugung ist sowohl über eine interaktive Benutzeroberfläche als auch iterativ über XL Code steuerbar. Die grafische Oberfläche bietet eine intuitive Vorschaufunktion, mit der der aktuelle Stand des Videos begutachtet werden und durch einzelne Bilder navigiert werden kann. Beim Entwurf wurde viel Wert darauf gelegt, dass die eigentliche Video-Export-Funktionalität komplett losgelöst von GroIMP ist. Das hat dazu geführt, dass ein Test des isolierten Plugins trotz nun korrigierter Fehler funktioniert. Erst in Kombination mit GroIMP sind Probleme aufgetreten. Durch diese Isolation kann die Ursache der Probleme für zukünftige Arbeiten besser eingegrenzt werden.

6.2 Ausblick

Dieser letzte Abschnitt der Arbeit zeigt mögliche Richtungen auf, in denen an diese Arbeit in Zukunft angeknüpft werden kann.

Mit vergleichsweise wenig Aufwand könnte beispielsweise die Interpolation der Bilder parallelisiert werden, um die Vorteile von Mehrkernprozessoren zu verwenden. Auch könnte die Speichernutzung weiterhin optimiert werden, indem für die einzelnen Bilder nicht ständig neue Objekte erzeugt, sondern die alten wiederverwendet werden. Auf letzteres wurde in dieser Arbeit jedoch verzichtet, da aufgetretene Speicherprobleme ausreichend gelöst wurden.

Eine genaue Analyse der aufgetretenen Synchronisierungsprobleme sowie ein detaillierteres Studium der GroIMP Software in einer fortführenden Arbeit könnten diese Fehler beheben. GroIMP besitzt selbst eine Art Job Manager, der für die Abarbeitung von Aufgaben (wie auch das Rendern von Bildern) zuständig zu sein scheint. Eine korrekte Verwendung dieses Managers könnte die Problematik der Synchronisation beheben.

Das externe Programm FFmpeg wurde nur sehr grundlegend an GroIMP angebunden. Hier könnten durch eine weitere Arbeit viel mehr Einstellungsmöglichkeiten durch Erweiterung der `VideoSettings` Klasse zugänglich gemacht werden, sodass noch mehr Dateiformate oder auch Video Codecs auswählbar sind. Auch wäre es denkbar, FFmpeg in Zukunft durch ein möglicherweise besseres Programm auszutauschen. Für den derzeitigen Stand bietet das Programm jedoch ausreichend Funktionalität an.

Die Berechnung der Zwischenbilder kann um weitere Algorithmen ergänzt werden, um das Resultat des finalen Videos zu verbessern und einen besseren Wachstumseindruck zu bekommen. Es existieren bereits Arbeiten die sich damit befassen, die Animation „weicher“ zu machen [24] [25]. Dies wird durch Modifikation der Wachstumsgrammatiken erreicht. Ein solcher Ansatz wäre auch als Integration in die Video-Export Funktion denkbar.

Anhang A

Testsysteme

Betriebssystem	Arch Linux x86_64
Kernel	4.12.8-2-ARCH
Desktop-Environment	XFCE4
Prozessor	Intel(R) Core(TM) i5-4690K CPU 3.50 GHz
Arbeitsspeicher	32 GB
Festplatte	Samsung SSD 850 Pro 256 GB
Homeverzeichnis	`/home/dominick/`
Tempverzeichnis	`/tmp/`

Tabelle A.1: Testumgebung A: Linux

Betriebssystem	Microsoft Windows 10 Professional x64
Kernel	*Stand 22.08.2017*
Desktop-Environment	*Standard*
Prozessor	Intel(R) Core(TM) i5-4690K CPU 3.50 GHz
Arbeitsspeicher	32 GB
Festplatte	Samsung SSD 850 Pro 256 GB
Homeverzeichnis	`C:\Users\dominick\`
Tempverzeichnis	`C:\Users\dominick\AppData\Local\Temp\`

Tabelle A.2: Testumgebung B: Windows

Anhang B

Java Messklassen

```
package de.grogra.video.test;

public class HeapMonitor {
        public enum Format {
                RAW, HUMAN_READABLE;
        }

        private static HeapMonitor instance;

        private Runtime runtime;
        private long maxJavaSpace;
        private long maxSpace;
        private long space;

        private Format format = Format.HUMAN_READABLE;

        private HeapMonitor() {
                runtime = Runtime.getRuntime();
                maxJavaSpace = runtime.maxMemory();
                maxSpace = runtime.totalMemory();
        }

        public void setFormat(Format format) {
                this.format = format;
        }

        public void measure() {
                maxJavaSpace = runtime.maxMemory();
                maxSpace = runtime.totalMemory();
                space = maxSpace - runtime.freeMemory();
        }
```

```
        public String toString() {
                if (format == Format.HUMAN_READABLE)
                        return "Heap measured:   S = " + humanReadable(space) + "   CM =
                             " + humanReadable(maxSpace) + "   M = "
                                        + humanReadable(maxJavaSpace) + "   S/CM = "
                                        + String.format("%.2f", ((double) 100 * space /
                                            maxSpace)) + "%";
                else
                        return "" + space + ";" + maxSpace + ";" + maxJavaSpace;
        }

        public static synchronized HeapMonitor instance() {
                if (instance == null)
                        instance = new HeapMonitor();
                return instance;
        }

        public static String humanReadable(long bytes) {
                String[] units = new String[] { "B", "KB", "MB", "GB" };
                long pre = bytes;
                long post = 0L;
                int i = 0;

                while (pre > 999L) {
                        post = pre % 1000L;
                        pre /= 1000L;
                        i++;
                }

                return String.format("%3d.%03d ", pre, post) + units[i];
        }
}
```

Listing B.1: Die Klasse `HeapMonitor` zum Messen der Speicherauslastung

```
package de.grogra.video.test;

public class TimeStopper {
        public enum Format {
                RAW, HUMAN_READABLE;
        }

        private long start;
        private long end;

        private Format format = Format.HUMAN_READABLE;

        private static TimeStopper instance;

        private TimeStopper() {

        }

        public static synchronized TimeStopper instance() {
                if (instance == null)
                        instance = new TimeStopper();
                return instance;
        }

        public void setFormat(Format format) {
                this.format = format;
        }

        public void start() {
                start = System.nanoTime();
        }

        public void stop() {
                end = System.nanoTime();
        }

        public String toString() {
                if (format == Format.HUMAN_READABLE)
                return "Time measured: " + String.format("%.4f", (double) (end - start)
                    / 1000000000) + " s";
                else
                        return String.format("%.4f", (double) (end - start) /
                            1000000000);
        }
}
```

Listing B.2: Die Klasse `TimeStopper` zum Messen von Ausführungszeiten

Anhang C

Messdaten

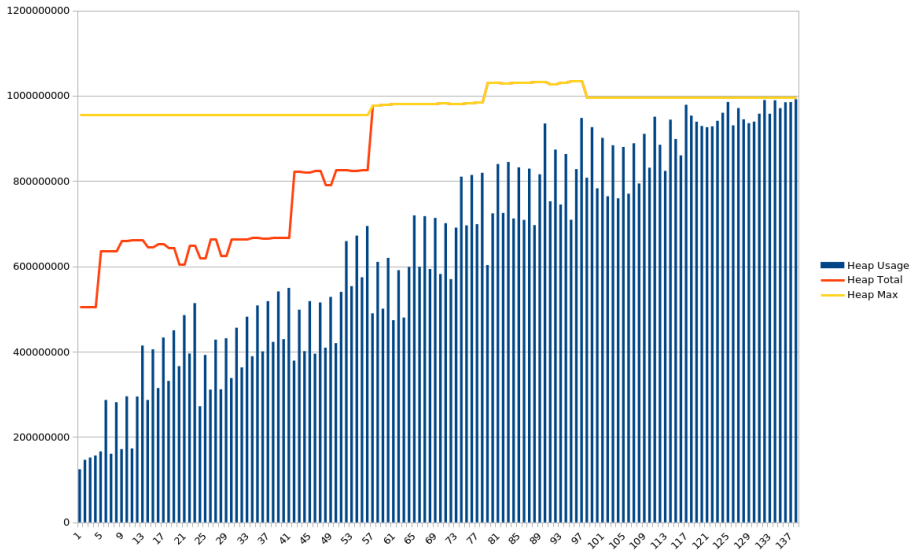

Abbildung C.1: Heapauslastung bei 640×360 Pixeln und 1 GB Heaplimit (Heap)

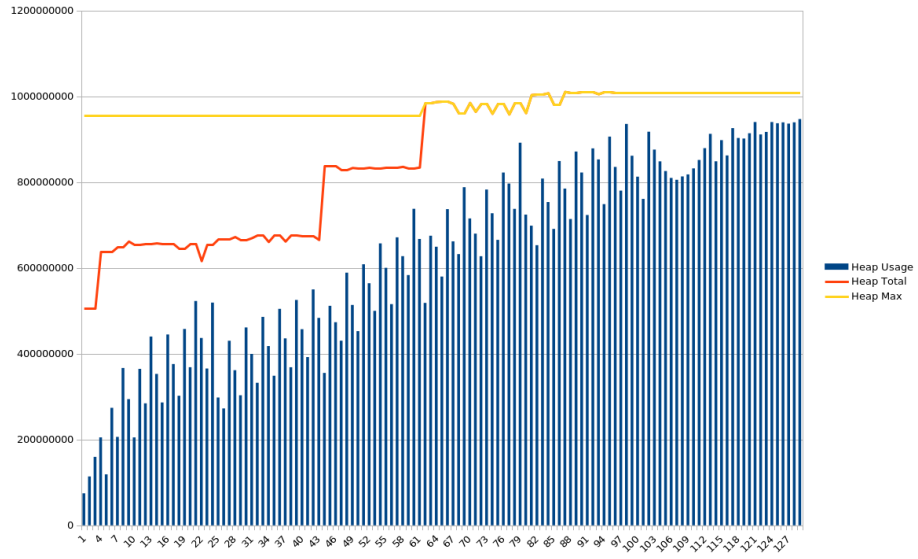

Abbildung C.2: Heapauslastung bei 1920×1080 Pixeln und 1 GB Heaplimit (Heap)

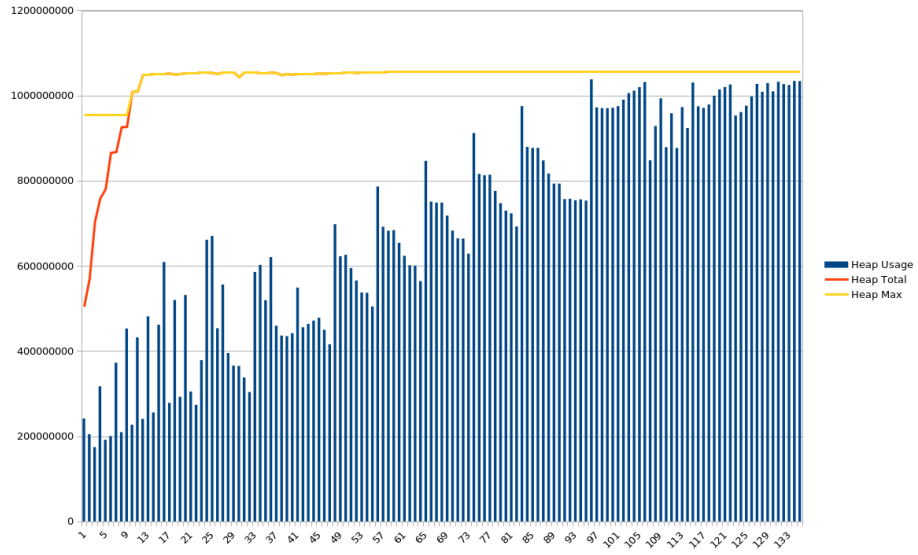

Abbildung C.3: Heapauslastung bei 3840×2160 Pixeln und 1 GB Heaplimit (Heap)

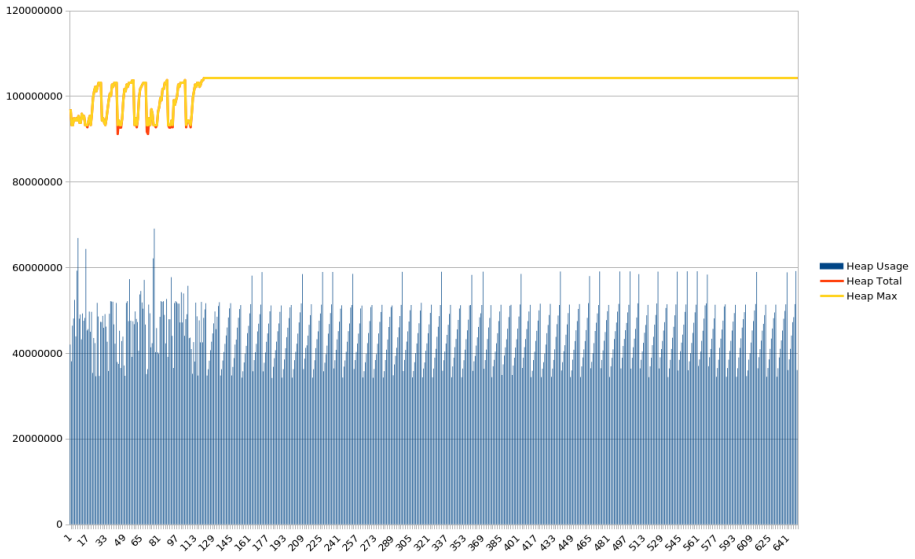

Abbildung C.4: Heapauslastung bei 1920×1080 Pixeln und 100 MB Heaplimit (Berechnet)

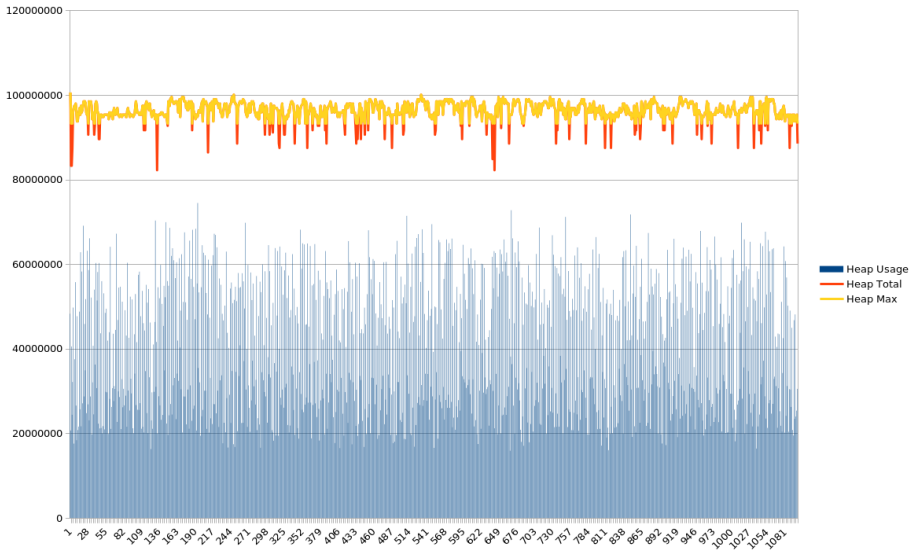

Abbildung C.5: Heapauslastung bei 640×360 Pixeln und 100 MB Heaplimit (Festplatte)

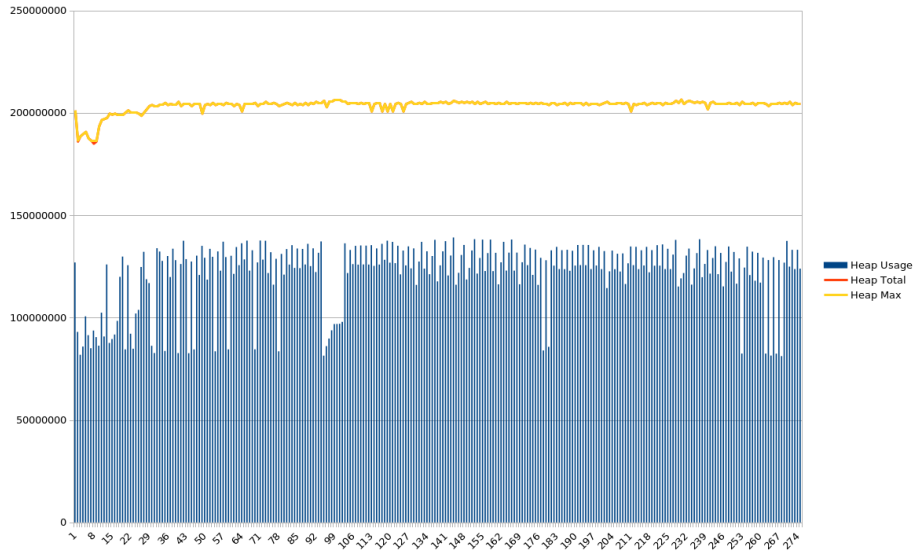

Abbildung C.6: Heapauslastung bei 3840×2160 Pixeln und 200 MB Heaplimit (Festplatte)

FPS	t_{Correct}	$t_{\min}$	$\Delta t[s]$	$\Delta t[\%]$	$t_1[s]$	$t_2[s]$	$t_3[s]$	$t_4[s]$	$t_5[s]$
10	$29,5000$	$29,5414$	$0,0414$	$0,14\%$	$29,5472$	$29,5431$	$29,5414$	$29,5459$	$29,5482$
30	$9,8333$	$9,7692$	$0,0641$	$0,65\%$	$9,7706$	$9,7703$	$9,7692$	$9,7713$	$9,7698$
60	$4,9167$	$4,7496$	$0,1671$	$3,40\%$	$4,7496$	$4,7508$	$4,7497$	$4,7550$	$4,7508$
120	$2,4583$	$2,3864$	$0,0719$	$2,93\%$	$2,3871$	$2,3872$	$2,3864$	$2,3870$	$2,3864$

Tabelle C.1: Vorschauzeitmessung (Heap). Auflösung: 160×90 Pixel, Bilder: 295

FPS	t_{Correct}	$t_{\min}$	$\Delta t[s]$	$\Delta t[\%]$	$t_1[s]$	$t_2[s]$	$t_3[s]$	$t_4[s]$	$t_5[s]$
10	$29,5000$	$29,5399$	$0,0399$	$0,14\%$	$29,5457$	$29,5441$	$29,5418$	$29,5399$	$29,5414$
30	$9,8333$	$9,7690$	$0,0643$	$0,65\%$	$9,7690$	$9,7707$	$9,7704$	$9,7699$	$9,7705$
60	$4,9167$	$4,7490$	$0,1677$	$3,41\%$	$4,7490$	$4,7893$	$4,7492$	$4,7493$	$4,7493$
120	$2,4583$	$2,3864$	$0,0719$	$2,93\%$	$2,3885$	$2,3869$	$2,3874$	$2,3864$	$2,3864$

Tabelle C.2: Vorschauzeitmessung (Heap). Auflösung: 1280×720 Pixel, Bilder: 295

FPS	t_{Correct}	$t_{\min}$	$\Delta t[s]$	$\Delta t[\%]$	$t_1[s]$	$t_2[s]$	$t_3[s]$	$t_4[s]$	$t_5[s]$
10	$29,5000$	$29,7558$	$0,2558$	$0,87\%$	$29,7558$	$29,7653$	$29,7587$	$29,7599$	$29,7623$
30	$9,8333$	$9,9490$	$0,1157$	$1,18\%$	$9,9643$	$9,9664$	$9,9515$	$9,9490$	$9,9544$
60	$4,9167$	$4,8895$	$0,0272$	$0,55\%$	$4,8954$	$4,8923$	$4,8992$	$4,8960$	$4,8895$
120	$2,4583$	$2,5044$	$0,0461$	$1,87\%$	$2,5044$	$2,5080$	$2,5156$	$2,5129$	$2,5099$

Tabelle C.3: Vorschauzeitmessung (Berechnet). Auflösung: 160×90 Pixel, Bilder: 295

FPS	t_{Correct}	$t_{\min}$	$\Delta t[s]$	$\Delta t[\%]$	$t_1[s]$	$t_2[s]$	$t_3[s]$	$t_4[s]$	$t_5[s]$
10	29,5000	33,7047	4,2047	14,25%	33,7138	33,7047	33,8706	33,9808	33,9442
30	9,8333	12,4824	2,6491	26,94%	12,4842	12,5166	12,5086	12,4824	12,4947
60	4,9167	7,3709	2,4542	49,92%	7,4324	7,3984	7,4134	7,4094	7,3709
120	2,4583	5,0343	2,5760	104,79%	5,0433	5,1080	5,0601	5,0492	5,0343

Tabelle C.4: Vorschauzeitmessung (Berechnet). Auflösung: 1280×720 Pixel, Bilder: 295

FPS	t_{Correct}	$t_{\min}$	$\Delta t[s]$	$\Delta t[\%]$	$t_1[s]$	$t_2[s]$	$t_3[s]$	$t_4[s]$	$t_5[s]$
10	29,5000	49,5442	20,0442	67,95%	49,5442	49,6778	49,7077	49,7287	49,8316
30	9,8333	29,7202	19,8869	202,24%	29,7202	29,7840	29,9169	29,7851	29,8432
60	4,9167	24,6503	19,7336	401,36%	24,6503	24,6753	24,7177	24,8146	24,7664
120	2,4583	22,2766	19,8183	806,17%	22,3131	22,3337	22,2766	22,3473	22,3939

Tabelle C.5: Vorschauzeitmessung (Berechnet). Auflösung: 3840×2160 Pixel, Bilder: 295

FPS	t_{Correct}	$t_{\min}$	$\Delta t[s]$	$\Delta t[\%]$	$t_1[s]$	$t_2[s]$	$t_3[s]$	$t_4[s]$	$t_5[s]$
10	29,5000	31,1646	1,6646	5,64%	31,2763	31,2556	31,1994	31,2463	31,1646
30	9,8333	11,1068	1,2735	12,95%	11,1068	11,1402	11,1363	11,1406	11,1336
60	4,9167	5,8623	0,9456	19,23%	5,9446	5,8992	5,8857	5,9190	5,8623
120	2,4583	3,3164	0,8581	34,90%	3,3470	3,3164	3,3178	3,3894	3,3173

Tabelle C.6: Vorschauzeitmessung (Festplatte). Auflösung: 160×90 Pixel, Bilder: 295

FPS	t_{Correct}	$t_{\min}$	$\Delta t[s]$	$\Delta t[\%]$	$t_1[s]$	$t_2[s]$	$t_3[s]$	$t_4[s]$	$t_5[s]$
10	29,5000	30,2032	0,7032	2,38%	30,2032	31,6128	31,6412	31,5924	31,5223
30	9,8333	11,3580	1,5247	15,51%	11,3580	11,4040	11,3968	11,3742	11,3871
60	4,9167	6,0932	1,1765	23,93%	6,1886	6,2040	6,1335	6,1258	6,0932
120	2,4583	3,4890	1,0307	41,93%	3,5411	3,5151	3,5059	3,4896	3,4890

Tabelle C.7: Vorschauzeitmessung (Festplatte). Auflösung: 1280×720 Pixel, Bilder: 295

FPS	t_{Correct}	$t_{\min}$	$\Delta t[s]$	$\Delta t[\%]$	$t_1[s]$	$t_2[s]$	$t_3[s]$	$t_4[s]$	$t_5[s]$
10	29,5000	31,4784	1,9784	6,71%	31,5646	31,5316	31,5839	31,4787	31,4784
30	9,8333	11,3068	1,4735	14,98%	11,3431	11,4094	11,3749	11,3068	11,3195
60	4,9167	6,0873	1,1706	23,81%	6,1433	6,1638	6,1386	6,1241	6,0873
120	2,4583	3,5297	1,0714	43,58%	3,5355	3,5297	3,5529	3,5558	3,5425

Tabelle C.8: Vorschauzeitmessung (Festplatte). Auflösung: 3840×2160 Pixel, Bilder: 295

Anhang D

XL Projekte

```
import static de.grogra.video.XLBinding.*;

public static final int RENDER_WIDTH = 1280;
public static final int RENDER_HEIGHT = 720;
public static final String RENDERER = "Raytracer";
public static final String INTERPOLATOR = "AlphaFading";
//public static final int INTERPOLATION_BEGIN = 1;
//public static final int INTERPOLATION_END = 5;
public static final int INTERPOLATION_STEPS = 10;
public static final String FILE_NAME = "/home/dominick/xl_test.mp4";
public static final int INPUT_FPS = 10;
public static final int OUTPUT_FPS = 25;

const int SIZE_X = 12;
const int SIZE_Y = 10;

static boolean testNeighbour (double d, int size)
{
        return Math.abs ((d > 2) ? (d - size) : (d < -2) ? (d + size) : d) < 1.1;
}

static boolean neighbour (Cell c1, Cell c2)
{
        return (c1 != c2) && testNeighbour(c1[x] - c2[x], SIZE_X)
                && testNeighbour(c1[y] - c2[y], SIZE_Y);
}


protected void init () {
        clear();
        [
```

```
                Axiom ==>> ^ for (int i : (1 : SIZE_X)) for (int j : (1 : SIZE_Y))
                                ( [Cell(i, j, 0, (((i == 4) && (j == 8))
                                                                || ((i == 5)
                                                                    && (j ==
                                                                    7))
                                                                || ((i in
                                                                    (3:5)) &&
                                                                    (j == 6)))
                                                                     ? 1 : 0)
                                                                    ]);
        ]
        renderImage(RENDERER, RENDER_WIDTH, RENDER_HEIGHT);
}

public void run () {
        [
                x:Cell(1), (!(sum ((* x -neighbour-> Cell *)[state]) in (2 : 3)))
                        ::> x[state] := 0;

                        x:Cell(0), (sum ((* x -neighbour-> Cell *)[state]) == 3)
                        ::> x[state] := 1;
        ]
        renderImage(RENDERER, RENDER_WIDTH, RENDER_HEIGHT);
}

public void makeVideo() {
        // interpolate(INTERPOLATOR, INTERPOLATION_STEPS, INTERPOLATION_BEGIN,
            INTERPOLATION_END);
        interpolate(INTERPOLATOR, INTERPOLATION_STEPS);
        exportVideo(FILE_NAME, INPUT_FPS, OUTPUT_FPS);
}
```

Listing D.1: XL Code des Projektes „GameOfLife“

```
import static de.grogra.video.XLBinding.*;

public static final int RENDER_WIDTH = 1280;
public static final int RENDER_HEIGHT = 720;
public static final String RENDERER = "Raytracer";
public static final String INTERPOLATOR = "AlphaFading";
//public static final int INTERPOLATION_BEGIN = 1;
//public static final int INTERPOLATION_END = 5;
public static final int INTERPOLATION_STEPS = 10;
public static final String FILE_NAME = "/home/dominick/xl_test.mp4";
public static final int INPUT_FPS = 10;
public static final int OUTPUT_FPS = 25;

/*

Ludo -- Solution using XL
Ole Kniemeyer

All transformations are invoked via the RGG toolbar (above the 3D view), [...]

*/


const Shader[] colors = {
        // use different shaders for the GUI and the raytracer
        new AlgorithmSwitchShader(RED, shader("Red"), null),
        new AlgorithmSwitchShader(BLUE, shader("Blue"), null),
        new AlgorithmSwitchShader(YELLOW, shader("Yellow"), null),
        new AlgorithmSwitchShader(GREEN, shader("Green"), null)};

const Shader alu = shader("Brushed Aluminium");
const Shader simpleDieShader = shader("Simple Die");
const Shader dieSpot = shader("Die Spot");
const Shader dieShader = shader("Die");

// much much code here ...


// creates the initial graph, quite tedious
protected void init()
[
        {
                clear();
                this[move] = 0;
        }
        // ...
]

// see pdf description
```

```
public void karlsruhe()
{
        this[move]++;
        setStatus("Move " + move);
        // ...
        renderImage(RENDERER, RENDER_WIDTH, RENDER_HEIGHT);
}

// more code ...

public void makeVideo() {
        // interpolate(INTERPOLATOR, INTERPOLATION_STEPS, INTERPOLATION_BEGIN,
            INTERPOLATION_END);
        interpolate(INTERPOLATOR, INTERPOLATION_STEPS);
        exportVideo(FILE_NAME, INPUT_FPS, OUTPUT_FPS);
}
```

Listing D.2: XL Code des Projektes „Ludo Game“

Anhang E

Fehlermeldungen

```
java.lang.NullPointerException
        at de.grogra.graph.ObjectTreeAttribute.getDerived(ObjectTreeAttribute.java:227)
        at de.grogra.graph.GraphState.getObject(GraphState.java:1807)
        ...
        at de.grogra.imp3d.ray2.Raytracer.render(Raytracer.java:156)
        at de.grogra.imp.Renderer.computeImage(Renderer.java:105)
        at de.grogra.video.render.RendererAdaptor.createImage(RendererAdaptor.java:68)
        at de.grogra.video.render.AutoRenderJob.process(AutoRenderJob.java:44)
        at de.grogra.video.util.WorkerThread.run(WorkerThread.java:25)
Exception in thread "ViewThread@de.grogra.imp3d.WireframeCanvas@430d5a8b" java.lang.
    NullPointerException
        at de.grogra.turtle.Shoot.getInt(Shoot.java:185)
        at de.grogra.turtle.F.getInt(F.java:259)
        ...
        at de.grogra.imp.awt.ViewComponentAdapter.run(ViewComponentAdapter.java:318)
        at java.lang.Thread.run(Thread.java:748)
Aug 26, 2017 4:10:41 PM de.grogra.imp.IMPJobManager run
WARNING: Unexpected Exception
java.lang.NullPointerException
        at de.grogra.turtle.Shoot.getFloat(Shoot.java:162)
        at de.grogra.turtle.F.getFloat(F.java:251)
        ...
        at de.grogra.pf.ui.awt.EventAdapter.run(EventAdapter.java:114)
        at de.grogra.imp.IMPJobManager.run(IMPJobManager.java:550)
        at java.lang.Thread.run(Thread.java:748)

Aug 26, 2017 4:10:41 PM de.grogra.imp.IMPJobManager run
WARNING: Unexpected Exception
java.lang.NullPointerException
        at de.grogra.turtle.Shoot.getFloat(Shoot.java:162)
        at de.grogra.turtle.F.getFloat(F.java:251)
```

```
        ...
        at de.grogra.pf.ui.awt.EventAdapter.run(EventAdapter.java:114)
        at de.grogra.imp.IMPJobManager.run(IMPJobManager.java:550)
        at java.lang.Thread.run(Thread.java:748)

Aug 26, 2017 4:10:41 PM de.grogra.imp.IMPJobManager run
WARNING: Unexpected Exception
java.lang.NullPointerException
        at de.grogra.turtle.Shoot.getFloat(Shoot.java:162)
        at de.grogra.turtle.F.getFloat(F.java:251)
        ...
        at de.grogra.pf.ui.awt.EventAdapter.run(EventAdapter.java:114)
        at de.grogra.imp.IMPJobManager.run(IMPJobManager.java:550)
        at java.lang.Thread.run(Thread.java:748)

Aug 26, 2017 4:10:41 PM de.grogra.imp.IMPJobManager run
WARNING: Unexpected Exception
java.lang.NullPointerException
        at de.grogra.turtle.Shoot.getFloat(Shoot.java:162)
        at de.grogra.turtle.F.getFloat(F.java:251)
        at de.grogra.graph.impl.Node$AccessorBridge.getFloat(Node.java:755)
        ...
        at de.grogra.pf.ui.awt.EventAdapter.run(EventAdapter.java:114)
        at de.grogra.imp.IMPJobManager.run(IMPJobManager.java:550)
        at java.lang.Thread.run(Thread.java:748)

Aug 26, 2017 4:10:41 PM de.grogra.imp.IMPJobManager run
WARNING: Unexpected Exception
java.lang.NullPointerException
        at de.grogra.turtle.Shoot.getFloat(Shoot.java:162)
        at de.grogra.turtle.F.getFloat(F.java:251)
        ...
        at de.grogra.pf.ui.awt.EventAdapter.run(EventAdapter.java:114)
        at de.grogra.imp.IMPJobManager.run(IMPJobManager.java:550)
        at java.lang.Thread.run(Thread.java:748)
```

Listing E.1: Exception beim Auto-Render 6-fach im Projekt „Koch“

Anhang F

CD-Rom

Auf der hier befindlichen CD-Rom befinden sowohl eine digitale Version dieser Arbeit, als auch der dokumentierte Quelltext des Plugins.

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis

Abkürzungsverzeichnis

JVM	Java Virtual Machine	7
GUI	Graphical User Interface	7
XL	eXtended L-system language	2
RAM	Random Access Memory	26
RGG	Relational Growth Grammars	11
SSD	Solid State Drive	46
FPS	Frames Per Second	18
FIFO	First In First Out	29

Literaturverzeichnis

[1] Department Ecoinformatics, Biometrics and Forest Growth, Georg-August University of Göttingen, "grogra," http://www.grogra.de [Letzter Zugriff: 05.07.2017].

[2] OpenAlea Community, "OpenAlea," http://virtualplants.github.io [Letzter Zugriff: 25.09.2017].

[3] Algorithmic Botany and University of Calgary, "http://algorithmicbotany.org/lstudio," http://amapstudio.cirad.fr/ [Letzter Zugriff: 25.09.2017], aktuelle Version: Version 4.4.1 Build 2976.

[4] AMAP Community, "AMAPStudio," http://amapstudio.cirad.fr [Letzter Zugriff: 25.09.2017].

[5] Benjamin Krüger. Technische Universität Dortmund, "Videoformate im Vergleich," https://www.nrwision.de/lernen/tv-wissen/videoformate-im-vergleich.html [Letzter Zugriff: 30.08.2017].

[6] Wikipedia, "Bewegte Bilder," https://de.wikipedia.org/wiki/Bewegte_Bilder [Letzter Zugriff: 29.07.2017].

[7] H. J. Wulff and S. Lenk, "Phi-effekt," http://filmlexikon.uni-kiel.de/index.php?action=lexikon&tag=det&id=288 [Letzter Zugriff: 30.08.2017].

[8] Wikipedia, "Motion Interpolation," https://de.wikipedia.org/wiki/Motion_Interpolation [Letzter Zugriff: 29.07.2017].

[9] ——, "Interpolation (Fotografie)," https://de.wikipedia.org/wiki/Interpolation_(Fotografie) [Letzter Zugriff: 29.07.2017].

[10] FFmpeg-Projekt, "FFmpeg," https://ffmpeg.org/ [Letzter Zugriff: 10.07.2017], aktuelle Version: 3.3.3.

[11] Wikipedia, "FFmpeg," https://de.wikipedia.org/wiki/FFmpeg [Letzter Zugriff: 30.09.2017].

[12] C. Ullenboom, *Java ist auch eine Insel: Einführung, Ausblick, Praxis*, 11th ed. Galileo Computing, 2014.

[13] Oracle Corporation, "Java," https://www.java.com/de/ [Letzter Zugriff: 31.07.2017], aktuelle Version: Version 8 Update 144.

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Juli 2008.

[15] Planetmath, "Semi-Thue System," http://planetmath.org/semithuesystem [Letzter Zugriff: 02.09.2017].

[16] O. Kniemeyer, "Rule-based modelling with the XL/GroIMP software," in *Harald Schaub, Frank Detje, Ulrike Brüggemann (eds.), The Logic of Artificial Life: Abstracting and Synthesizing the Principles of Living Systems; Proceedings of the 6th GWAL, Bamberg 14.-16. 4.* AKA Akademische Verlagsges. Berlin, 2004.

[17] ——, "Design and Implementation of a Graph Grammar Based Language for Functional-Structural Plant Modelling," Ph.D. dissertation, Brandenburgische Technische Universität Cottbus, 2008.

[18] P. Prusinkiewicz, "Website of Przemyslaw Prusinkiewicz," http://pages.cpsc.ucalgary.ca/~pwp/ [Letzter Besuch: 02.09.2017].

[19] Department Ecoinformatics, Biometrics and Forest Growth, Georg-August University of Göttingen, "Relational Growth Grammars," http://wwwuser.gwdg.de/~groimp/grogra.de/grammars/rgg.html [Letzter Zugriff: 02.09.2017].

[20] D. Leppich, "Entwicklung eines Plugins für die 3D-Modellierungsplatform GroIMP," Institut für Informatik, Georg-August Universität Göttingen, Tech. Rep., Juli 2017.

[21] MKLab, "StarUML," http://staruml.io/ [Letzter Zugriff: 03.09.2017], aktuelle Version: 2.8.0.

[22] O. Kniemeyer, "Erklärungen zu GroIMP und der Pluginarchitektur," E-Mail, 2017.

[23] VideoLAN non-profit organization, "VLC media player," https://www.videolan.org/vlc/ [Letzter Zugriff: 27.08.2017].

[24] S. Chuai-Aree, W. Jäger, H. G. Bock, and S. Siripant, "Smooth Animation for Plant Groth Using Time Embedded Component and Growth Function," *Computational Mathematics and Modeling*, 2002.

[25] Y. Rodkaew, S. Chuai-Aree, S. Siripant, C. Lursinsap, and P. Chongstitvatana, "Animating Plant Growth in L-System by Parametric Functional Symbols," *International Journal of Intelligent Systems*, vol. 19, 2004.