

Anbindung der Programmiersprache XL an die 3D- Modelliersoftware 3ds max über ein Plug-in

Bachelorarbeit

zur Erlangung des Grades Bachelor of Science (B.Sc.)

Brandenburgische Technische Universität Cottbus

Institut für Informatik

Lehrstuhl Grafische Systeme

Uwe Mannl,

Matrikel-Nummer 2103858

Betreuer:

Prof. Dr. Winfried Kurth,

Dipl.-Phys. Ole Kniemeyer

Cottbus, den 13.09.2006

1	EINLEITUNG	5
2	L-SYSTEME	7
2.1	TURTLE-GRAFIK.....	7
2.2	TURTLE-GRAFIK UND L-SYSTEME	8
2.2.1	<i>D0L-Systeme</i>	9
2.2.2	<i>Geklammerte L-Systeme</i>	10
2.2.3	<i>Parametrisierte L-Systeme</i>	10
2.2.4	<i>Stochastische L-Systeme</i>	10
2.2.5	<i>Kontextsensitive L-Systeme</i>	11
2.3	SENSITIVE WACHSTUMSGRAMMATIKEN	11
2.4	RELATIONALE WACHSTUMSGRAMMATIKEN	12
3	XL.....	13
3.1	AUFGABENSTELLUNG.....	14
3.2	JAVA - C++ ANBINDUNG	15
3.3	XL IN XL4MAX	17
3.4	ZUSÄTZLICHES IN XL4MAX.....	20
4	IMPLEMENTIERUNG	23
4.1	VERWENDETE SOFTWARE.....	23
4.2	VISUAL STUDIO .NET UND MAX-SDK	23
4.3	ECLIPSE.....	24
4.4	CODE-ERKLÄRUNG.....	26
4.5	C++	26
4.5.1	<i>maxProject1.cpp</i>	27
4.5.2	<i>xlEditor.cpp</i>	31
4.5.3	<i>jniInterface.cpp</i>	32
4.5.4	<i>jniMethods.cpp</i>	34
4.6	JAVA	38
4.6.1	<i>de.grogra.ext.max</i>	38
4.6.2	<i>de.grogra.ext.max.geomobj</i>	39
4.6.3	<i>de.grogra.ext.max.shape</i>	40
4.6.4	<i>de.grogra.ext.max.lsystem</i>	40
4.6.5	<i>de.grogra.ext.max.model</i>	42
5	XL4MAX-PLUG-IN.....	47
5.1	VORAUSSETZUNGEN VON XL4MAX	47
5.2	INSTALLATION VON XL4MAX	48
5.3	NUTZUNG DES PLUG-INS	49
5.3.1	<i>XL-Editor</i>	49
5.3.2	<i>Implementierung des XL-Editors</i>	50
5.3.3	<i>Gebrauch des XL-Editors</i>	51
5.3.4	<i>Einstellungsmenu</i>	52
5.3.5	<i>Implementierung des Einstellungsmenus</i>	53
5.3.6	<i>Gebrauch des Einstellungsmenus</i>	54
5.3.7	<i>XL-Methods</i>	54
5.3.8	<i>Implementierung von XL-Methods</i>	54
5.3.9	<i>Gebrauch von XL-Methods</i>	55
6	BEISPIELE	57
6.1	SIMPLE.XL	57
6.2	GAMEOFLIFE.XL.....	59
6.3	BUSH.XL	62
7	FAZIT.....	65
8	QUELLENVERZEICHNIS	67
9	ANHANG	69
9.1	AUFGABENSTELLUNG.....	69

1 Einleitung

Computergrafik - ein Bereich mit schier unbegrenzten Möglichkeiten. Was vor mehr als 20 Jahren mit Filmen wie „Blade Runner“ begann und damals in der breiten Masse der Bevölkerung großes Erstaunen hervorrief, ist inzwischen Alltag. Was sich damals noch auf einige gerenderte Sequenzen beschränkte, führte 1995 zu einem komplett animierten Kinofilm mit dem Namen „Toy Story“. Inzwischen ist die Menschheit solche Titel gewohnt und das Angebot schon übersättigt. Allein im Jahr 2006 kommt im Schnitt pro Monat mehr als 1 vollständig animierter Film in die Kinos. Dass darunter allerdings die Qualität leidet, weniger auf dem technischen als auf dem inhaltlichen Sektor, wird leider immer deutlicher. Hoffentlich erkennt bald auch der letzte Produzent, dass gute Filme nicht alleine durch die Technik entstehen.

Thema dieser Arbeit ist allerdings das Verfahren allein. Es geht dabei um den Versuch, ein weit verbreitetes Modellier- und Animationsprogramm für eine neue und für die Zukunft viel versprechende Technologie zu erweitern. Bei dem Programm handelt es sich um die Software Autodesk 3ds max. Schon für bekannte Filmproduktionen wie „Armageddon“, „Matrix“ sowie mehreren „Harry Potter“- und „Herr der Ringe“-Teilen eingesetzt, ist sie auch im Bereich der Spieleentwicklung von großer Bedeutung, auch wenn Titel wie „Need for Speed“ oder „Halo“ nur begeisterten *Zockern* bekannt sein dürften.

Die Technologie, von der hier die Rede ist, lässt sich am einfachsten mit dem Begriff „Ersetzung“ umschreiben. Präzise ausgedrückt geht es um die Anwendung von relationalen Wachstumsgrammatiken in 3ds max. Hinter diesem, recht kompliziert klingendem, Begriff steckt ein recht simples Prinzip zur Erzeugung komplexer Modelle und der Simulation von Abläufen. Denn eines hat sich in den vielen Jahren der Entwicklung der Computergrafik nicht geändert: Es erfordert immer noch großes Geschick, virtuelle Objekte zu erschaffen, die im Aussehen und im Verhalten der Realität in nichts nachstehen. Es werden schon Prozessoren entwickelt, die komplette Wälder in Echtzeit berechnen können und beeindruckende Bilder liefern. Was bringt uns jedoch die Möglichkeit, Licht und Schatten und sonstige visuelle Effekte hervorzaubern zu können, wenn der einzelne Baum nicht wie ein Baum aussieht? Oder wenn ein Baum wie der andere aussieht? Diesen Missstand an Details nimmt der menschliche Geist genauso wahr; und genau hier versuchen sich die Wachstumsgrammatiken. Die Umsetzung dieser XL getauften Technik ist schon vorhanden in der Software GroIMP. Allerdings ist diese Software bisher nur einem kleineren Kreis an Personen bekannt und in ihrem jetzigen Zustand nicht geeignet für den Einsatz in Animationsstudios - alleine, weil die Möglichkeit der Animation fehlt. Diese Lücke versucht das in dieser Arbeit entwickelte Plug-in zu schließen. Auch wenn es selbst nicht die eben angesprochene, gewünschte Reife besitzt, so ist es doch ein Ansatz. Und dank der Offenheit auf Seiten von GroIMP und XL4Max besteht jederzeit die Möglichkeit, das vorliegende Produkt in Zukunft weiterzuentwickeln.

In Kapitel 2 wird zunächst auf die Grundlagen von Wachstumsgrammatiken eingegangen, allem voran mit dem von Aristid Lindenmayer entwickelten System zur Pflanzenmodellierung. Diese noch recht einfache Technik erlaubte schon eine beeindruckende Möglichkeit, Bäume und andere Pflanzen nicht nur zu modellieren, sondern ihren kompletten Wachstumsprozess zu beschreiben. Über die verschiedenen Arten der Grammatiken geht es dann in Kapitel 3 um die konkrete Umsetzung von XL in Java und in einem Plug-in für 3ds max (XL4Max). Kapitel 4 beschäftigt sich ausgiebig mit den Details, speziell mit den Problemen bei der Implementierung von XL4Max. Dabei wird spezifisch auf das eingegangen, was aus dem Paket von XL auch endgültig

die Umsetzung in XL4Max fand. Mit der Nutzung des vorliegenden Plug-ins, also der Handhabung der grafischen Benutzeroberfläche (GUI), befasst sich Kapitel 5. Hierbei wird nochmals auf Probleme eingegangen, wie sie bei der Umsetzung der GUI auftraten. Kapitel 6 soll dann mit dem ersten Beispiel eine gute Einführung in die Benutzung von XL4Max bieten. Weiterhin dienen die anderen Modelle zum Verständnis von XL bei der Umsetzung komplexerer Beispiele.

Mit der vorliegenden Arbeit soll nicht nur dem Nutzer ein leichter Einstieg in die Thematik von XL und ihrer Anwendung in XL4Max ermöglicht werden. Auch soll ein potentieller Programmierer damit eine Übersicht über die angewendeten Techniken erhalten, sofern er die Notwendigkeit sieht oder einfach nur Spaß daran hat, das vorliegende Programm zu verbessern oder auszubauen.

2 L-Systeme

Die Fähigkeiten von XL4Max beruhen im Großen und Ganzen auf der Umsetzung der sogenannten Lindenmayer-Systeme, kurz L-Systeme genannt. Benannt wurde dieses Prinzip nach dem Biologen und Botaniker Aristid Lindenmayer. Tätig an der University of Utrecht, beschäftigte er sich mit den theoretischen Grundlagen des Pflanzen- und Zellenwachstums. Daraus entwickelte er 1968 die L-Systeme; einen mathematischen Formalismus, der in jüngster Zeit auch Anwendung in der Computergrafik findet.

Um L-Systeme, wie sie bei uns zur Anwendung kommen, verstehen zu können, bedarf es einigen Vorwissens um die Turtle-Grafik.

2.1 Turtle-Grafik

Die Turtle-Grafik ist eine einfache Möglichkeit, um bestimmte Muster auf einen geeigneten Untergrund zu bringen. Dies kann Papier sein, so wie es in der Drucktechnik bei der Nutzung von Stiftplottern zum Einsatz kommt, aber auch der Bildschirm, in dem der Computer die entsprechenden Befehle umsetzt. Dabei wird die Hintergrundfarbe durch die aufzutragende Farbe ersetzt. Die Möglichkeiten der Turtle-Grafik beschränken sich allerdings nicht nur auf den 2D-Bereich, sondern können durch eine Erweiterung des Befehlssatzes auch im 3D-Bereich zur Anwendung kommen. So lassen sich virtuelle Szenen erstellen, die der Nutzer regelrecht durchwandern kann, um die erstellten Grafiken von allen Seiten zu betrachten. Dabei ist er natürlich immer auf die ihm zur Verfügung stehenden Mittel eingeschränkt.

Im einfachen Fall einer 2D-Turtle-Grafik stellt man sich am besten ein weißes Blatt Papier vor, auf welches an einer bestimmten Stelle ein Stift gesetzt wird. Im Normalfall fängt eine Turtle-Grafik immer in den Koordinaten (0, 0) an. Wichtig ist, dass der Stift eine festgelegte Bewegungsrichtung besitzt. Jetzt lassen sich mit Hilfe der folgenden Befehle Bewegungen des Stiftes ausführen:

- F - Der Stift wird in der festgelegten Bewegungsrichtung nach vorne bewegt. Dabei bleibt der Stift auf dem Papier, so dass eine Linie entsteht.
- f - Der Stift wird in der festgelegten Bewegungsrichtung nach vorne bewegt. Während dieser Bewegung berührt der Stift allerdings das Papier nicht, so dass keine Linie entsteht.
- $+$ - Die Bewegungsrichtung des Stiftes ändert sich um einen festen Winkel gegen den Uhrzeigersinn. Dabei erfolgt keine Bewegung des Stiftes, sondern nur eine Rotation.
- $-$ - Die Bewegungsrichtung des Stiftes ändert sich um einen festen Winkel mit dem Uhrzeigersinn.

Dies sind nur die grundlegendsten Turtle-Befehle und auch diese lassen sich noch variieren. So kann die Länge der Linie, die durch F erzeugt wird, durch einen vorher ausgeführten Befehl geändert werden, oder direkt beim Aufruf durch F mittels Parameter, in dem man das Kommando wie folgt schreibt:

- $F(x)$ - Der Stift geht in der Bewegungsrichtung um x Einheiten nach vorne und malt dabei eine Linie.

Entsprechend verändert sich der Befehl f .

Für die Drehung des Stiftes lässt sich dann folgendes Muster anlegen:

$+(a)$ - Der Stift führt eine Rotation gegen den Uhrzeigersinn um den Winkel a aus.

Gleiches gilt natürlich wieder für das Kommando „-“.

Weitere Beispiele für Turtle-Befehle, wie sie auch in XL4Max zum Einsatz kommen, sind D (zum Setzen der Liniendicke), P (für die Farbe der Linie) und die Rotationskommandos im 3D-Raum wie RU , RH und RL . Eine komplette Übersicht findet sich in der Hilfe zu XL4Max. Die umgesetzten Befehle lehnen sich an die Turtle-Kommandos von [1] an, deren nähere Beschreibungen in [2] und [3] zu finden sind.

2.2 Turtle-Grafik und L-Systeme

Der Zusammenhang zwischen Turtle-Grafiken und L-Systemen besteht in ihrer Art der Anwendung. L-Systeme beschreiben Produktionsregeln, welche Ersetzungen anhand der definierten Grammatik ausführen. Dabei liegt das Hauptmerkmal darauf, dass alle Ersetzungen parallel ausgeführt werden. Durch die Möglichkeit der rekursiven Anwendung der Produktionsregeln entstehen dann die verschiedenen Entwicklungsstadien.

Die Struktur eines L-Systems ist wie folgt aufgebaut:

$$G = (V, \omega, P)$$

wobei V das Alphabet ist, $\omega \in V^*$ ein nichtleeres Wort über dem Alphabet V und P die Produktions-, also Ableitungsregeln. ω steht für das Startwort, das sogenannte Axiom.

Für ein einfaches Beispiel mittels Turtle-Grafik lässt sich folgendes L-System aufstellen:

$$G = (\{F, +, -\}, F, \{F \rightarrow F+F-F+F\})$$

Dieses L-System beschreibt eine Linie, die sich Koch-Kurve nennt und vom Mathematiker Helge von Koch im Jahr 1904 entwickelt wurde. In textueller Form sehen die Ableitungsschritte wie folgt aus:

0. Schritt: F
1. Schritt: F+F-F+F
2. Schritt: F+F-F+F+F+F-F-F+F-F-F+F+F+F-F+F

Wie zu erkennen ist, wird das Startwort durch die einzige vorhandene Produktionsregel ersetzt, und dies geschieht rekursiv in jedem weiteren Schritt. Jedes F des Wortes wird durch die rechte Seite der Ableitungsregel ersetzt. Zur besseren Veranschaulichung eignet sich die grafische Umsetzung wie in den folgenden Abbildungen 1a bis 1d:

0. Schritt:

Abbildung 1a: Start der Produktionsschritte

1. Schritt:



Abbildung 1b: Koch-Kurve nach einem Produktionsschritt

2. Schritt:

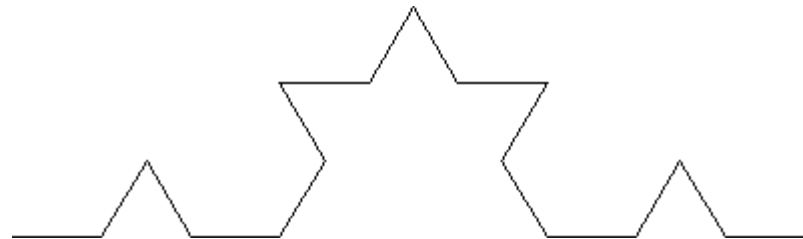


Abbildung 1c: Koch-Kurve nach zwei Produktionsschritten

3. Schritt:

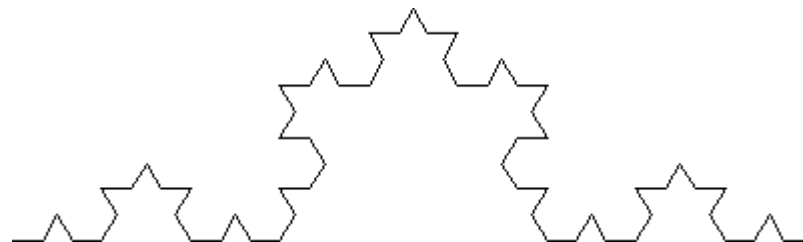


Abbildung 1d: Koch-Kurve nach drei Produktionsschritten

In diesem Fall ist aber zu beachten, dass das gesamte Objekt mit jedem Erzeugungsschritt sehr in der Größe anwachsen würde, da die Länge der F s konstant bleibt. Dies kann man umgehen, indem man die Ersetzungsregel so erweitert, dass nicht nur jedes F ersetzt wird, sondern auch dessen Länge ausgelesen und bei der Ersetzung mit berücksichtigt wird. So ist es auch unter XL4Max möglich, wie in dem beiliegenden Beispiel „Koch“ aus „Fraktal.xl“ zu sehen ist.

Die L-Systeme lassen sich in verschiedene Kategorien unterteilen: [4]

2.2.1 D0L-Systeme

Die einfachste Art der L-Systeme sind die D0L-Systeme, zu denen auch das eben angeführte Koch-Beispiel gehört. D0L steht für deterministische und kontextfreie L-Systeme. Hier lässt sich auch an einem anderem, einfachen Beispiel die parallele Abarbeitung der Ableitungsregeln gut erklären. Gegeben sei folgendes L-System:

$$G = (\{A, B\}, B, \{A \rightarrow AB, B \rightarrow A\})$$

Damit ergeben sich folgende Ableitungsschritte:

- 0. Schritt: B
- 1. Schritt: A
- 2. Schritt: AB
- 3. Schritt: ABA

Vom zweiten zum dritten Schritt ist zu erkennen, dass A mit AB nicht sequentiell ersetzt wird. Dabei würde das Wort ABB erzeugt werden und dann eine Ableitung von B nach A erfolgen. Dies würde zu dem Wort AAA führen. Stattdessen erfolgt eine getrennte Ableitung des Wortes AB in den Teilen A nach AB und B nach A, und am Ende ein Zusammensetzen dieser Teibleitungen zu ABA.

2.2.2 Geklammerte L-Systeme

Mit Hilfe der Klammerung lassen sich Verzweigungen erstellen. Dazu werden 2 neue Befehle eingeführt, die die folgende Notation besitzen:

- / - Es erfolgt eine Speicherung des aktuellen Zustandes (Position, Drehung, Dicke, ...) des Stiftes auf einen Stack.
- / - Das oberste Element des Stacks wird herausgenommen und der damit gespeicherte Zustand wiederhergestellt. Ein Positionswechsel des Stiftes wird dabei nicht mit dem Zeichnen einer Linie begleitet.

Ein gutes Beispiel zur Veranschaulichung ist folgendes L-System, gegeben durch

$$G = (\{F, +, -, [, \}, F, \{F \rightarrow F[-F]F[+F][F]\})$$

Daraus ergeben sich die folgenden grafischen Produktionsschritte der Abbildung 2, vom 0. Schritt links bis zum 3. Schritt rechts:

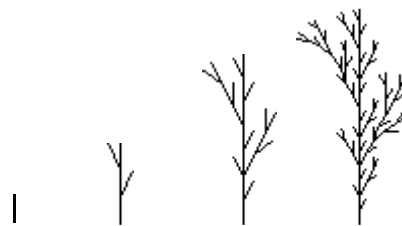


Abbildung 2: Produktionsschritte des Beispiels für geklammerte L-Systeme

2.2.3 Parametrisierte L-Systeme

Bei den Turtle-Befehlen wurde schon angesprochen, dass diese durch die Angabe eines Parameters erweitert werden können. Damit lassen sich beliebige Winkel und verschieden lange Strecken von F erzeugen. Eine Parametrisierung ist dabei auch auf der linken Regelseite möglich. Durch Anwendung einer solchen Regel wird das parametrisierte Zeichen im Wort durch die rechte Seite der Regel ersetzt. Ein arithmetischer Ausdruck für den Parameter erlaubt die Auswertung dessen Wertes. In den neu eingesetzten Zeichen ist eine Verwendung des Wertes mit Hilfe des Parameters möglich. [5]

2.2.4 Stochastische L-Systeme

Um eine Variation der erzeugten Pflanzen zu erhalten, wie sie auch in der Natur existiert, werden Abweichungen eingeführt. Dabei soll jedoch das allgemeine Aussehen der Pflanze gleich bleiben, und sich nur Einzelheiten verändern. Dazu werden die Produktionsregeln so erweitert, dass mehrere Regeln mit gleicher linker Seite existieren und jeder dieser Regeln eine Wahrscheinlichkeit zugeordnet wird. Dieser

Wahrscheinlichkeitswert gibt an, mit welcher Erwartung diese Regel ausgeführt wird. Die Summe der Wahrscheinlichkeiten zu einer gleichen linken Seite muss dabei immer eins sein.

Stochastische Regeln würden dann wie folgt aufgeschrieben werden:

$$p1 = F \rightarrow^{.33} F[+F]F[-F]F$$

$$p2 = F \rightarrow^{.33} F[+F]F$$

$$p3 = F \rightarrow^{.34} F[-F]F$$

2.2.5 Kontextsensitive L-Systeme

Die kontextsensitiven L-Systeme erweitern die D0L-Systeme in der Form, dass auf der linken Seite der Ableitungsregeln jetzt auch Worte der Länge größer eins stehen können. Diese zusätzlichen Zeichen sind Kontextbedingungen und sorgen dafür, dass eine Regel nur angewendet werden kann, wenn die Bedingung erfüllt ist. Dies gleicht den stochastischen L-Systemen, bei denen nur keine zusätzliche Bedingung die Auswahl der rechten Seite bestimmt, sondern eine Wahrscheinlichkeit.

Als Notation werden die Zeichen < und > verwendet, wobei der linke Kontext von dem zu ersetzenden Zeichen durch < abgetrennt wird und der rechte Kontext durch >. Folgendes Beispiel zur Illustration:

$$G = (\{A, B\}, BAAA, \{B < A \rightarrow B, B \rightarrow A\})$$

Die Ableitungsschritte sind dann entsprechend:

- 0. Schritt: BAAA
- 1. Schritt: ABAA
- 2. Schritt: AABA
- 3. Schritt: AAAB

Steht links von einem A ein B, so wird dieses zu einem B, und im selben Schritt wird ein B zu einem A ersetzt. Das B wandert durch das gesamte Ableitungswort und wird bei Erreichen des rechten Endes im nächsten Schritt ersetzt. Übrig bleibt ein Wort bestehend aus As.

2.3 Sensitive Wachstumsgrammatiken

Da die bisher beschriebenen L-Systeme nicht ausreichen, um die ganze Vielfalt der in der Natur vorkommenden Pflanzen und ihres Wachstumsverhaltens zu beschreiben, entwickelte Winfried Kurth die sensitiven Wachstumsgrammatiken [6].

Diese ermöglichen, im Ablauf des Wachstumsprozesses verschiedene andere Einflüsse geltend zu machen. Dazu gehören zum Beispiel der Geotropismus und der Phototropismus. Diese bestimmen die Richtungen der Gravitation beziehungsweise des Lichtes, und verändern somit das spätere Aussehen der Pflanze. So kann durch die Wirkung des Geotropismus ein Herabhängen der Äste erzeugt werden, welches sich in der Astspitze am deutlichsten auswirkt. Der Phototropismus ermöglicht dagegen ein Wachsen in Richtung des Lichtes - in den meisten Fällen ist damit die Sonne gemeint - zu simulieren.

Andere mögliche Einflüsse sind die Abhängigkeiten von der Dichte der Bewachsung und von der Pflanze zur Verfügung stehenden Licht. So kann ein Baum, welcher dicht gedrängt zwischen vielen anderen Bäumen steht, sich schlechter entwickeln. Dies liegt

einerseits an den zur Verfügung stehenden Nährstoffen, aber auch am Licht, welches ihm durch die Behinderung der anderen Bäume nicht mehr zur Verfügung steht. Weitere Details zu den sensitiven Wachstumsgrammatiken und deren Umsetzung in der Software GROGRA finden sich unter [3] und unter [7].

2.4 Relationale Wachstumsgrammatiken

Die relationalen Wachstumsgrammatiken (RGG), wie sie letztendlich eine Umsetzung in der Sprache XL finden, beruhen auf dem Prinzip der Graphersetzung [8]. Eine gute Einführung in diese Thematik findet sich unter [9].

Die Idee der RGG beruht auf den regelbasierten Ersetzungen in L-Systemen und ihrer Erweiterung durch die sensitiven Wachstumsgrammatiken. Erweitert wird dieses Konzept durch die Einführung der Graphersetzung, welche dem Benutzer eine Vielfalt von neuen Möglichkeiten zur Behandlung komplexer Modelle gibt.

Anhand des Beispiels „GameOfLife.xl“ lässt sich das Prinzip der Graphersetzung in den RGG erklären. Ein Graph besteht aus Knoten und Kanten, wobei die Knoten beliebigen Objekten entsprechen. In unserem Beispiel ist ein Knoten eine Zelle (Cell). Kanten spiegeln die Relationen zwischen den Objekten wieder. Im Fall *Game of Life* sind dies die Nachbarschaftsbeziehungen der Zellen untereinander.

Weitere wichtige Fähigkeiten der RGG sind die sogenannten *Graph Queries* und die Möglichkeit der Formulierung imperativer Code-Blöcke. Diese werden im Gegensatz zu den Graphersetzungen sequentiell abgearbeitet.

3 XL

Eine konkrete Implementierung von relationalen Wachstumsgrammatiken in eine Programmiersprache liegt in Form von XL vor. XL kombiniert die regelbasierte Programmierung von Graphgrammatiken und L-Systemen mit der imperativen und objektorientierten Programmiersprache Java [10].

Am *Game of Life*, wie es die Software GroIMP als Beispiel beinhaltet, lassen sich sehr gut einige Möglichkeiten von XL erläutern. (Die Umsetzung des *Game of Life*, wie es dem Plug-in XL4Max beiliegt, ist aus Gründen der besseren Ausführungsgeschwindigkeit nicht mit dem von GroIMP identisch.)

Der Hauptkern des Programms, wie er in GroIMP vorkommt, lässt sich auf 3 Zeilen reduzieren. Dazu gehört die Abfrage, ob 2 Zellen Nachbarn sind:

```
return (c1 != c2) && (c1.distanceLinf (c2) < 1.1);
```

Die anderen beiden Befehlszeilen dienen dazu, eine lebende Zelle unter bestimmten Voraussetzungen als tot zu markieren beziehungsweise eine tote Zelle wieder lebendig zu machen:

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

x:Cell(0), (sum ((* x -neighbour-> Cell *)[state]) == 3) ==>>
x(1);
```

Der erste Befehl zur Abfrage der Nachbarschaftsbeziehung steht in einer normalen, imperativen Java-Methode, welche innerhalb der Ersetzungsregeln aufgerufen wird, und zwar bei Verwendung der Graphen-Kante „-neighbour->“. In den beiden Ersetzungsregeln finden sich zwei weitere interessante Möglichkeiten von XL: die Überprüfung, ob ein Wert innerhalb eines bestimmten Bereiches liegt, durch „in (2:3)“, und die Zuweisung von gefundenen Objekten zu Variablen mittels „x:Cell“, wodurch dann in der Ersetzungsregel auf dieses bestimmte Objekt zugegriffen werden kann.

Die Kombination der Programmierung von Graphgrammatiken mit der Programmiersprache Java drückt sich insbesondere darin aus, dass XL eine Obermenge von Java ist. Alles, was in Java möglich ist, kann auch in eine XL-Datei geschrieben werden. Zusätzlich zu Java ist die Erstellung von Ableitungs- bzw. Ersetzungsregeln in bestimmten Blöcken möglich. Während Java-Methoden ihren sequentiellen Abarbeitungsblock mittels geschweifeter Klammern umgrenzen, werden die Regeln der XL-Grammatik von eckigen Klammern umgrenzt. Eine typische XL-Methode sieht entsprechend so aus:

```
public static void run ()
[
    Axiom ==> Sphere(5);
]
```

Wie in Java, so sind auch in XL Importe fremder Klassen möglich. Es lassen sich genauso mehrere Klassen innerhalb einer XL-Datei deklarieren. Eine Besonderheit von XL dagegen ist die Möglichkeit der Erzeugung von Modulen. Diese sind eine vereinfachte Form der Deklaration von Klassen. Es wird kein ganzer Rumpf mit

Konstruktor oder Methoden angegeben, sondern nur optionale Parameter. Ein typisches Beispiel wäre:

```
module A (int x);
```

Damit kann „A“ wie jedes andere Objekt benutzt werden, indem Instanzen dieses Typs erzeugt werden. Die Angabe des Parameters ermöglicht zusätzlich die Speicherung eines Int-Wertes. Bei der Suche nach Knoten kann dann nach Objekten vom Typ „A“ gesucht werden, die genau den gewünschten Wert besitzen. Ein Beispiel dazu:

```
Axiom ==> A(5);  
A(5) ==> Sphere(10);
```

Statt den Wert als Hilfe zur Suche zu verwenden, ist es auch möglich, mit ihm weiterzuarbeiten. So lässt sich allgemein der Wert jedes Moduls „A“ in einer Variable abspeichern und in derselben Regel benutzen, wie in diesem Beispiel:

```
Axiom ==> A(5);  
A(x) ==> Sphere(x);
```

Dies erzeugt eine Kugel (Sphere) mit Übergabe des Wertes „5“.

In den nächsten Kapiteln wird an verschiedenen Stellen genauer auf die Syntax von XL eingegangen. Zum einen bei der Implementierung des Java-Codes, wo die wichtigsten Objekttypen aufgeführt sind, zum anderen aber auch in den in Kapitel 6 vorgestellten Beispielen. Dort erfolgt auch eine genaue Erklärung des *Game of Life*.

Zusätzlich enthält auch die Hilfe zu XL4Max eine Beschreibung der Anwendung von XL. Für eine explizite Erklärung aller Möglichkeiten der XL-Sprache empfiehlt sich die *XL Language Specification*, welche neben einer Einleitung unter [11] zu finden ist.

3.1 Aufgabenstellung

Die Aufgabenstellung der Bachelorarbeit umfasst die Implementierung der Sprache XL in die 3D-Modelliersoftware Autodesk 3ds max. Bis 1999 war die Firma *Discreet Logic* der Hersteller der Software 3ds max. Autodesk kaufte in diesem Jahr die Firma auf und gründete den Unternehmensbereich *Discreet*, welcher sich fortan um die Weiterentwicklung der Produkte von *Discreet Logic* kümmerte. Im Jahre 2005 erfolgte dann die Umbenennung des Unternehmensbereiches in *Autodesk Media and Entertainment*. Im folgenden Text wird deshalb von Autodesk als Hersteller von 3ds max gesprochen.

3ds max bietet die Möglichkeit, Erweiterungen im Form von Plug-ins einzubinden. Diese Plug-ins werden für die vorliegende Version von 3ds max in Visual Studio .NET 2002 programmiert, genauer gesagt im Visual C++-Teil dieser Entwicklungsumgebung. Zur Unterstützung erhält der Programmierer ein sogenanntes *Software Development Kit*, kurz SDK. Genauere Erklärungen zur Programmierung finden sich im Kapitel 4 „Implementierung“.

Bei der Entwicklung von XL4Max war es allerdings nicht Aufgabe, einen eigenen Interpreter für die Sprache XL zu entwickeln. Stattdessen war schon ein XL-Compiler vorhanden, wie er auch in der Software GroIMP [12] zum Einsatz kommt. Der erste Hauptteil der Aufgabe bestand also darin, den 3ds max-Szenengraphen für diesen Compiler zugänglich zu machen, damit dieser auf ihm arbeiten kann. Ein Szenengraph

beschreibt den aktuellen Zustand einer Szene. Alle vorhandenen Objekte werden mittels Knoten im Graphen repräsentiert, sowie die Beziehungen zwischen den Objekten (in 3ds max Beziehungen hierarchischen Typs) als Kanten zwischen den Knoten gezeigt werden. Dieser Compiler ist wie die Software GroIMP komplett in Java programmiert. 3ds max besitzt aber keine Möglichkeit, in Java geschriebene Programme direkt auszuführen. Auch bietet das SDK in dieser Hinsicht keinerlei Unterstützung. Der zweite Hauptteil der Aufgabe war also nun die Anbindung des Java-Codes an den C++-Teil von XL4Max.

Die restlichen gestellten Anforderungen an das Plug-in können der im Anhang 9.1 hinzugefügten Aufgabenstellung entnommen werden.

3.2 Java - C++ Anbindung

Um eine Verbindung zwischen Java-Code und C++-Code herzustellen, war eine Einarbeitung in die Thematik des *Java Native Interfaces* nötig, kurz JNI genannt. Diese standardisierte Schnittstelle erlaubt gegenseitige Aufrufe von nativen Methoden bzw. Java-Methoden. Die Funktionalität ist also in beide Richtungen gegeben.

Im Fall von XL4Max werden auch beide Richtungen benötigt. So erfolgen der Aufruf des XL-Compilers und das Ausführen von XL-Methoden aus C++ heraus. Um dies zu bewerkstelligen, ist der erste Schritt, wie er für die entgegengesetzte Richtung „Java ruft nativen Code auf“ benötigt wird, das Starten einer *Java Virtual Machine* (JVM). Dies ist ein Teil der Java-Laufzeitumgebung, welche allgemein das Ausführen von Java-Programmen erlaubt. Sie wird also nicht nur im Fall der Benutzung des JNIs benötigt. Sondern immer, wenn ein Java-Programm gestartet wird, läuft dieses in einer JVM ab. Für den Nutzer ist dies nur meist nicht von Interesse und geschieht ohne ein explizites Zutun. Die JVM ist auch nicht als explizites Programm mit einer Oberfläche zu erkennen, sondern nur als Prozess „java.exe“ (so jedenfalls beim Start unter Microsoft Windows).

Nach dem Start der JVM muss auf relativ wenige Methoden der JNI zurückgegriffen werden, um dann eine Java-Methode auszuführen. Dazu zählen das Finden der entsprechenden Java-Klasse, das Heraussuchen der aufzurufenden Java-Methode dieser Klasse und danach der eigentliche Start dieser Methode. Die dafür nötigen Befehle sind im Kapitel 4 „Implementierung C++“ genauer aufgeführt. Es ist von Vorteil, die Aufrufe der JNI-Methoden mit Fehlerabfangalgorithmen auszustatten. Dies ist gerade während der Programmierung sinnvoll, da zum Beispiel im Fall einer nicht gefundenen Java-Klasse so ein Absturz der Software 3ds max verhindert wird. Der eigentliche Aufruf der Java-Methode bietet darüber hinaus die Möglichkeit, die in Java erzeugten Ausnahmen (*Exceptions*) auch auf der Seite von C++ abzufangen und auszuwerten. Dies ist in XL4Max nicht explizit nötig gewesen, aber eine nähere Einarbeitung in dieses Thema ermöglichte zumindest den wiederholten Aufruf des Java-Compilers trotz vorhergehenden Fehlers auf der Java-Seite.

Die Einbindung der anderen Richtung, also der Ausführung von C++-Methoden aus Java heraus, ist für die Arbeit des XL-Compilers am Szenengraphen von 3ds max nötig. Leider war dessen Einbindung mit mehr Tücken verbunden, da weniger hilfreiche Informationen dazu verfügbar sind. So liefen erste Versuche darauf hinaus, in Java mittels des Befehls

```
System.load("C:/Programme/3dsmax6/plugins/jvmInterface.dlu");
```

die C++-Bibliothek einzubinden. Standardmäßig werden an diese Java-Methode *Dynamic Link Libraries* übergeben, also Dateien mit der Endung „.dll“. Anscheinend

sind die kompilierten 3ds max Plug-ins gleich aufgebaut, wodurch dieser Befehl funktioniert. Probleme gab es allerdings in vielerlei Hinsicht. So ist der Aufruf der obigen Methode nicht möglich, wenn diese selbst nur Code einer Java-Methode ist, die aus C++ aus der Datei „jvmInterface.dlu“ aufgerufen wurde. (Eine Art Kreis von C++ zu Java zu C++ mit derselben C++-Methode.) Es wäre die Notwendigkeit entstanden, beide Richtungen der Java-C++-Anbindung in zwei getrennte Plug-ins zu verteilen.

Auch entsteht ein zusätzlicher Aufwand, da mehrere Dinge nötig sind, um eine native Methode auf diese Art und Weise aus C++ heraus aufzurufen. Eine gute Anleitung solcher Schritte findet sich unter [13]. Es wird ersichtlich, dass zusätzlicher Aufwand darin besteht, zu den Java-Klassen noch Header-Dateien zu generieren, die daraufhin in den C++-Code eingebunden werden müssen.

Dieser enorme Aufwand kann jedoch auf wenige Zeilen Overhead-Code reduziert werden. Es ist nur noch nötig, die nativen Methoden einmal in Java zu deklarieren. In C++ erfolgt die komplette Beschreibung der Methoden (den Code, den sie ausführen), um dann mittels Befehl „JNIEnv->RegisterNatives(...)“ diese Methoden der JVM zugänglich zu machen. Dadurch entfällt auch der Aufwand, zwei Plug-ins für 3ds max zu programmieren, da hier ohne Probleme die Arbeit in beiden Richtungen möglich ist.

Abbildung 3 zeigt schematisch, wie die JNI die Zusammenarbeit des Java- und des C++-Teils von XL4Max realisiert. Es ist zu beachten, dass dies nur den groben Aufbau widerspiegelt. Zum Beispiel ruft der XL-Compiler nicht direkt Methoden der „MaxAPI.java“ auf. Dies geschieht nur durch eigens implementierte Methoden von XL4Max, die der Compiler über verschiedene Schnittstellen indirekt anspricht.

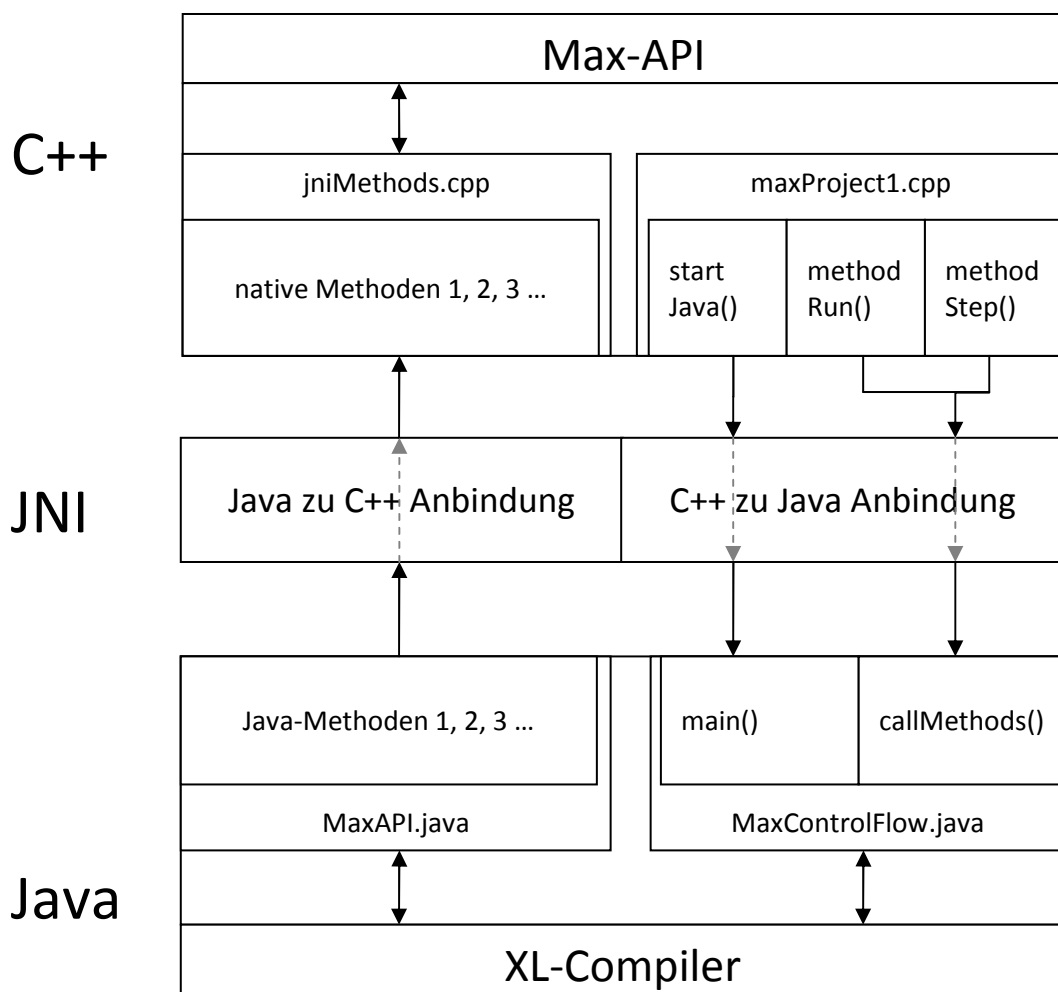


Abbildung 3: Schema der Funktionsweise des JNI in XL4Max

In der Max-API, der Programmierschnittstelle von 3ds max, werden Objekte, genauer gesagt deren Knoten im Szenengraph, durch Zeiger angesprochen. Diese Objekte müssen auch eindeutig durch Java identifiziert werden können. Erkennt der XL-Compiler zum Beispiel in der Szene eine Box, deren Kantenlänge durch eine XL-Regel auf „1“ gesetzt werden soll, so ist es selbstverständlich, dass der XL-Compiler auch genau diese Box anspricht. Also muss eine eindeutige Repräsentation der Knoten auch auf Java-Seite erfolgen. In dem Fall ist von Vorteil, dass diese Zeiger auf ein Objekt während der ganzen Laufzeit von 3ds max ihre Gültigkeit behalten. So musste nur noch ein Weg gefunden werden, diese Zeiger in den entsprechenden Java-Objekten zu speichern. Das geschieht durch eine Integer-Variable auf der Java-Seite, welche die Speicheradresse zu einem Knoten aufnimmt. Aus C++ wird also die Speicheradresse übergeben, in Java in der Variable „ref“ gespeichert und bei Zugriff auf dieses Objekt wieder an C++ zurückgegeben. Der C++-Teil von XL4Max führt dann ein *Cast* der Zahl auf einen Zeiger vom Typ „INode*“ aus. Damit ist der Zugriff auf die 3ds max-Objekte zu jeder Zeit gesichert.

3.3 XL in XL4Max

Zu betonen ist, dass XL4Max nicht alle Möglichkeiten von XL umsetzt. Dies war in der vorgegebenen Zeit nicht möglich, und auch entwickelte sich der XL-Compiler noch während der Implementierung der vorliegenden Schnittstelle weiter. Es sind jedoch alle grundlegenden Funktionen umgesetzt worden. Als Referenz sind hier die in der Hilfe aufgeführten Objekte, Methoden und Operatoren zu betrachten.

Während Operatoren wie „==>“, „: :>“ und andere durch die Syntax von XL definiert sind, war die Umsetzung zur Verfügung stehenden Objekte und Methoden nicht vorgegeben. Es musste also entschieden werden, auf welche Objekte vom 3ds max-Szenengraphen der Benutzer zugreifen kann und mit welchen Methoden er diese Objekte bearbeiten darf.

Als Objekte kamen daher alle *Standard Primitives* und *Shape*-Objekte in Frage. Jedoch beschränkte sich deren Auswahl auf die Möglichkeiten des 3ds max-SDKs. So war es zum Beispiel nicht möglich, den Geometrietyp *Plane* mit einzubauen. Die korrekte Umsetzung eines entsprechenden Beispiels aus der Hilfe des SDKs führte ohne erkennbaren Grund immer wieder zum Absturz von 3ds max.

An dieser Stelle sei noch einmal verdeutlicht, dass alle Objekte in einer 3ds max-Szene durch Knoten in einem Szenengraph repräsentiert werden. Dies ist ein Hierarchiegraph, weswegen alle vorhandenen Objekte Kinder eines anderen Knotens sind. Die Bezeichnung „Kind sein“ bedeutet also, dass von einem anderen Knoten zu ihnen eine Hierarchiekante (oder auch Nachfolgerkante genannt) existiert. Ist in der schematischen Ansicht von 3ds max keine solche Kante zu einem Knoten zu sehen, bedeutet das, dass dieses Objekt ein Kind der Szenen-Wurzel (*Root*) ist. Dieser Wurzel-Knoten ist zwar nicht sichtbar, für die Programmierung aber wie jeder andere Knoten handhabbar. Neu erstellte Objekte werden auch automatisch an die Wurzel gehangen, genauso wie Objekte, deren Vorgänger (*Parent*) gelöscht wird. Letzteres bewirkt, dass sich XL4Max anders verhält als GroIMP. Wird in XL4Max der Vorgänger eines Knotens gelöscht und dieser nicht explizit an einen anderen Knoten gehängt, so wird das Objekt gelöscht. In GroIMP dagegen existiert das Objekt noch weiter im Szenengraph, solange eine andere Kante als eine Hierarchiekante existiert. In dem Fall ist das Objekt nur nicht mehr sichtbar.

Um dem Nutzer nun ein schnelles und komfortables Arbeiten zu ermöglichen, war es unerlässlich, einen „Reset“-Button in die GUI einzufügen. Dieser sollte die Szene wieder

in den Zustand versetzen, den sie vor der Ausführung der XL-Methode besaß. Dazu müssen alle Objekte gelöscht werden, die durch XL erstellt worden sind. Andere, schon in der Szene existierende Objekte, sollten dabei unangetastet bleiben. Jetzt wäre eine Filterung anhand des Knotennamens nicht nur langsam, sondern auch nicht eindeutig. So wurde als praktikabelste Lösung der Weg über einen zusätzlichen Knoten genommen. Dieser spezielle Knoten mit dem Namen „XLRoot“ ist eine neue Wurzel für alle in XL erstellten Objekte. So werden durch ein „Reset“ alle an „XLRoot“ hängenden Objekte gelöscht. So kann der Nutzer zum Beispiel einen durch XL erzeugten Baum in eine schon vorhandene Landschaft setzen und muss für einen neuen Versuch nicht die Szene neu laden. Siehe dafür das Beispiel „Bush“.

Zu den umgesetzten Objekten hat der Benutzer die Möglichkeit, alle auch in 3ds max aufgeführten Parameter zu verändern. Dies geschieht entweder mittels Methoden, die zu jedem Objekt definiert sind, oder mittels Attributen (Properties). Die Methoden halten sich dabei an vorgegebene Namenskonventionen, so dass dem Nutzer vereinfacht wird, den Namen auch ohne Konsultation der Hilfe zu bestimmen. Die Methoden zum Auslesen von Parametern fangen mit „get“ an, die zum Setzen mit „set“. Darauf folgt der Name des Parameters, wie er in der GUI angezeigt wird. Leerzeichen werden dabei durch einen Unterstrich ersetzt. So lässt sich der Radius einer Sphere mit dem Befehl `getRadius()` auslesen, zum Setzen des Radius 1 eines Torus' wird die Methode `setRadius_1(...)` gebraucht. Der Parametertyp beziehungsweise Rückgabotyp entspricht dem in 3ds max benutzten Typ. Zur Erleichterung des Nutzers gibt es in 3ds max nur die 3 Datentypen `int`, `float` und `boolean`.

Die bei Methodenaufrufen genutzte Namenskonvention gilt auch für die Parameterumsetzung mittels Attributs. Hier entfällt selbstverständlich das „get“ und „set“, und die Beachtung von Groß- und Kleinschreibung ist nicht nötig. So lautet der Aufruf mit Property zu obigen Beispielen „Sphere[radius]“ und „Torus[radius_1] = ...“. Eine Auflistung aller Methoden und Properties mit Typangaben befindet sich in der Hilfe von XL4Max.

XL4Max beherrscht auch sogenannte Subproperties, nur ist es schwer vorherzusehen, welche der Nutzer gerne wie einsetzen würde. Diese beschränken sich deshalb auf wenige sinnvoll erscheinende Stellen. So kann zum Beispiel ein Teapot seinen Bestandteil Body mit dem Aufruf von „Teapot[teapot_parts][body] = true“ sichtbar machen.

Auch die Attribute von selbst definierten Klassen in den XL-Dateien lassen sich mittels Properties wie die von vorgegebenen Klassen verändern. Zu beachten ist nur, dass diese Attribute auf jeden Fall als „public“ deklariert sein müssen, da es sonst zu einem Fehler während der Ausführung kommt.

Eine weitere interessante Fähigkeit von XL, welche auch in XL4Max umgesetzt ist, ist die Benutzung von *Wrapper-Nodes*. Dies sind Objekte, die im 3ds max-Szenengraphen in Form von Dummy-Objekten angelegt werden. Dummy-Objekte sind am besten als Null-Objekte zu beschreiben. Sie besitzen alle Standardattribute von normalen Objekten wie Position und Rotation, sind aber nicht sichtbar. Zur besseren Handhabbarkeit werden Dummy-Objekte während der Modellierarbeit als durchsichtige Boxen dargestellt. Durch XL erstellte Dummy-Objekte sind jedoch auf Grund von Geschwindigkeitsvorteilen standardmäßig nicht sichtbar.

Wrapper-Knoten können einzelne Zahlenwerte bis hin zu ganzen Objekten in sich speichern, um diese später wieder abzurufen. Zum Einsatz kommt dies im Beispiel „Ants.xml“, wo mittels einer *Int-Wrapper-Node* gespeichert wird, vor wievielen Schritten sich die Ameise auf einer entsprechenden Zelle befand.

Es wurden in XL4Max alle relevanten Turtle-Befehle umgesetzt. Dazu lehnt sich XL4Max sehr an die Umsetzung in GroIMP an. Somit stehen die Rotationsknoten „RU“, „RH“, „RL“, „RG“, „RV“ sowie „AdjustLU“ zur Verfügung. Desweiteren natürlich „F“ zum Zeichnen, „M“ zum Überspringen eines Bereiches (entspricht dem kleinen „f“ der im Kapitel 2 angeführten Turtle-Befehle) sowie „L“, „D“ und „P“ zum Festlegen der Eigenschaften Länge, Dicke bzw. Farbe. Diese existieren in jeweils verschiedenen Varianten. Abbildung 4a zeigt eine Übersicht aller vorhandenen Turtle-Kommandos. Die Befehle „D“ und „L“ lassen noch weitere Varianten zu, wie sie in Abbildung 4b gezeigt sind. „P“ existiert noch in der Modifikation „Pl“, welche die Farbe nur für das nächste „F“ setzt.

Ein neuer, eigens entwickelter Turtle-Befehl ist RT. Dies erzeugt ein Dummy-Objekt, welches seinen Up-Vektor (auch Head-Vektor genannt, Erklärung unter [3] zu finden) in Richtung eines übergebenen Objekts dreht. Damit ist es zum Beispiel möglich, eine *Spline*-Kurve in die Richtung eines Objektes wachsen zu lassen, wobei die Stärke der Drehung eingestellt werden kann.

Weitere Turtle-Befehle, wie sie unter [1] aufgeführt und in GroIMP umgesetzt wurden, ließen sich in der vorgegebenen Zeit nicht mehr in XL4Max implementieren. Auch aufgrund ihrer Spezialisierung erschien eine Umsetzung nicht unbedingt nötig.

Turtle-Befehl	Erklärung
F(x)	Vorwärtsbewegung um Länge x
F0	Vorwärtsbewegung um durch L festgelegte Länge
M(x)	Vorwärtsbewegung ohne Zeichnen um Länge x
M0	Vorwärtsbewegung ohne Zeichnung um durch L festgelegte Länge
L(x)	Festlegen der Länge für nachfolgende F0 und M0
D(x)	Festlegen der Dicke für nachfolgende F0 und M0
P(x)	Festlegen der Farbe für nachfolgende F0 und M0
RL(x)	Rotation um die nach links gerichtete Achse (x-Achse)
RU(x)	Rotation um die nach vorn gerichtete Achse (y-Achse)
RH(x)	Rotation um die nach oben gerichtete Achse (z-Achse)
RG	Rotation strikt nach unten
RV(x)	Rotation mit Tendenz nach unten
RT(Node)	Rotation Richtung Node
Plus(x)	RU(x)
Minus(x)	RU(-x)
AdjustLU	Rotation um RH, so dass Up-Vektor nach oben zeigt

Abbildung 4a: Übersicht über Turtle-Befehle

Parameter	Erklärung
D	Aktueller Durchmesser wird auf Standardwert gesetzt (1)
D(x)	Aktueller Durchmesser wird auf x gesetzt
DAdd(x)	Aktueller Durchmesser wird mit x addiert
DMul(x)	Aktueller Durchmesser wird mit x multipliziert
DI(x)	Durchmesser für nächstes F wird auf x gesetzt
DIAdd(x)	Durchmesser für nächstes F wird mit x addiert
DIMul(x)	Durchmesser für nächstes F wird mit x multipliziert

Abbildung 4b: Übersicht über Parameter zu Turtle-Befehlen D und analog L

In GroIMP existieren verschiedene Kantentypen. Diese können jeweils verschiedene Beziehungen zwischen Objekten ausdrücken. So lassen sich neben den Hierarchiekanten „Nachfolger“ und die sogenannte „Branch“ noch weitere, eigens definierte Kanten

verwenden. In XL4Max wurden auch diese zusätzlichen Kanten implementiert, zur Vereinfachung entfällt jedoch die „Branch“-Kante. Sie ist auch als Nachfolgerkante anzusehen. Da 3ds max selbst nur Hierarchiekanten unterstützt, mussten für zusätzliche Kantentypen alle dafür nötigen Implementierungen selbst vorgenommen werden.

Gute Beispiele für die Verwendung zusätzlicher Kanten neben den Hierarchiekanten sind „Ants.xl“ und „AnimAnts.xl“. Im letzteren werden ganze vier zusätzliche Kantentypen benötigt, um alle Beziehungen zwischen den Ameisen, Plätzen und den Gedächtnisknoten zu bewerkstelligen.

Die zehn zusätzlich verwendbaren Kantentypen werden von XL mit den Namen „EDGE_0“, „EDGE_1“ usw. bereitgestellt. Bei der Erstellung einer XL-Datei bewährt es sich dabei, diese Konstanten zuzuweisen, wie zum Beispiel:

```
const int memory = EDGE_0;
```

Auf die restlichen Methoden und Objekte der Sprache XL wird, wie schon erwähnt, im Kapitel 4 „Implementierung“ näher eingegangen, so wie auch in der Hilfe zu XL4Max eine komplette Übersicht zu finden ist.

3.4 Zusätzliches in XL4Max

Neben den im letzten Kapitel schon aufgeführten Features bietet XL4Max noch zwei weitere wichtige Funktionen. Zum einen ist dies die Möglichkeit, *Shape*-Objekte zu extrudieren.

Während der Implementierungsphase fiel auf, dass die von XL erstellten *Shape*-Objekte mit ihren zugehörigen Einstellungen mit einer Dicke versehen werden können und somit renderbar sind. Aber all diese Objekte sind *durchschaubar*, in der Hinsicht, dass sie nur einen Rand besitzen, aber keine ausgefüllte Fläche. Die einzige Lösung, um dem Nutzer die Möglichkeit zu geben, diese Tatsache zu ändern, war die Anwendung von Modifikatoren. Solche in 3ds max genannten *Modifiers* sind Werkzeuge und werden jedem Objekt einzeln zugewiesen (siehe dazu [14], S.173ff). Es gibt eine Fülle solcher Modifikatoren, so dass eine Implementierung jedes Modifikators mit allen jeweiligen Optionen eine nicht zu bewältigende Aufgabe gewesen wäre. So fiel die Wahl nur auf den „Extrude“-Modifikator. Dieser ermöglicht nicht nur das Füllen der Flächen zwischen den *Spline*-Elementen, sondern auch noch eine Extrusion. Das bezeichnet in der Geometrie eine Dimensionserhöhung einer zweidimensionalen Form durch Parallelverschieben im Raum [15]. Man erhält also aus dem zweidimensionalen *Shape*-Objekt einen dreidimensionalen Körper. Ein Beispiel dafür findet sich in der Datei „Circle.xl“ mit der Methode „Circle_2“, wo die erzeugte Struktur im Gegensatz zu „Circle_1“ ein echter geometrischer Körper ist.

Die Implementierung dieses Modifikators war mit kleineren Hindernissen versehen. So fiel erst durch intensivere Tests des Plug-ins auf, dass ein Objekt, welches mit einem Modifikator versehen ist, nicht mehr zugänglich war für das Ändern seiner Attribute. Wie sich herausstellte, liegt dies an der Umsetzung von Modifikatoren in 3ds max. In den Objekteigenschaften wird ein Modifikator auf einen Modifikator-Stack zu einem Objekt gepackt. Erst in der schematischen Ansicht ist allerdings zu erkennen, dass ein Modifikator ein Kind des Objektknotens ist. Allerdings ist ein Zugriff auf diesen Modifikator-Knoten nicht mittels normaler Hierarchie-Beziehungen möglich. Wird also versucht, zu einem Knoten im Szenengraph das entsprechende 3ds max-Objekt anzusprechen, so wird das oberste Objekt im Modifikator-Stack zurückgegeben. Dagegen ist das gewünschte *Shape*-Objekt das unterste Objekt in diesem Stack. So wird beim Zugriff auf das Objekt eines Knotens, welches mittels Extrude versehen wurde, nur der Extrude-Modifikator angesprochen.

Entsprechend musste der Code also erweitert werden, dass erst nach dem eigentlichen *Base*-Objekt eines Knotens gesucht wird, bevor Attribute zu einem solchen Knoten gesetzt oder ausgelesen werden.

Das zweite wichtige Feature von XL4Max ist die Möglichkeit, Animationen zu erstellen. 3ds max ist nicht nur eine Modelliersoftware, sondern auch ein Animationsprogramm. Das bedeutet, dass sich die Zustände von Objekten ändern und diese Änderungen über die Zeit abgespeichert werden können. Somit befindet sich die Szene an sich in jeder Zeiteinheit in einem eigenen Zustand, der gerendert werden kann. Mehrere dieser gerenderten Zustände, also Bilder, lassen sich aneinander fügen und ergeben einen animierten Film.

XL4Max nutzt diese Möglichkeit der Animation aus, indem es die verschiedenen Zustände der XL-Objekte zwischen den Ableitungsschritten abspeichert. Zwischen der wiederholten Ausführung einer XL-Methode wird dazu die aktuelle Zeit um einen vorgegebenen Wert, das *Key Interval*, erhöht. Zugleich wird bei jeder Änderung einer Eigenschaft eines Objektes diese als *Key* abgelegt. Ein *Key* ist somit eine Speicherung des Zustandes eines Attributes zu einer gewissen Zeit. Zwischen zwei *Keys* mit verschiedenen Werten in den Attributen interpoliert 3ds max automatisch diesen Wert. Die Interpolation ist dabei nicht linear, sondern *Smooth*. Eine Bewegung würde also langsam anfangen, in der Hälfte zwischen den beiden *Keys* am schnellsten sein und in der Nähe des zweiten *Keys* wieder verlangsamen. Nicht immer ist jedoch eine gleichmäßige Bewegung gewünscht. So kann es durchaus sein, dass man Sprünge in einer Bewegung haben möchte. In dem Fall wird das Attribut an einem *Key* sprunghaft auf einen anderen Wert gesetzt.

Das größte Problem bei der Umsetzung der Animation bestand in der Tatsache, dass Objekte nicht zu einem bestimmten Zeitpunkt erstellt oder gelöscht werden können. Objekte können nur während des gesamten Zeitraums einer Animation existieren, oder eben gar nicht. So musste auf einen Trick zurückgegriffen werden. Objekte werden, wenn sie durch den XL-Compiler erstellt werden, in ihrem Attribut der Sichtbarkeit beeinflusst. Es wird ein Objekt bis zur aktuellen Zeit auf unsichtbar gesetzt, ab der aktuellen Zeit auf sichtbar. Im Falle, dass ein Objekt gelöscht werden soll, tritt der umgekehrte Fall ein. Ein Objekt wird bis zur aktuellen Zeit auf sichtbar gesetzt, und ab dem aktuellen Zeitpunkt auf unsichtbar. Diese Variante löst allerdings noch nicht alle vorhandenen Probleme. Hierarchiekanten können nämlich ebenfalls in der Zeit nicht verändert werden. So hat ein Objekt während der ganzen Animationszeit ein und denselben Vorfahren. Wird durch die XL-Datei jetzt die Eltern-Kind-Beziehung von Knoten verändert, so gibt es keine Möglichkeit, dies korrekt in der Animation umzusetzen. Der Nutzer hat in dem Fall die Aufgabe, die XL-Datei entsprechend anzupassen. Als Beispiel ist hierfür „AnimAnts.xl“ anzusehen. Während das Original „Ants.xl“ die Positionen der Ameisen durch Hierarchiekanten setzt (ein Objekt hat immer eine relative Position zu seinem Vorfahren), so muss dies bei „AnimAnts.xl“ anders geschehen. In dem Fall wird die Position des neuen Platzes ausgelesen und auf die Ameise übertragen.

Die Hauptintention der Implementierung von Animationen war die Möglichkeit, Ableitungsschritte zusammenzufassen. So würde sich bei komplexen Regelanwendungen, die eine lange Rechenzeit in Anspruch nehmen, die Chance ergeben, diese trotzdem flüssig hintereinander anzuschauen. Voraussetzung dafür ist wiederum, dass das System, auf dem XL4Max läuft, genug Leistungsreserven mit sich bringt. Bei der Erstellung eines Videos der fertigen Animation würde selbst dies dann keine Rolle mehr spielen - es muss nur genügend Zeit für den Render-Vorgang aufgebracht werden. Es werden nur die XL-Regeln, wobei keine Notwendigkeit besteht, dass der Nutzer anwesend ist. Nach dieser längeren Berechnungsphase erfolgt dann wie gewohnt in 3ds max das Rendern der Einzelbilder und das damit automatisch verbundene Zusammenfassen zu einem Film.

Eine ähnliche Vorgehensweise findet sich bei vielen Anwendungsgebieten in der Animationstechnik. Gerade bei aufwendigen Berechnungen durch Simulationen, wie Kleidungs- oder Flüssigkeitssimulation, wird auf diese Art vorgegangen. Leider hat XL4Max in der vorgegebenen Zeit nicht das Stadium erreichen können, in dem es eine Fehlerfreiheit der Animationsfunktion garantieren kann.

4 Implementierung

Die Anforderungen an XL4Max sind einerseits durch die Aufgabenstellung vorgegeben, andererseits musste individuell entschieden werden, was überhaupt realisierbar ist. So war eine weitgehende Einarbeitung in die Arbeitsweise des SDK von 3ds max nötig, um herauszufinden, welche als wichtig erscheinenden Funktionen auf welche Art und Weise umgesetzt werden können.

4.1 Verwendete Software

Die Aufgabenstellung lies offen, für welche Version von 3ds max das Plug-in entwickelt werden sollte. Da es nicht zu realisieren war, erst alle Versionen mit dem entsprechendem SDK auszuprobieren, um daraufhin zu entscheiden, wurde absichtlich nicht die neueste Version gewählt, sondern die schon länger auf dem Markt befindliche Version 6. Es ist zu vermuten, dass viele Anwender nicht die neueste Version nutzen, schon allein wegen des Preises. Außerdem erschien der Anteil neuer Funktionen, die die zur Zeit der Programmierung aktuelle Version 8 enthält, für XL4Max nicht relevant.

Wie hinterher festzustellen war, lässt sich XL4Max sogar ohne Neukompilierung mit der Version 8 von 3ds max betreiben. Allerdings war es nicht möglich, jede Funktion so ausgiebig zu testen wie unter 3ds max 6. Es kann daher unter Umständen vorkommen, dass sich eine interne Methode von 3ds max in einer neueren Version anders verhält.

Die Implementierung erfolgte in zwei verschiedenen Software-Produkten. Die Programmierung des 3ds max-Plug-ins an sich erfolgte in Microsoft Visual Studio .Net 2002. Als Programmiersprache kam hier C++ zum Einsatz. Dies ist durch das Max-SDK fest vorgegeben. Auch hier war festzustellen, dass eine Programmierung von XL4Max für einer neueren Version von 3ds max keinen Vorteil bringen würde, da diese auch keine aktuellere Version des Visual Studio .NET unterstützen.

Die Java-seitige Implementierung von XL4Max erfolgte unter Zuhilfenahme der Entwicklungsumgebung Eclipse. Diese ermöglichte eine schnelle und komfortable Arbeit auf bis dahin ungewohntem Terrain.

4.2 Visual Studio .Net und Max-SDK

3ds max liefert zu jeder Version ihrer Modelliersoftware ein entsprechendes *Self Development Kit* (SDK) mit. Dieses wird separat installiert und liefert neben den eigentlichen Header-Dateien, die unter C++ zum Entwickeln eingebunden werden müssen, noch reichlich Beispiele, einen Wizard und eine Hilfe mit. Die Hilfe ist zwar umfangreich und erklärt als Referenz alles, was während der Programmierung verwendet werden kann, hat aber leider in sich nur sehr wenige Querverweise. So war man ständig gezwungen, während der Suche nach bestimmten Funktionen auch wirklich jeden Suchtreffer abzarbeiten, um auch kein wichtiges Detail zu verpassen, was vielleicht nur an anderer Stelle erwähnt wurde. Zugute halten muss man allerdings, dass das Sparks-Archive enthalten ist. Sparks ist die Bezeichnung für das Developer-Programm von Autodesk, und das Sparks-Archive ist eine Sammlung von Forenbeiträgen. Andere Entwickler, die schon Fragen bzgl. der Entwicklung von Plug-ins für 3ds max hatten, stellten diese in einem speziell dafür bereitgestellten Forum und erhielten daraufhin Antworten direkt von den Entwicklern von Autodesk oder anderen, gut versierten Programmierern. So fand man in diesem Archiv eine Menge von Antworten auf Fragen, die sich immer wieder auftraten. Auch wurden in diesen Bereichen viel mehr Beispiele für Umsetzungen angeführt als es die Hilfe an sich tut.

Der Wizard des SDKs ist sehr hilfreich, da er beim Erstellen eines neuen Projektes unter Visual C++ alle anfallenden Arbeiten abnimmt. So wird man neben dem Namen des Plug-ins auch gefragt, von welcher Art das Plug-in sein soll. In 3ds max unterscheidet man eine Vielzahl von Plug-ins, sozusagen für jede Art von Element in der Software ein eigener Plug-in-Typ. Dies kann, um einige Beispiele zu nennen, entweder eine eigene Kamera sein, ein Licht, ein neues Patch- oder NURBS-Objekt, aber auch ein Utility, so wie XL4Max es ist. Entsprechend erstellt der Wizard dann die nötigen Header- und Klassendateien und trägt automatisch die benötigten *include*-Dateien mit ein.

Zu den Beispielen wäre zu sagen, dass diese zwar in großem Umfang vorhanden sind, aber leider nur sehr unzureichend dokumentiert. Durch die Menge verliert sich schnell die Übersicht, und die einzelnen Beispiele sind selten auf ein bestimmtes Problem spezialisiert, denn sie versuchen immer gleich eine große Menge abzudecken. Dadurch war es fast unmöglich, durch direkte Suche Hinweise für eine Umsetzung ähnlicher Probleme zu finden wie denen, die sich bei der Programmierung von XL4Max auftraten.

Zur Arbeit unter Microsoft Visual Studio .NET braucht nicht viel gesagt zu werden. Tatsache scheint aber zu sein, dass es bei Produkten von Autodesk immer Probleme beim Debuggen gibt. Weniger bei dem Vorgang direkt, sondern mehr beim Start des integrierten Debuggers von Visual Studio .NET. So war es schon bei anderen realisierten Projekten, der Entwicklung von Plug-ins für Autocad, nicht möglich, die Funktion „Starten“ zu benutzen. Es ist auch bei 3ds max wieder nur möglich gewesen, dieses entweder direkt durch Ausführen von „3dsmax.exe“ oder mittels „Starten ohne Debuggen“ zum Laufen zu bewegen. Danach ist es möglich, den Debugger in das laufende Programm einzuklinken - unter der Voraussetzung, diese Möglichkeit wurde korrekt in den Projektoptionen gesetzt. Leider findet diese Vorgehensweise keine Erwähnung in der Hilfe des SDKs. Erst eine Suche in diversen Foren brachte den entscheidenden Hinweis für eine leichtere Programmierarbeit.

4.3 Eclipse

Im Gegensatz zu Visual Studio .NET war die Arbeit unter der Entwicklungsplattform, die für den Java-Teil von XL4Max eingesetzt wurde, bis dahin ungewohnt. Auch Kenntnisse in der Java-Programmierung beschränkten sich bis zum Beginn der Bachelorarbeit auf kleine Ein-Klassen-Programme. Von Dingen wie *Packages* und *SVN* hatte man bis dahin nur gelesen und gehört, aber diese nie angewandt.

Um die Schnittstelle zwischen dem XL-Compiler und 3ds max schaffen zu können, war es zunächst nötig, diesen Compiler in das eigene System einzubinden. Mittels eines extra eingerichteten Zugangs zum SVN-Server des GroGra-Projektes war das schnell geschehen. Leider scheint das dafür zuständige Subclipse-Plug-in noch nicht ganz ausgereift zu sein, denn ein Absturz des Systems während eines Updates führte dazu, dass danach keine Arbeit mit dem SVN-Server möglich waren. Es blieb nichts anderes übrig, als den kompletten Java-Code zu löschen und ein vollständiges *Checkout* durchzuführen.

Auch Eclipse an sich scheint sich ab und zu nicht so zu verhalten, wie man es erwarten würde. So passierte es nie, dass das automatische *Build* nach einem kompletten *Checkout* auch wirklich alle Projekte sauber kompilierte. Erst ein Umschalten auf manuelles *Build* und dessen mehrfache Ausführung hintereinander führten schließlich zum gewünschten Ergebnis - einem komplett kompilierten XL-Compiler.

Der wohl schwierigste Fakt bei der Programmierung lag darin, dass am Anfang keine Möglichkeit bestand, den ausgeführten Java-Code zu debuggen. Durch das Durchforsten verschiedener Foren und Dokumentationen war herauszufinden, dass es jedoch möglich sein sollte, sich in eine laufende JVM einzuklinken und dort ein *Debugging* durchzuführen. Dies bedarf jedoch spezieller Aufrufparameter der JVM, welche beim Aufruf eines Java-Programms von der Konsole wie folgt lauten:


```
Java Program -Xdebug
-Xrunjdw:transport=dt_socket,server=y,suspend=n
```

Die Übergabe der Parameter an die JVM beim Aufruf aus einem C++-Programm heraus erfolgt auf anderem Weg. Es werden mehrere Objekte der Klasse „JavaVMOption“ erstellt, sinnvollerweise in einem Array. Und jedem dieser Objekte wird dann ein String mit entsprechendem Parameter zugewiesen. Dieses Array wird der Methode dann zum Start der JVM übergeben. Die Zuweisung eines Parameters sei hier stellvertretend für alle anderen angeführt:

```
options[optionsCount++].optionString = "-Xdebug";
```

Diese an sich korrekte Zuweisung wollte jedoch keinen Start der JVM ermöglichen. Erst relativ spät wurde dann durch Zufall im Internet eine Lösung dieses Problems gefunden. Die obige Zeile muss wie folgt geändert werden:

```
options[optionsCount++].optionString = strdup ("-Xdebug");
```

Es wurde nicht dazu erwähnt, warum es auf diese Weise funktioniert. Denn auch in der Platform-SDK von Microsoft wird die Notwendigkeit von `strdup(const char*)` an dieser Stelle nicht klar.

Nichts desto trotz erleichterte die Möglichkeit des *Debuggens* die restliche Arbeit unter Java sehr. So wurden erst mittels der Möglichkeit, Variablen während des Programmlaufs beobachten zu können, mehrere Fehler in der Implementierung von Algorithmen gefunden. Es passierte während der Programmierung leider oft, dass vergessen wurde, inwieweit Unterschiede zwischen Java und der sonst gewohnten Programmiersprache Delphi (oder auch Objekt Pascal genannt) existieren. So werden in Java bei der Übergabe von Objekten in den Methodenparametern die Objekte anhand ihrer *Referenz* übergeben. In Delphi wird stattdessen eine Kopie des Objektes an die Methode übergeben. Dies hatte zum Beispiel im Fall der Turtle-States eine erst nicht erklärbare Auswirkung. Der erste entwickelte Algorithmus beruhte darauf, dass beim Setzen des Turtle-States eine Kopie dessen erzeugt wird und dann der originale Zustand verändert werden kann. Da dies aber nicht passierte, war der Fehler nur noch intensiveren Tests zu finden.

Nachdem XL4Max in einem recht gut funktionierendem Stadium vorlag, war allerdings festzustellen, dass die Geschwindigkeit bei der Ausführung sehr zu wünschen übrig ließ. Um eine Optimierung der Laufzeit vorzunehmen, wurde ein sogenanntes *Profiling* mittels speziellen Werkzeugs durchgeführt. Bei der Suche nach einem passenden Hilfsmittel wurde der Artikel „Java Application Profiling using TPTP“ [16] gefunden. Dieser handelt von einem Programm namens TPTP, welches direkt in Eclipse in Form eines Plug-ins eingebunden werden kann. Von [17] kann man sich die dafür notwendigen Komponenten besorgen. Nach erfolgreicher Installation wurde versucht, herauszufinden, wo die Schwachstellen in dem bis dahin vorliegenden Code von XL4Max liegen. Die erste Feststellung lag jedoch darin, dass aktiviertes *Profiling* die Ausführung eines XL-Beispiels um den Faktor 5-10 verlangsamt. Im Großen und Ganzen lässt sich aber dazu sagen, dass auf diesem Weg schon die eine oder andere Optimierung durchgeführt werden konnte, manche aber zu Gunsten der Stabilität auch wieder rückgängig gemacht werden mussten. So konnte die Methode `supplyNodes(...)` in der `Graph.java` zwar insofern umgeschrieben werden, dass eine deutlich schnellere Abarbeitung stattfand, dies führte aber bei Nichtbeachtung bestimmter Regeln während der Arbeit in 3ds max zu Abstürzen. So fiel die Wahl auf eine höhere Stabilität des Plug-ins.

Durch die später hinzugefügte Möglichkeit der Erstellung von Byte-Code durch den XL-Compiler wäre dieser Geschwindigkeitsvorteil sowieso wieder relativiert worden, wie spätere Tests ergaben.

4.4 Code-Erklärung

Bei der Erstellung der Quelltexte wurden gewisse Konventionen eingehalten. Diese entsprechen im Allgemeinen den typischen *Java Code Conventions*, wie sie unter [18] einzusehen sind. Teilweise wurde auch ein eigenes System für den Aufbau der Quelltexte festgelegt, welches ein maximales Maß an Übersichtlichkeit verschafft.

Zu den Konventionen gehört zum einen die Vereinheitlichung der Variablen- und Methodennamen. Diese sind grundsätzlich klein geschrieben, und nur bei zusammengesetzten Wörtern ist der Anfangsbuchstabe des Teilworts ein Großbuchstabe. Diese Zusammensetzung erfolgt dabei ohne Zuhilfenahme von Unterstrichen und soll für ein besseres Verständnis der Funktion der Variable oder Methode sorgen. Anders ist es bei der Verwendung von Konstanten, welche komplett groß geschrieben werden und in denen eine Abgrenzung von Teilwörtern mittels Unterstrichen erfolgt.

Code-Blöcke sind selbstverständlich mittels eines Tabulators eingerückt. Die öffnende geschweifte Klammer eines Code-Blocks steht auch in C++, wie es in Java üblich ist, noch in der Zeile mit dem zugehörigen Methodennamen, welcher den Block einleitet. Die Klammern, die zu einer Methode die Parameter enthalten, werden der Übersichtlichkeit wegen auch mit einem Leerzeichen vom Methodennamen getrennt.

Der gesamte Quelltext ist mit Kommentaren durchsetzt. Diese finden sich vor den meisten wichtigen Methoden, um deren Funktion näher zu beschreiben, insbesondere in der C++-Klasse „jniMethods.cpp“, da hier die eigentliche Schnittstelle zu 3ds max liegt. Alle der aufgeführten Methoden tätigen entweder direkte Veränderungen am Szenengraphen oder lesen diesen aus. Deswegen ist es hier besonders wichtig, genau zu wissen, was welche Methode bewirkt. Da diese Methoden nicht direkt vom Nutzer aufgerufen werden, gilt dies nur für den Programmierer. Anders in der Java-Klasse „MaxLibrary.java“, da die hier aufgeführten Methoden dem Nutzer direkt zugänglich sind. Genauere Erklärungen dieser Methoden finden sich deshalb neben den Kommentaren im Quelltext auch in der XL4Max-Hilfe. Restliche Kommentare beschränken sich auf die grobe Erklärung von Klassen oder Variablen.

Erwähnenswert ist in diesem Zusammenhang die Bedienerfreundlichkeit von Eclipse. Hier können Methoden mit speziellen Kommentaren versehen werden, welche mit „/**“ eingeleitet werden. Es werden neben der Funktion der Methode auch Erklärungen zu den Übergabeparametern sowie dem Rückgabewert angegeben. Erfolgt dann eine Angabe dieser Methode an anderer Stelle, so wird dem Programmierer die komplette Erklärung geboten. Dies erleichtert gerade bei überladenen Methoden sehr, sich die eigentliche Aufgabe der Funktion wieder ins Gedächtnis rufen zu können. Auch bei Nutzung fremder Methoden bringt dies einen großen Vorteil mit sich.

Andere Java-übliche Konventionen, wie zum Beispiel mit einem Großbuchstaben beginnende Klassennamen, sind in C++ nicht durchweg eingehalten worden. In dem angesprochenen Fall wird selbst vom Wizard des 3ds max-SDKs keine Richtlinie eingehalten.

4.5 C++

In diesem Kapitel wird genauer auf den Implementierungsteil von XL4Max auf der Seite von C++ eingegangen. Dabei erfolgt eine Beschränkung auf die selbst erstellten Klassen

und auf die veränderten Klassen, die das 3ds max-SDK erstellt. Dazu gehören die Klassen `maxProject1`, `jniMethods`, `jniInterface` und `xlEditor`.

Verzichtet wird dabei jedoch auf eine Erklärung jeder einzelnen Variable oder Methode. Stattdessen erfolgen nur die Vorstellungen der relevantesten Methoden und ihrer Funktion sowie ein Ansprechen der Funktionen, deren Implementierung die größten Probleme bereitete.

4.5.1 `maxProject1.cpp`

„`maxProject1.cpp`“ ist die eigentliche Hauptklasse auf Seiten von C++. Sie ist die einzige der angesprochenen Klassen, welche durch den SDK-Wizard erstellt wurde. Sie enthält somit alle wichtigen Angaben, die 3ds max von XL4Max benötigt, um es als Plug-in zu integrieren.

Diese Datei enthält neben der eigentlichen Beschreibung der Klasse `maxProject1` noch eine Beschreibung der Klasse `maxProject1ClassDesc`. Diese besteht nur aus wenigen Zeilen Code und ist dazu vorhanden, um 3ds max Details wie die Art des Plug-ins und den Namen mitzuteilen, aber auch den Zeiger auf das Plug-in-Objekt.

Die Klasse `maxProject1` beinhaltet alle GUI-Elemente von XL4Max. So wurden erst mittels der Ressourcenansicht von Visual C++ alle benötigten Elemente des *Rollouts*, dem eigentlichen Dialogfeld des Plug-ins, erstellt. Dazu gehören die Buttons zum Starten des XL-Editors, Buttons zum Kompilieren des XL-Codes und zum Zurücksetzen der Szene sowie Checkboxes und sogenannte *Spinner-Controls* zum Setzen bestimmter Optionen. Diese Einstellungen betreffen hauptsächlich das Verhalten des Plug-ins bei der Animation. Alle GUI-Elemente sind mit einer eindeutigen ID versehen, mit Hilfe derer ein eindeutiger Zugriff auf dieses Element aus dem Quelltext möglich ist.

Wird dann ein Element tatsächlich genutzt, zum Beispiel durch das Drücken eines Buttons, so wird dies in der Window-Prozedur verarbeitet. Diese wird bei einem Ereignis von Windows heraus aufgerufen. In dieser Prozedur erfolgt eine Einteilung der Ereignisse nach bestimmten Kriterien. Dazu wird nachgeschaut, ob es sich um eine Fensteraktion, eine Aktion mit der Maus oder ein Kommando handelt. Zum letzten Fall gehört auch die Nutzung der GUI. Danach wird gefiltert, um welches Element es sich gehandelt hat, und es kann eine für dieses Element entsprechende Funktion aufgerufen werden.

Im Fall des Rollouts mit dem Titel „XL4Max“ handelt es sich bei der Window-Prozedur um das Callback „`maxProject1DlgProc`“. So ruft der Button „Help“ mit der ID „`IDC_OPEN_HELP`“ die Methode `themaxProject1.openHelp()` auf. Es müssen nicht unbedingt Methoden aufgerufen werden. So führt ein Klick auf die Checkbox „Animate“ mit der ID „`IDC_CHECK_ANIMATE`“ gleich eine ganze Reihe mehrerer Befehle aus. Dazu gehört das Setzen der Variable „`animate`“ mittels Befehl

```
animate = SendMessage (hWndAnimate, BM_GETCHECK, NULL, NULL);
```

Die `SendMessage`-Methode ist eine WinAPI-Methode, welche Befehle an Fenster oder deren Elemente schickt. In diesem Fall wird das Handle zur Checkbox (`hWndAnimate`) und der Befehl `BM_GETCHECK` übergeben. Dieser fragt den Status einer Checkbox oder eines Radio-Buttons ab. Der Ergebniswert kann dann direkt in einer Integer-Variable gespeichert werden. Dies bedeutet eine Null, wenn der Status „nicht aktiviert“ ist, und einen Wert ungleich Null, falls die Checkbox aktiviert ist. Zusätzlich erfolgt mit der Checkbox „Animate“ noch das Ausgrauen oder entsprechend verfügbar Machen der Checkboxes „Smooth Animation“ und „Smooth Creation and Deletion“. Diese sollen für den Nutzer nur anzuklicken sein, wenn überhaupt eine Animation gewünscht ist.

Das Callback für das Rollout „XL-Methods“ arbeitet nach dem gleichen Prinzip. Zu beachten ist nur, dass die Buttons in diesem Rollout nicht vom Start des Plug-ins existieren. Diese werden erst mittels der Methode `createMethodButtons(...)` erstellt. Eine Schwierigkeit ergibt sich hier in der Zuweisung des Buttons zu der zugehörigen XL-Methode.

Als optimale Lösung erwies sich hier die Zuweisung eines Integer-Wertes zu jedem Button. In der Methode `createMethodButtons(...)` wird dazu eine Variable `buttonCount` hochgezählt. In dieser Variable steckt eine eindeutige Nummer, die jedem Button beim Erstellen mittels `CreateWindow(...)` übergeben wird. Der Parameter `hMenu` dieser Methode ist dafür geeignet, denn dessen Inhalt wird dann später an die Callback-Prozedur des Rollouts übergeben. Dort wird dann mittels kleinerer Berechnung herausgefiltert, zur wievielten XL-Methode der Button gehört. Dieser Wert wird dann entsprechend, ob der Button „Step“ oder „Run“ gedrückt wurde, der Methode `methodStep(...)` bzw. `methodRun(...)` übergeben, zu denen später noch eine Erklärung kommt.

Während des Starts von 3ds max werden schon einige vom Plug-in benötigte Variablen initialisiert. Der Hauptanteil der Initialisierungsphase liegt allerdings beim Start von XL4Max, also beim ersten Öffnen des Rollouts. In der Methode `BeginEditParams(...)` erfolgt die eigentliche Erstellung der Rollouts anhand der in Visual C++ erstellten Dialoge. Es werden außerdem GUI-Elemente zu Objekten zugewiesen, mit denen dann im Quelltext gearbeitet werden kann. So erfolgt mittels des Befehls

```
iSpinDelay = SetupFloatSpinner (hPanel, IDC_SPIN_DELAY,
                                IDC_EDIT_DELAY, 0.0f, 10.0f, delay, 0.05f);
```

die Initialisierung des für 3ds max typischen *Spinner-Controls*. `hPanel` ist das Handle zum Rollout, „IDC_SPIN_DELAY“ die ID zu den Spinner-Kontrollen (eine Zusammensetzung zweier Buttons, auf einem ein Pfeil nach oben, auf dem anderen ein Pfeil nach unten) und die ID „IDC_EDIT_DELAY“ ist die des zugehörigen Edit-Feldes, in dem der eigentliche Wert angezeigt wird. Die vier Zahlenwerte stehen nacheinander für Bereichsanfang, Bereichsende, voreingestellter Default-Wert sowie die Schrittweite. Man erkennt, dass es sich um ein *Spinner-Control* für Float-Werte handelt. 3ds max liefert noch ein Äquivalent für die Nutzung von Integer-Werten. Die Zuweisung erfolgt dann an das Objekt `iSpinDelay`, über welches dann ein Zugriff auf verschiedene Methoden möglich wird. Dazu gehört zum Beispiel das Auslesen und Setzen des Wertes, die Neu-Festlegung des Bereiches oder sogar die Zuordnung eines anderen Edit-Feldes. Alle Methoden sind ausführlich in der Hilfe des 3ds max-SDKs beschrieben.

In der Methode `Init(...)`, welche noch vor `BeginEditParams(...)` aufgerufen wird, erfolgt die Initialisierung der Buttons zu Objekten, sowie das Setzen aller Default-Werte auf eventuell gespeicherte Werte in der zum Plug-in gehörenden Ini-Datei „XL4Max.ini“. Diese Datei liegt nach korrekter Installation im XL4Max-Verzeichnis, und kann mittels vorhandener Win-API-Befehle bearbeitet werden. Das Auslesen der zuletzt benutzten XL-Datei erfolgt mittels Befehl:

```
GetPrivateProfileString ("Options", "XLFile", DEFAULT_XL_FILE,
                        fileName, sizeof (fileName), iniFile);
```

Ini-Dateien sind immer nach demselben Schema aufgebaut. Die Sektion ist in diesem Fall „Options“ und der gesuchte Schlüssel „XLFile“. Der Wert wird in die Variable `fileName` geschrieben; falls nicht vorhanden, wird der Wert der Konstante `DEFAULT_XL_FILE` genommen. Die Angabe, welche Ini-Datei ausgelesen werden soll,

bestimmt die Variable „iniFile“. Diese setzt sich aus dem Arbeitsverzeichnis von 3ds max, dem Verzeichnis von XL4Max und dem Dateinamen der Ini-Datei zusammen.

Entsprechend wird beim Schließen von 3ds max verfahren. Alle Optionen, die sich als Werte in Variablen wieder finden, werden in die Ini-Datei geschrieben. Dem Beispiel beim Auslesen folgend ist der Befehl zum Schreiben wie folgt:

```
WritePrivateProfileString ("Options", "XLFile", fileName,
    iniFile);
```

Bei den Zahlwerten, welche auch für die booleschen Werte der Checkboxen zum Einsatz kommen, ist es allerdings nötig gewesen, diese vorher in einen String zu schreiben. Dies ist notwendig, da kein Befehl zum Schreiben solcher Werte in Ini-Dateien existiert. Um also den Wert des *Run Delays* zu speichern, sind folgende Zeilen Code nötig:

```
char runDelay[20] = "";
sprintf (runDelay, "%f", delay);
WritePrivateProfileString ("Options", "Run_Delay", runDelay,
    iniFile);
```

Es erfolgt eine Umschreibung des Float-Wertes in ein Character-Array von ausreichender Größe.

Beim Schließen von 3ds max erfolgt außerdem das Beenden der Java Virtual Machine. Der Start der JVM erfolgt allerdings erst beim Öffnen von XL4Max. Grund dafür ist, dass ein Start der JVM schon beim Start von 3ds max unnötig Speicher verbrauchen würde. Der Zeitpunkt zum Beenden ist auch aus gutem Grund gewählt. Geschähe dies beim Schließen des Plug-ins, in der Absicht, Speicher zu sparen, so wäre es dann nicht mehr möglich, eine neue JVM zu starten. Es existiert zwar der Befehl `pJVM->DestroyJavaVM ()`, aber dieser funktioniert nur teilweise. Es wird nicht die komplette JVM geschlossen, so dass es nicht möglich ist, die JVM zu re-initialisieren, oder eine neue JVM zu starten. Dies ist eine bekannte Einschränkung, wurde aber von Seiten Suns nicht in neueren Versionen behoben.

Die Buttons „Reset“ und „Compile“ sind mit den Methoden `resetScene(...)` beziehungsweise `startJava(...)` verknüpft. Der Button „Reset und Compile“ löst dementsprechend beide Methoden aus. Er dient hauptsächlich dazu, die Arbeit mit XL4Max zu vereinfachen.

`resetScene(...)` kann dabei in zwei verschiedenen Weisen aufgerufen werden. Erfolgt der Aufruf mit „`deleteScene = false`“, so werden nur geringfügige Änderungen am Szenengraphen von 3ds max vorgenommen. Wird dagegen ein Aufruf mit „`deleteScene = true`“ vorgenommen (wie es beim Drücken von „Reset“ der Fall ist), so wird der Szenengraph von 3ds max in eine Art gesäuberten Zustand versetzt. Je nachdem, ob ein Knoten mit der Bezeichnung „XLRoot“ existiert oder nicht, werden alle Kindsknoten von ihm gelöscht, oder es wird ein neuer Knoten „XLRoot“ erzeugt. An ihn wird ein anderer Knoten „Axiom“ gehängt, welcher für die XL-Beispiele von elementarer Bedeutung ist.

Die Methode `startJava(...)` dagegen startet den eigentlichen XL-Compiler. Dazu müssen aber vorher einige Bedingungen erfüllt sein, wie eine gestartete JVM, dass die nativen Methoden registriert sind (dazu später mehr), und dass eine XL-Datei zur Übergabe an den Compiler erstellt und ausgewählt wurde. Ist dies der Fall, so wird als erstes das Rollout „XL-Methods“ von allen bisherigen Methodeneinträgen mit ihren Buttons gesäubert und wieder in den Zustand gesetzt, den es beim Start des Plug-ins hatte. Dies ließ sich sehr leicht mit folgendem Befehl bewerkstelligen:

```
EnumChildWindows (hPanelButtons, EnumChildProc, NULL);
```

EnumChildWindows(...) ist eine Methode, die alle Elemente von hPanelButtons, dem entsprechenden Rollout, durchgeht und auf ihnen die Window-Prozedur EnumChildProc anwendet. Die entsprechende Window-Prozedur enthält dabei nur den Befehl DestroyWindow (hWndChild), welcher das übergebene Element löscht.

Um jetzt die entsprechende Methode zum Start des Compilers aufzurufen, muss erst die entsprechende Java-Klasse gefunden werden. Dies erfolgt mittels

```
jclass cls = pJEnv->FindClass  
    ("de/grogra/ext/max/model/MaxControlFlow");
```

was im erfolgreichen Fall eine Referenz auf die Klasse „MaxControlFlow“ liefert. Diese befindet sich, wie zu sehen ist, in dem Package „de.grogra.ext.max.model“. Daraufhin muss die „main“-Methode dieser Klasse ermittelt werden:

```
jmethodID mid = pJEnv->GetStaticMethodID (cls, "main",  
    "([Ljava/lang/String;)V");
```

An die aufgeführte Methode wird die Klasse, die die Methode enthält, der Methodenname sowie deren Parameter übergeben. Da Methoden überladen sein können, ist die Angabe von Parametern zwingend, um die gewünschte Methode anzusprechen. Der genaue Aufbau einer solchen Signatur ist unter [19] nachzuschauen. Hier sei nur gesagt, dass der Übergabeparameter „(Ljava/lang/String;)“ ein String-Array bedeutet, während der Rückgabeparameter „V“ für „Void“ steht und ausdrückt, dass nichts zurückgegeben wird. Vor dem Aufruf der Methode müssen die Übergabeparameter in einer für Java verständlichen Form bereit stehen. Dies bedeutet, dass das String-Array in einem Datentyp vorliegen muss, der durch die JNI bereitgestellt wird. Da es aber kein entsprechendes Array gibt, muss auf das *Object*-Array zurückgegriffen werden. Um ein solches Array zu füllen, sind folgende Zeilen Code nötig:

```
jstring jstr = pJEnv->NewStringUTF (fileName);  
jclass jStringClass = pJEnv->FindClass ("Ljava/lang/String;");  
jobjectArray argsS = pJEnv->NewObjectArray (1, jStringClass,  
    jstr);
```

In der ersten Zeile wird der Pfad zu der XL-Datei in einen Java-String gepackt. In der Variable jStringClass wird gespeichert, welchen Typ von Objekten das *Object*-Array aufnehmen soll. Die Erzeugung erfolgt durch den letzten Befehl mit der Übergabe, wie viele Parameter es sein sollen, von welcher Klasse sie sind und welches das Initialelement ist. Bei einer Array-Größe größer eins würden alle Elemente des Arrays mit diesem Wert initialisiert werden. In dem Fall müsste nachträglich noch jedes Element einzeln gesetzt werden. Dies ist aber an dieser Stelle nicht nötig gewesen, und der Aufruf der festgelegten Methode erfolgt mittels

```
pJEnv->CallStaticVoidMethod (cls, mid, argsS);
```

Der Ablauf des Aufrufens von Java-Methoden aus C++ ist dem gezeigten Beispiel immer ähnlich. Erst wird die Klasse bestimmt, dann die Methode gesucht und diese dann mit festgelegten Parametern gestartet. Da ein Aufruf allerdings an verschiedenen Stellen fehlschlagen kann - die Klasse oder die Methode existieren nicht oder bei der Ausführung des Java-Codes kommt es zu einem Fehler - ist an all diesen

fehlerträchtigen Stellen ein „pJEnv->ExceptionClear ();“ eingebaut. Dies hat gerade zur Entwicklungszeit viel Ärger erspart, da nach einer auftretenden *Exception* der Fehlerspeicher geleert wird. Damit wird ein neues Aufrufen der JNI oder Java-Methode möglich.

Gerade beim ersten Kompilieren einer XL-Datei braucht der Computer sehr viel Zeit. Da durch „Compile“ auch gleich eine eventuell in der XL-Datei vorhandene Init-Methode aufgerufen wird, dauert das Kompilieren im ungünstigen Fall mehrere Sekunden. Um den Nutzer dabei nicht in der Ungewissheit zu lassen, ob vielleicht etwas schief gegangen ist, erscheint während der ganzen Zeit ein kleiner Hinweis im Vordergrund von 3ds max. Außerdem werden sämtliche Eingaben gesperrt.

Ähnliches passiert bei der Methode `methodRun`. Hier wird eine Progress-Bar erzeugt, welche zwar keinen zeitgesteuerten Ablauf darstellen kann, aber alle sonstigen Eingaben blockiert und einen Fortschritt darstellt. Nach jeder Ausführung der Methode erfolgt eine Erhöhung des Balkens um zehn Prozentpunkte. Eine solche Progress-Bar in der Methode `methodStep` wäre zwar auch wünschenswert gewesen, da auch ein einzelner Aufruf einer XL-Methode sehr lange Zeit in Anspruch nehmen kann, dies würde aber bei schnell ausgeführten Aufrufen zu einem Flackern der GUI von 3ds max führen.

Die Oberfläche von 3ds max zu blockieren und den Benutzer mit einem Hinweis darauf aufmerksam zu machen, dass das Plug-in gerade arbeitet, sollte ursprünglich mittels eigens erstelltem Dialog geschehen. Da dieses aber auf der WinAPI beruhte, gab es damit verbundene Probleme. Das Dialogfeld wurde nicht richtig dargestellt, wenn es vom selben Thread wie dem von XL4Max erstellt wurde. Auch wäre es in dem Fall gar nicht möglich gewesen, einen „Cancel“-Button anzusprechen, da nicht auf Eingaben reagiert wird.

Der nächste Versuch, das Dialogfeld in einem extra Thread aufzurufen, scheiterte daran, dass der neue Thread den alten nicht stoppen konnte. Es wurde keine Möglichkeit gefunden, den Ablauf von `methodRun` zu unterbrechen. Schließlich blieb noch die Idee, Java in einem extra Thread zu starten und das Dialogfeld vom eigentlichen Thread des Plug-ins zu erzeugen. Dies funktionierte zwar prinzipiell (das Dialogfeld wurde korrekt geschlossen), aber es bereitete 3ds max große Probleme, den Szenengraphen zu verändern. Ein Versuch auf diese Art endete im kompletten Absturz von 3ds max.

Die Vorgehensweise der beiden Methoden `methodStep(...)` und `methodRun(...)` ist im Grunde die gleiche. Ihnen wird die Nummer der aufzurufenden XL-Methode übergeben, welche sie als Argument direkt an die Java-Methode weitergeben. `methodRun(...)` führt die entsprechende Java-Methode zusätzlich in einer Schleife aus, und zwar solange, bis der Benutzer auf „Cancel“ in der GUI oder „Escape“ auf der Tastatur gedrückt hat, oder im Falle der Erstellung einer Animation das Ende des Animationsbereiches erreicht ist. Die Wartezeit zwischen zwei Ausführungen einer Methode kann dabei durch den Benutzer in „Run Delay“ eingestellt werden. Dies ist zum Beispiel bei einer zu schnellen Ausführung von Ableitungsschritten sehr von Vorteil, um das Geschehen besser nachvollziehen zu können.

4.5.2 xlEditor.cpp

Diese Datei enthält die Klasse für den XL-Editor. Es ist ein einfacher Editor, nach einem Schema aufgebaut, wie es von dieser Art Unmengen im Internet zu finden gibt.

Das Aussehen ist dabei an den Windows typischen Editor angelehnt. Er besitzt daher auch alle wichtigen Editier-Funktionen. Weggelassen wurde jedoch eine Funktion zum Drucken und zum Wechseln der Schriftart. Im Gegensatz zum Windows-Editor besitzt der XL-Editor zusätzlich eine Toolbar mit Icons, um einen schnellen Zugriff auf die wichtigsten Funktionen bereitzustellen. Alle zu sehenden Bitmaps in der Toolbar und im Menu mussten jedoch einzeln von Hand zugewiesen werden. Besonders die Toolbar mit der Definition jedes einzelnen Eintrags bläht an dieser Stelle den Quelltext auf.

Da der Editor ein Bestandteil des Plug-ins ist und somit von 3ds max, unterliegt er auch dessen Kontrolle. So wurden während der ersten Programmierversuche erst alle Tastatureingaben an 3ds max gesendet, statt in das Edit-Feld. Erst der Aufruf der Funktion `DisableAccelerators()` im Callback des Edit-Fensters ermöglichte Eingaben. Diese Methode deaktiviert sämtliche 3ds max-Tastatenkürzel (Hotkeys), welche sonst benutzt würden, um schnelleren Zugriff auf Funktionen von 3ds max zu ermöglichen. Diese Hotkeys werden mit der besagten Methode abgeschaltet, sobald der XL-Editor den Eingabefokus erhält, und wieder aktiviert, wenn das Feld den Fokus verliert. Dadurch werden alle Tastatureingaben korrekt weitergeleitet.

Ein Highlight des XL-Editors ist die Fähigkeit, Dateien per Drag&Drop anzunehmen. Dies bedeutet, dass der Nutzer nur seine XL-Datei im Windows Explorer anklicken muss, um diese dann auf das Edit-Feld zu ziehen. Dadurch erfolgt ein automatisches Laden der XL-Datei, ohne das entsprechende „Open File...“-Menu nutzen zu müssen. Die Umsetzung ist insofern interessant, als dass ein extra Callback angelegt werden musste. Dieses gilt speziell für das Edit-Feld und ist somit auch nur hier gültig. Dateien können deswegen auch nur auf das Edit-Feld gezogen werden, welches dann sein übergeordnetes Fenster, den XL-Editor, benachrichtigt, damit dieser die Datei lädt.

Beim Entwickeln des Editors wurde auch darauf geachtet, dass dieser nicht nur mit standardmäßigen Windows-Textdateien zurande kommt. Viele der schon entwickelten XL-Beispiele der Software GroIMP entstanden unter Linux, und eine Übernahme einer Textdatei von Linux nach Windows, um eventuell ein Beispiel von GroIMP nach 3ds max zu portieren, bringt das Problem des fehlenden „Cartridge Returns“ (CR) mit sich. In Unix-Systemen erfolgt ein Zeilenumbruch nur mittels „Line Feed“ mit dem ASCII-Code 10, während in Windows-typischen Textdateien erst ein „Cartridge Return“ mit Code 13 und dann ein „Line Feed“ steht. Deshalb wird durch XL4Max beim Auslesen von Dateien ein fehlendes CR mit eingefügt, um ein gleiches Aussehen beim Betrachten der Datei auf beiden Systemen zu ermöglichen. Wird diese Datei dann gespeichert, erfolgt dies im Windows-typischen Format.

Wünschenswert wäre es gewesen, dass der XL-Editor auch Dateien lesen kann, welche im Unicode-Zeichensatz erstellt worden sind. Leider hätte dies eine Umstellung des gesamten Projektes in Visual C++ auf Unicode bedeutet, was jedoch gerade mit den Methoden des SDK nicht verträglich ist. So bringt der Versuch, solche Dateien zu öffnen, ein nicht vorhersehbares Resultat mit sich.

Auf weitere Einzelheiten des Editors lohnt es sich hier nicht einzugehen, da dessen Funktionen sowohl in der Hilfe von XL4Max ausreichend beschrieben sind, als auch bei der Erklärung des Plug-ins im Kapitel 5.3.1 „Nutzung des Plug-ins (XL-Editor)“ erneut darauf eingegangen wird.

4.5.3 jniInterface.cpp

In dieser Klasse geschieht die Initialisierung der Java Virtual Machine. Sie besitzt nur die Methode `initJVM(...)`, welche beim Start von XL4Max aufgerufen wird. Diese Methode hat sich jedoch im Laufe der Entwicklung des Plug-ins immer weiter

verbessert. Einerseits bietet sie in ihrem jetzigen Zustand den größtmöglichen Komfort bei der Programmierung. Andererseits ist es möglich, eine beliebige Java-Version gemäß den Mindestvoraussetzungen einzusetzen, da XL4Max in die Lage versetzt wurde, selbständig den Pfad zur benötigten Datei „jvm.dll“ zu finden.

So wird durch zwei Registry-Aufrufe zuerst die auf dem System standardmäßig verwendete Java-Version und danach der Pfad zu deren *Runtime*-Bibliothek, der „jvm.dll“, ermittelt. Zum Lesen der Windows-Registrierung sind hier zwei Befehle nötig. Dies ist `RegOpenKeyEx(...)` zum Öffnen einer übergebenen Sektion, sowie `RegQueryValueEx(...)` zum Auslesen eines Schlüssels. Dabei wird letztere Methode zur Sicherheit zwei Mal aufgerufen. Beim ersten Mal wird gesondert getestet, ob besagter Schlüssel überhaupt existiert, und erst beim zweiten Aufruf wird der dann sicher vorhandene Wert in eine Variable gespeichert.

Zum Start der JVM ist des Weiteren die Angabe eines Class-Path nötig. Dieser Pfad zeigt auf die Java-Klassen, in denen die Methoden liegen und die dann von C++ aus aufgerufen werden können. Er verweist auf das Hauptverzeichnis des Java-Teils von XL4Max. Somit muss beim Finden der entsprechenden Klasse die Package-Struktur beachtet werden. Die Klasse „MaxControlFlow“ zum Beispiel befindet sich ausgehend vom Class-Path im Verzeichnis „de\grogra\ext\max\model\MaxControlFlow“. Dieser Pfad muss in der schon angesprochenen Methode

```
jclass cls = pJEnv->FindClass
    ("de/grogra/ext/max/model/MaxControlFlow");
```

beachtet werden.

Da der Nutzer nicht mit vielen Dateien in einer verworrenen Verzeichnisstruktur belastet werden soll, wurden alle Java-Klassen in eine einzelne Jar-Datei gepackt, der „XL4Max.jar“. Dieses Archiv-Format ist durch das JNI ohne Probleme lesbar und erfordert keinerlei Umstellung bei der Erzeugung der JVM. Es wird nun statt des Stammverzeichnisses zu den Java-Dateien diese Jar-Datei angegeben, die in sich dieselbe Verzeichnisstruktur gespeichert hat. Es entfällt dadurch jegliche Umstellung beim Finden und Aufrufen von Java-Methoden.

Der Start der JVM erfolgt mit verschiedenen anderen Parametern. Darunter auch die Angabe, wie viel Speicher die JVM belegen darf. Diese Optionen „-Xms“, „-Xmx“ und „-Xss“ werden standardmäßig durch die JVM gesetzt. Durch Angabe konkreter Werte in der XL4Max.ini lassen sich diese Parameter aber auch vom Nutzer festlegen. Somit ist es möglich, der JVM mehr Speicher zur Verfügung zu stellen. Es ist denkbar, durch zu viele Ableitungsschritte und daraus folgend zu viele Objekte im Szenengraphen den Speicher der JVM zu überfüllen, da während eines Ableitungsschrittes alle Objekte auch in Java verwaltet werden müssen.

Im Kapitel 4.3 „Eclipse“, wurde bereits die anfängliche Schwierigkeit angesprochen, auch den Java-Code zu debuggen. So war es zwingend erforderlich, bei Eintragung der Optionen diese nicht als statische Strings zu übergeben, sondern sie vorher mittels `strdup(...)` in einen extra Bereich im Speicher zu kopieren. Dies ist eine Eigenart von Suns JVM, die leider in keinerlei Dokumentation zu finden ist. Erst nach intensiven Recherchen war eine Lösung unter [20] zu finden.

Eine Erleichterung der Programmierarbeit war durch die Nutzung von Präprozessor-Anweisungen zu erreichen. Diese in Verbindung mit Projektmappenkonfigurationen ermöglichten die schnelle Erstellung verschiedener Konfigurationen. So existiert eine Konfiguration namens „Debug“, in der auch Optionen in die JVM geladen werden, um den Java-Code zu debuggen. Da dies aber eindeutige Geschwindigkeitseinbußen nach

sich zieht, existiert nebenher die Konfiguration „Release“. In dieser werden nicht nur die Möglichkeiten des Java-Debuggens entfernt, sondern zusätzlich noch alle im Quelltext befindlichen `assert (...)`-Anweisungen. Diese sind hauptsächlich in der „jniMethods.cpp“ vorhanden und dienen während der Entwicklungszeit zum Absichern verschiedenster Bedingungen.

4.5.4 jniMethods.cpp

In dieser Datei befindet sich die Sammlung von nativen Methoden, welche ein Auslesen und Bearbeiten des 3ds max-Szenengraphen ermöglichen. All diese Methoden werden während der Arbeit des XL-Compilers aus Java heraus aufgerufen. Damit dies möglich wird, ist es nötig, die Methoden vorher in der JNI zu registrieren. Der Befehl dafür lautet:

```
pJEnv->RegisterNatives (cls, nmAPI, nmcountAPI);
```

In der Variable `cls` steckt die Referenz zur Java-Klasse „de.grogra.ext.max.MaxAPI“, welche alle nativen Methoden aufgelistet hat. Zur Veranschaulichung dient hier das Beispiel der Methode „maxPrintln“, welche eine Textausgabe im MAXScript Listener tätigt. Ein Eintrag in der Java-Klasse „MaxAPI.java“ sieht wie folgt aus:

```
public static native void maxPrintln (String in_text, boolean
    debugPrint);
```

Diese Methode wird dann aus anderen Java-Klassen heraus aufgerufen. Das entsprechende Gegenstück auf der C++-Seite ist folgendes:

```
JNIEXPORT void JNICALL maxPrintln (JNIEnv *env, jclass in_cls,
    jstring in_string, jboolean debugPrint);
```

Wie zu erkennen ist, gleichen sich die beiden Code-Aufrufe im Rückgabeparameter, im Methodennamen und in den Übergabeparametern. Die zwei zusätzlichen Parameter `JNIEnv *env` und `jclass in_cls` müssen mit angegeben werden und können helfen, die entsprechende JNI und die Java-Klasse, die den Aufruf tätigt, zu ermitteln. Während der Programmierung von XL4Max war es jedoch an keiner Stelle nötig, auf besagte Variablen zuzugreifen.

Im Methodenaufruf `RegisterNatives (cls, nmAPI, nmcountAPI)` existieren zusätzlich noch die Parameter `nmAPI` und `nmcountAPI`. `nmAPI` ist ein Array des Typs `JNINativeMethod`, in dem alle nativen Methoden aufgelistet sind. Die Eintragung einer Methode erfolgt anhand von:

```
nmAPI[nmcounter].name = "maxPrintln";
nmAPI[nmcounter].signature = "(Ljava/lang/String;Z)V";
nmAPI[nmcounter++].fnPtr = maxPrintln;
```

`name` gibt den Namen an, unter dem die Methode in Java aufzurufen ist. Dieser muss nicht zwingend mit dem Methodennamen auf der C++-Seite übereinstimmen. Zur Vereinfachung aber wurde genau auf diese Vergabe gleicher Bezeichner während der Implementierungsphase von XL4Max geachtet. `signature` vermittelt der JNI die entsprechende Signatur, und `fnPtr` ist ein Zeiger auf die auszuführende C++-Funktion. Nach diesem Muster werden alle nativen Methoden in C++ sowie in Java angegeben und in der JNI registriert.

Die in der „jniMethods“-Klasse vorhandenen Methoden können nicht durch den Nutzer direkt aufgerufen werden. Dies war auch nie wünschenswert, da so ein größerer Teil der Implementierung auf die Java-Seite verlagert werden konnte. Somit beschränken sich die Funktionalitäten der nativen Methoden auf die reine Arbeit mit dem 3ds max-Szenengraphen, während nötige Berechnungen schon in Java ausgeführt werden. Dazu gehören auch Berechnungen, wie sie für die Arbeit in GroIMP genutzt werden, und deshalb mittels schon vorhandener Bibliotheken durchgeführt werden.

Die Nutzung von Referenzen in Java zur Identifizierung der Szenenobjekte ist ein weiterer Grund, warum die nativen Methoden nicht direkt aufgerufen werden sollten. Die Erzeugung eines Knotens im Szenengraphen erfolgt mittels der Befehle:

```
Object *obj = (Object*) pluginIP->CreateInstance (sclass_id,
        Class_ID (class_id_a, class_id_b));
INode *node = pluginIP->CreateObjectNode (obj);
```

sclass_id, class_id_a sowie class_id_b sind vom SDK vorgegebene Long-Werte. Jedes Objekt besitzt eine *Superclass*-ID, welche das Objekt einer bestimmten Kategorie zuordnet wie geometrisches Objekt, *Shape*-Objekt oder *Helper*-Objekt. Zusätzlich zur *Superclass*-ID besitzt jedes erzeugbare Objekt zwei Klassen-IDs. Die zweite dieser IDs ist bei vorgegebenen 3ds max-Objekten in den meisten Fällen eine Null und sollte bei der Erstellung eigener Objekttypen auf einen Wert ungleich Null gesetzt werden. Die drei IDs zusammen ermöglichen eine eindeutige Zuweisung zu einem Objekttyp, welcher mittels `CreateInstance(...)` erzeugt wird. Um das Objekt dann noch als neuen Knoten in den Szenengraphen einzufügen, ist der Befehl `CreateObjectNode(...)` nötig.

Der Zeiger `node` ist dann eine Referenz auf das entsprechende Objekt. Mittels

```
return (int) node;
```

wird diese Referenz an Java übergeben. Jedes Java-Objekt, welches einen Szenengraphenknoten repräsentiert, leitet von der Java-Klasse „Node“ ab. Diese besitzt das Attribut `ref`, in dem die übergebene Referenz gespeichert wird. Möglich wird das, da ein Zeiger in C++ 32 Bit groß ist, genauso lang wie der Datentyp `int` in Java ist.

In umgekehrter Richtung wird die native Methode aufgerufen, und dabei der Wert der Variable `ref` übergeben. So zum Beispiel in der Methode zum Löschen von Knoten:

```
JNIEXPORT void JNICALL maxDeleteObject (JNIEnv *env, jclass
        in_cls, jint source)
```

In `source` steckt die Referenz auf das zu löschende Objekt, und mittels

```
INode *sourceNode = (INode*) source;
```

wird aus dieser Referenz ein Zeiger erstellt, der auf das entsprechende Objekt im Speicher verweist. Damit wird ein voller Zugriff auf alle Eigenschaften des Knotens im Szenengraph möglich.

Erfolgt aus Java heraus die Übergabe ganzer Objekte, also nicht primitiver Datentypen, so besteht für diese noch eine Besonderheit. Als Beispiel sei hier die Methode `maxPrintln(...)` aus der Klasse „jniMethods.cpp“ angeführt. Diese native Methode erhält aus Java einen Java-String. Mit dem Befehl

```
const char *c_string = env->GetStringUTFChars (in_string,
        NULL);
```

wird dieser String `in_string` in ein für C++ verständliches *Character-Array* `c_string` geschrieben und danach durch `mprintf(...)` im MAXScript Listener - einer Art Konsole - ausgegeben. Bevor die Methode beendet werden darf, womit der Kontrollfluss wieder an Java zurückgeht, muss der übergebene String noch freigegeben werden. Dies hängt mit dem *Garbage-Collector* von Java zusammen.

Der *Garbage-Collector* hat die Aufgabe, den Hauptspeicher von nicht mehr benötigten Objekten frei zu räumen. Da in Java alle nicht-primitiven Objekte (das sind die Datentypen, die von *Object* erben) nur Zeiger auf einen Speicherbereich mit den eigentlichen Daten sind, muss der *Garbage-Collector* wissen, ob noch eine Referenz auf ein entsprechendes Objekt existiert. Erst durch den Aufruf von

```
env->ReleaseStringUTFChars (in_string, c_string);
```

wird dem *Garbage-Collector* mitgeteilt, dass die an C++ übergebene Referenz zu dem Java-String nicht mehr benötigt wird. So kann dieses Objekt nach Bedarf aus dem Speicher entfernt werden.

Anmerkung: In der Literatur gibt es widersprüchliche Aussagen darüber, ob es Zeiger in Java gibt oder nicht. So wird unter [21] behauptet „In Java gibt es keine Zeiger“ und „Eine Referenz unter Java ist nicht als Zeiger auf Speicherbereiche zu sehen“. Andere Quellen, wie unter [22], meinen dagegen „Der Objektzugriff in Java ist über Referenzen genannte Zeiger implementiert“. Einig sind sich alle Angaben dahingehend nur, dass in Java mit Referenzen gearbeitet wird. Es werden, abgesehen von den Primitivtypen, bei Methodenaufrufen also nicht die übergebenen Objekte im Speicher dupliziert, sondern nur Referenzen auf diese Objekte übergeben.

Wenn in diesem Text die Rede von Zeigern ist, so ist dies mit der Bezeichnung Referenz gleichzusetzen.

Der Nutzer hat die Möglichkeit, den Zugriff auf Objekte im Szenengraph auch mittels Namen zu steuern. So werden in der Ableitungsregel

```
b:Box, (b.getName().equals("Box01")) ==> ...
```

nur Boxen beachtet, die den Namen „Box01“ besitzen. Solche Objekte müssen, wie schon erläutert wurde, ein Kindsknoten vom Knoten „XLRoot“ sein. Dies hängt mit der Arbeitsweise der Methode `supplyNodes(...)` der Java-Klasse „Graph.java“ zusammen. Eine Ausnahme bildet aber die dem Nutzer verfügbare Funktion `CloneNodes(...)`, welche mit Angabe eines Knotennamens aufgerufen werden kann. In dem Fall erfolgt die Suche nach einem entsprechenden Knoten schon an der Wurzel des Szenengraphen. Damit hat der Nutzer die Möglichkeit, Objekte in 3ds max zu erstellen, welche er später durch XL duplizieren will, ohne dass die Originalobjekte beim Drücken von „Reset“ gelöscht werden.

XL4Max bietet die schon angeführte Möglichkeit, Ableitungsschritte als Animation in 3ds max abzulegen. Dazu werden sogenannte *Keys* abgelegt, die zu den verschiedenen Zeiten die unterschiedlichen Zustände der Objekte in sich speichern. Diese *Keys* von Hand in jeder Methode zu erzeugen, welche eine relevante Veränderung des Szenengraphen durchführt, überstieg leider das in der vorgegebenen Zeit Machbare. So ist in all diesen Methoden ein einfacher Mechanismus eingebaut, welcher die „Auto Key“-Funktion von 3ds max anspricht. Mittels

```
SuspendAnimate ();  
AnimateOn ();
```

wird die entsprechende Funktion aktiviert. `SuspendAnimate()` dient dabei nur zur Sicherheit, dass eine eventuell laufende Animation in 3ds max gestoppt wird. `AnimateOn()` schaltet daraufhin 3ds max in einen Zustand, in dem es jegliche Veränderung von Objektattributen überwacht und entsprechend *Keys* erzeugt. Wurden in einer C++-Methode alle nötigen Änderungen am Szenengraphen durchgeführt, wird noch mittels `ResumeAnimate()` die „Auto Key“-Funktion von 3ds max deaktiviert.

Dieses Prinzip funktioniert beim Setzen der Attribute von Objekten und von Materialien zuverlässig. Da aber die meisten XL-Programme darauf beruhen, dass Ableitungsschritte dazu führen, dass Objekte ersetzt werden, tat sich hier während der Programmierung von XL4Max ein großes Problem auf. Denn das Erzeugen und das Löschen von Objekten zu bestimmten Zeiten sind nicht möglich. Entweder, Objekte existieren in einer Szene während der ganzen Animationszeit, oder sie existieren nicht. Somit musste ein Umweg gefunden werden, Objekte erst zu einer bestimmten Zeit sichtbar zu machen beziehungsweise verschwinden zu lassen. Es blieb nur die Wahl, in dem sogenannten *Controller*, welcher für die Sichtbarkeit eines Objekts zuständig ist, verschiedene Werte für verschiedene Zeiten einzutragen. Wird also ein Objekt erst zu einer Zeit `actTime` erstellt, so wird mittels der Befehle

```
node->SetVisibility (TIME_NegInfinity, 0.0f);
node->SetVisibility (actTime, 1.0f);
```

die Sichtbarkeit in der Vergangenheit (`TIME_NegInfinity`) auf null gesetzt, und zur `actTime` auf eins. Ist die Checkbox „Smooth Creation und Deletion“ deaktiviert, so wird zusätzlich noch mit „`node->SetVisibility (actTime - 1, 0.0f);`“ im Zeitpunkt unmittelbar vor dem aktuellen die Sichtbarkeit auch noch auf null gesetzt. 3ds max ist darauf programmiert, standardmäßig eine *Smooth*-Interpolation der Werte an den *Keys* vorzunehmen, was dazu führen würde, dass Objekte nicht abrupt, sondern schon vorher langsam sichtbar werden.

Eine genauere Betrachtung der Methoden dieser Klasse würde den Umfang dieses Textes sprengen. Es sei nur soviel gesagt, dass die Umsetzung der Funktionalität einzelner Methoden an sich nicht sehr schwer war. Vieles wurde durch das SDK vermittelt, leider aber nicht jedes Detail. So waren an vielen Stellen kleine Probleme vorhanden, deren Lösung nirgends erklärt und nicht immer ganz schlüssig war.

So muss zum Beispiel vor dem Setzen des Textes eines Text-Objektes unbedingt auch die Schriftart des Textes neu gesetzt werden. In diesem Fall wird einfach die Schriftart des Objektes ausgelesen und verwendet. Ohne dieses Vorgehen war 3ds max einfach nicht dazu zu bewegen, einen neuen Text anzunehmen.

Ein anderes Manko ist, dass keine *Plane* aus der Gruppe der *Standard Primitives*-Objekte erzeugt werden kann. Aus unerfindlichen Gründen wird trotz gleicher Vorgehensweise wie bei anderen Objekten dieses nicht erzeugt. Auch intensive Suchen im Internet brachten keine Lösung, so dass der Anschein aufkam, dieses Problem trete nicht auf jedem System auf. Leider gab es keine Möglichkeiten, dies weiter auszutesten. So bleibt dem Nutzer nur die Wahl, statt einer *Plane* eine *Box* mit der Höhe null zu erstellen. So geschehen im Beispiel „Bush.xl“, auch wenn dies leider Geschwindigkeitseinbußen nach sich zieht. Denkbar wäre zwar auch die Umsetzung eines *Rectangle*-Objektes aus der Klasse der *Shape*-Objekte, welches dann mit dem „Extrude“-Modifikator belegt werden müsste. Es ist aber fraglich, ob diese Variante schneller ist als die im Beispiel verwendete.

4.6 Java

Der Java-Teil der Implementierung umfasst im Gegensatz zum C++-Teil sehr viel mehr Klassen. Diese fast 70 Klassen hier einzeln aufzuführen, würde keinen Vorteil bringen. Es erfolgt deshalb eine Einschränkung auf die Erklärung der *Packages* und einzelner ausgewählter Klassen, denen eine besondere Bedeutung zukommt.

4.6.1 de.grogra.ext.max

In diesem *Package* befindet sich eine Ansammlung diverser Klassen, darunter die Klasse „Node“, eine der wichtigsten Klassen überhaupt. Sie ist die Grundklasse für alle im Szenengraphen enthaltenen Knoten, die diese Klasse deswegen erweitern.

„Node“ enthält eine lange Reihe von Methoden, die jeder Knoten auf sich anwenden kann. Dazu gehören Transformationsmethoden wie `move`, `rotate`, `scale`; Methoden zum Berechnen der Entfernung zu anderen Knoten und zum Setzen der Transformationsmatrix oder des Materials bzw. der Farbe. Ferner aber auch Methoden, welche weniger vom Benutzer als mehr vom XL-Compiler benötigt werden, wie beispielsweise Methoden zum Auslesen der Kinder und der Eltern im Szenengraph, zum Bestimmen des Knotentyps und besonders wichtig: zum Setzen und Entfernen von Kanten zu anderen Knoten. Da im Szenengraph von 3ds max nur Hierarchie-Kanten existieren können, musste die Erweiterung um zusätzliche Kanten in der Klasse „Node“ erfolgen. Dies bedeutet ein explizites Speichern, was für eine Kante (bestimmt durch die Kantenbits) zu anderen Knoten führt, und welche Kanten an diesem Knoten ankommen. Die Zusammenführung der Hierarchiekanten aus 3ds max mit den zusätzlichen Kanten in Java brachte gerade in den Methoden `addEdgeBits(...)` und `removeEdgeBits(...)` große Probleme mit sich. So muss nicht nur gefiltert werden, welche Kante bearbeitet werden soll und an welcher Stelle dies geschieht. Es ist auch nötig, ständig in Java mitzuführen, wie viele zusätzliche Kinder, also Kanten zu anderen Knoten, ein Knoten besitzt. Dessen korrekte Angabe ist für den *EdgeIterator* nötig, der vom XL-Compiler benötigt wird und über alle ein- und auslaufenden Kanten eines Knotens läuft. Mehr dazu in diesem Kapitel 4.6.5 „de.grogra.ext.max.model“. Um also die Anzahl der zusätzlichen Kinder immer aktuell zu halten, muss bei jedem `addEdgeBits` und `removeEdgeBits` abgeglichen werden, ob eine Kante im Hierarchiegraph von 3ds max existiert oder nicht - und dementsprechend müssen Veränderungen an den Variablen vorgenommen werden.

Bei `removeEdgeBits` gilt noch zu beachten, dass diese Methode auch dafür sorgt, dass ein Knoten gelöscht werden muss. Existiert zu einem Knoten ab einem gewissen Zeitpunkt keine Hierarchiekante mehr, so ist es die Absicht von XL, dass dieser Knoten dann nicht mehr sichtbar sein soll. 3ds max verhält sich aber in dieser Situation anders als GroIMP und hängt den Knoten an die Wurzel des Szenengraphen. Deshalb wird von Hand mitgeführt, welche Knoten ihre Kante zum Elternknoten verlieren, und diese werden in dem Fall auf die Liste `Graph.nodesToDelete` gesetzt. Die Knoten, die dann doch wieder einem anderen Knoten als Kind zugewiesen werden, werden in der Methode `addEdgeBits` wieder von dieser Liste entfernt.

Wichtig ist, dass es nicht gestattet ist, eine Instanz von „Node“ zu erzeugen, zum Beispiel mittels der Regel:

```
Axiom ==> Node;
```

Dies würde nämlich auf Seiten von 3ds max kein Objekt im Szenengraphen erzeugen. Da dieses aber der XL-Compiler in so einem Fall erwarten würde, käme es zu Konflikten bei der Anwendung aller nötigen Methoden wie `addEdgeBits` auf dieses neue Objekt.

Eine Auflistung aller restlichen, für den Nutzer wichtigen Methoden findet sich in der Hilfe von XL4Max.

Dasselbe gilt für die Methoden der Klasse „MaxLibrary“. Das ist eine standardmäßig in jedes XL-Projekt importierte Klasse mit verschiedenen, nützlichen Funktionen. Diese Bibliotheksmethoden können vom Nutzer direkt aus XL heraus aufgerufen werden, und sollen so das Einbinden zusätzlicher Methoden und Java-Klassen ersparen. Zum Beispiel sind hier zwei einfach zu benutzende Funktionen, um dem Nutzer eine Zufallszahl zu liefern; eine für Int-Werte und eine für Float-Werte. Eine Erklärung all dieser Methoden ist auch in der Klasse „MaxLibrary.java“ zu finden.

Die „Dummy“-Klasse ist eine Hilfsklasse für alle Objekte, die nicht in der Szene sichtbar sein sollen, aber im Szenengraphen vorhanden sein müssen. Dazu zählen neben den in diesem Package vorhandenen Klassen „Axiom“, „Root“ und „WrapperNode“ auch die Rotationsknoten der Turtle-Befehle. „Axiom“ und „Root“ spiegeln die entsprechenden Knoten „Axiom“ und „XLRoot“ im 3ds max-Szenengraphen wieder. „WrapperNode“ spielt als abstrakte Klasse eine besondere Bedeutung. Die Klassen „BooleanNode“, „ByteNode“ usw. erweitern diese Klasse. Es ist mit Hilfe dieser Klassen dann möglich, bestimmte Werte, entsprechend des Typs im Namen der Klasse, in einem Knoten des Szenengraphen zu speichern und diesen wieder auszulesen. Ein Beispiel für deren Anwendung findet sich in „Ants.xl“, wo ein solcher Knoten genutzt wird, um einer Ameise ein Gedächtnis zu bestimmten Zellen zu geben.

Die Zellen-Klasse „Cell“ wurde explizit implementiert, da diese auch in vielen XL-Beispielen wie „Ants.xl“ oder „GameOfLife.xl“ Anwendung findet. Im Fall von „Game of Life“ wurde während der Programmierung auch das erste Mal deutlich, wie wichtig die Anwendung der „GraphPropertyQueue“ ist. Erst mit Hilfe dieser Warteschlange wurde es möglich, dass Properties auch verspätet gesetzt werden können. So wird die Methode zum Ändern des Zustands auf die besagte Warteschlange gesetzt, und erst ausgeführt, nachdem für jede einzelne Zelle die Ableitungsregel angewendet wurde, also entschieden ist, welcher Zustand der nächste sein wird.

Die restlichen Klassen sind „MaxAPI“ und „MaxHelper“. „MaxAPI“ wurde schon im C++-Teil angesprochen, und dient nur zur Auflistung der nativen Methoden, welche dann aus anderen Java-Klassen heraus aufgerufen werden. „MaxHelper“ ist eine Hilfsklasse, ursprünglich gedacht für alle Helper-Objekte aus 3ds max, von denen jedoch nur „Dummy“ umgesetzt wurde.

4.6.2 de.grogra.ext.max.geomobj

Dieses Package enthält Klassen für die umgesetzten geometrischen Objekte aus 3ds max, dort als *Standard Primitives* aufgeführt. Nur „Hedra“ stammt aus dem Bereich der *Extended Primitives*, welches als einziges aus dem Bereich genauso einfach umzusetzen war.

„MaxGeomobject.java“ ist eine Hilfsklasse für diese Objekte. Von ihr leiten die anderen Klassen ab. Sie sorgt somit für eine korrekte Zuordnung der schon erwähnten *Superclass-ID*, welche 3ds max für die Objekterstellung benötigt. Außerdem ist die in ihr enthaltene Methode `getNodeForGeomobj(...)` dafür zuständig, schon im 3ds max-Szenengraphen vorhandene Objekte ihrem Typ zuzuordnen. Damit erfolgt dann eine Erstellung einer Instanz der richtigen Klasse.

Die spezifischen Methoden und Konstruktoren, die jede Klasse enthält, sind wie von jeder anderen Klasse, auf die der Nutzer Zugriff hat, in der Hilfe von XL4Max aufgeführt und beschrieben. Es sind Methoden, mit denen der Nutzer jedes Attribut, wie

es auch in 3ds max zu den entsprechenden Objekten vorhanden ist, verändern kann. Damit ist sichergestellt, dass es möglichst wenige Einschränkungen durch die Nutzung von XL gibt.

In jeder Klasse existiert auch eine Reihe von Konstanten. Jede steht, bis auf gewisse Ausnahmen, für ein Attribut eines Objektes. Die dahinter stehende Zahl wird beim Setzen einer Eigenschaft mittels `maxSetFloatProperty(...)` bzw. `maxSetIntProperty(...)` übergeben und entspricht dem Wert, die die eigentliche 3ds max-Methode zum Ändern des Attributes benötigt, um überhaupt das zu ändernde Attribut zu bestimmen. Auch werden diese Konstanten Java-intern benötigt, um die Properties, wie zum Beispiel `Sphere[radius]`, zu handhaben.

4.6.3 de.grogra.ext.max.shape

Dieses Package entspricht vom Aufbau her dem im Kapitel 4.6.2 „de.grogra.ext.max.geomobj“ angesprochenen Package. Es enthält die Hilfsklasse „MaxShape.java“, und von ihr leiten die restlichen Objekte des *Package* ab. Diese entsprechen den *Shape*-Objekten in 3ds max.

Eine Ausnahme bilden die Klassen „Spline.java“ und „Vertex.java“. „Spline“ entspricht dem *Shape*-Objekt „Line“ aus 3ds max. Verzichtet wurde jedoch auf dessen Eigenschaften. „Vertex“ ist sogar ein anderer Objekttyp und leitet sich von „Dummy“ ab. Dieser Umweg ist auf die Besonderheiten bei der Erstellung einer „Line“ zurückzuführen. Deren Umsetzung ist sehr kompliziert, und die direkte Angabe von Punkten im Raum, durch die der *Spline* gehen soll, entspricht nicht dem Prinzip der Ableitungsregeln. So erfolgte eine Umsetzung in der Art, dass mittels „Spline“ ein Startpunkt für eine *Spline*-Kurve geschaffen wird, und „Vertex“ einen Knotenpunkt zu dieser *Spline* erzeugt. Einfache Beispiele für deren Anwendung finden sich in „Splines.xl“.

Da ein neuer *Vertex* nur zu einem *Spline* gehören kann, ist dies das *Spline*-Objekt, welches in der Hierarchie als nächstes vor dem *Vertex* steht. Es erfolgt eine Suche Richtung XL-Root, welches die Methode `searchVertex(...)` realisiert. Um diese Suche zu beschleunigen, speichert jedes *Vertex*-Objekt ab, zu welchem *Spline*-Objekt es gehört. Trifft somit ein *Vertex* bei der Suche nach einem *Spline* auf einen anderen *Vertex*, so steht fest, dass er zum selben *Spline*-Objekt gehören muss. Zu beachten ist, dass beim Hinzufügen eines neuen *Vertex* zu einem *Spline* die absolute Angabe durch XL in eine relative Angabe zum *Spline*-Ursprung umgerechnet werden muss. Dies ist eine Eigenart der von 3ds max vorgegebenen Methode zum Erweitern von *Splines*.

Ein Missstand dieser Art der Erzeugung von *Splines* wird deutlich, wenn man sich vor Augen führt, in welcher Weise dies geschieht. Der *Vertex* ist nämlich nur ein Hilfsobjekt, anhand dessen Position im Raum ein neuer *Vertex* im eigentlichen *Spline*-Objekt in 3ds max erstellt wird. Ist dies einmal geschehen, hat der *Vertex* des Szenengraphen keinerlei Einfluss mehr auf die Position des tatsächlichen *Vertex* im *Spline*-Objekt. So ist es nachträglich nicht mehr möglich, das Aussehen der *Spline*-Kurve zu ändern.

4.6.4 de.grogra.ext.max.lsystem

In diesem Package befinden sich alle für XL4Max umgesetzten Turtle-Befehle. Diese lehnen sich den Turtle-Befehlen von GroIMP an und lassen sich in bestimmte Kategorien einteilen. Nur „F“ und „F0“ erzeugen sichtbare Objekte in der Szene. „M“ und „M0“ verhalten sich wie die „F“s, erzeugen dabei aber keine sichtbaren Objekte. Die „0“ hinter den Befehlen bedeutet, dass eine bestimmte vorgegebene Länge genutzt wird, die mittels „L“ festgelegt wird. Ein „D“ dagegen bestimmt die Dicke der nachfolgenden „F“s. Ein „Add“ und ein „Mul“ hinter einem Befehl stehen dafür, dass entsprechend der

angegebene Wert auf den schon gesetzten Wert aufaddiert bzw. hinzu multipliziert wird. Mittels „P“ wird die Farbe der folgenden Objekte festgelegt. Ein „l“ (der Kleinbuchstabe L) hinter einem Befehl bedeutet, dass dieser nur für das nächstfolgende „F“ in der Hierarchie gilt. „Plus“, „Minus“ und „AdjustLU“ sowie alle Befehle, die mit „R“ beginnen, bedeuten eine Rotation.

„F“ erzeugt in der Szene einen Zylinder. Dies entspricht am ehesten der Vorstellung eines Striches mit Dicke. Die Segmentzahl ist dabei auf sechs heruntersgesetzt, um zwar noch ein rundes Aussehen im Sichtfenster von 3ds max zu ermöglichen, andererseits aber die Polygonzahl der Szene nicht unnötig in die Höhe zu treiben.

Alle anderen Turtle-Befehle leiten sich vom *Dummy*-Objekt ab. Damit passiert es zwar, dass der Szenengraph mit vielen nicht sichtbaren Objekten gefüllt ist. Aber dies entspricht auch der Handhabung in GroIMP, denn nur so ist gesichert, dass alle Ableitungsregeln mit allen Ersetzungen durchgeführt werden können. Es werden Ersetzungen, nicht nur von „F“s, sondern auch von Rotationsknoten und allen anderen Turtle-Befehlen möglich.

Das Prinzip der Umsetzung der Turtle-Befehle beruht auf dem Speichern von Zuständen, sogenannten Turtle-States. XL4Max besitzt dafür die eigene Klasse „MaxTurtleState.java“, welche die Werte für Länge, Durchmesser und Farbe speichert. Zusätzlich sind noch Variablen für die Umsetzung von Turtle-Befehlen nötig, die nur das nächstfolgende „F“ beeinflussen. Die restlichen Rotationsbefehle werden nicht als Zustand gespeichert, sondern direkt umgesetzt, indem das *Dummy*-Objekt entsprechend rotiert wird.

Jeder in XL erzeugte Knoten besitzt eine Instanz der Klasse „MaxTurtleState“, auch wenn ein Knoten kein Turtle-Befehl ist. Die Abarbeitung der Befehle erfolgt dann in der Methode `commitTurtleStates()` in der Klasse „Graph.java“. Diese Funktion kann erst aufgerufen werden, wenn der XL-Compiler seine Arbeit beendet hat und alle Hierarchiekanten gesetzt sind. Es erfolgt eine Traversierung des Szenengraphen, beginnend beim XL-Root. Dieser besitzt einen Standard-MaxTurtleState, den er auf all seine Kinder vererbt. Diese Kinder werden auf einen Stack gepackt, um später wieder heruntergenommen werden zu können und denselben Ablauf durchzuführen. Die Arbeit mittels Stapelspeicher und dem *LIFO*-Prinzip (Last In - First Out) ermöglicht eine korrekte Abarbeitung des Szenengraphen. In einer Hierarchie werden so Turtle-Befehle wirklich nur auf direkte Nachfahren angewendet und nicht auf Objekte, die durch eine andere Kante in Beziehung mit dem Vorfahren stehen.

Bevor ein Knoten seine Kinder auf den Stack packt und ihm seinen Turtle-State mit übergibt, muss natürlich der Knoten selbst untersucht werden. In einer *case*-Anweisung wird herausgefunden, ob der Knoten einen Turtle-Befehl repräsentiert. In diesem Fall verändert der Knoten seinen eigenen Turtle-State entsprechend dem Turtle-Befehl. Zum Beispiel würde ein „P“ beim Erzeugen mittels Ableitungsregel

```
Axiom ==> P(14) F(1);
```

in einer internen, privaten Variable den Farbwert „14“ abspeichern. Erst in der Methode `commitTurtleStates(...)` wird dieser Wert ausgelesen und in den Turtle-State des „P“s gespeichert. Dieses übergibt danach den Turtle-State an sein Kind „F“, welches die Methode `setTurtleState(...)` besitzt. In dieser Funktion wird dann der 3ds max-Befehl zum Ändern der Farbe des eigentlichen Zylinders aufgerufen.

Da „F“ mit verschiedenen Variablen erzeugt werden kann, sollen diese nicht durch andere Turtle-Befehle beeinflusst werden. Wird die obige Regel zum Beispiel wie folgt umgeschrieben

```
Axiom ==> P(14) F(10, 1, 15);
```

so hat das „P“ keinen Einfluss mehr auf die Farbe des „F“'s, welche durch einen solchen Aufruf auf 15 gesetzt wird. Ähnlich ist das Prinzip bei der Dicke und bei „F0“ auch bei der Länge. Dazu werden weitere Variablen in den Knoten nötig, wie `hasDiameter` und `hasPaint`, die sich merken, wie der Konstruktor von „F“ bzw. „F0“ aufgerufen wurde. Entsprechendes ist in den Klassen „M“ und „M0“ zu finden.

Eine Besonderheit der Turtle-Befehle von XL4Max ist das Kommando „RT“, da es nicht an einen existierenden Turtle-Befehl von GroIMP angelehnt ist. „RT“ wird mit der Übergabe eines anderen Knotens, des Zielknotens, erzeugt, und optional kann noch eine Stärke angegeben werden. Daraufhin wird eine entsprechende C++-Methode aufgerufen, die die Transformationsmatrix vom Knoten „RT“ so verändert, dass dessen *Up*-Vektor in die Richtung des Zielknotens zeigt. Die dafür zuständige Methode `maxSetDirection(...)` führt dazu einige mathematische Berechnungen durch. Zuerst wird ein neuer *Up*-Vektor bestimmt, der in die Richtung des Zielvektors zeigt. Dann wird mit Hilfe des aktuellen und des gewünschten Vektors ein neuer Vektor bestimmt, der senkrecht zu beiden steht. Über diesen erfolgt mittels einer 3ds *max*-Methode eine Rotation, in Abhängigkeit von der angegebenen Stärke. Die genaue Vorgehensweise des Algorithmus ist in der Klasse „jniMethods.cpp“ kommentiert, in der die Methode `maxSetDirection(...)` enthalten ist.

4.6.5 de.grogra.ext.max.model

Dieses Package enthält die eigentliche Schnittstelle zum XL-Compiler. Dazu gehören die Klassen „Compiletime“, „Runtime“, „Graph“ und „MaxXLFilter“. Letztere leitet von der Klasse „XLFilter“ aus dem Package „de.grogra.ext.maxmaya“ ab. Beide zusammen sind dafür zuständig, den XL-Compiler zu initialisieren und ihm dabei die benötigten Informationen (welches Compiletime-Model und welches Runtime-Model genutzt werden soll) mitzuteilen. In der Methode `getStaticTypeImports()` wird die Klasse „MaxLibrary“ zurückgegeben, was dem Benutzer ermöglicht, direkt alle darin enthaltenen Methoden aufzurufen. In `getPackageImports()` dagegen werden ganze *Packages* angegeben, aus denen der Benutzer dann Java-Klassen verwenden kann. Durch beide Methoden entfallen lästige und unnötige Imports für den Nutzer, da diese Angaben so gut wie immer benötigt werden.

Das schon angesprochene Compiletime-Model wird durch die Klasse „Compiletime.java“ festgelegt. Es beschreibt die Struktur des Szenengraphen zur Kompilierzeit. Dies bedeutet, dass bestimmte Konstrukte, wie sie in der XL-Datei vorliegen, hier beschrieben sein müssen. `getNodeClass()` liefert die Klasse „Node.java“, welche den Standard-Knotentyp von XL4Max beschreibt. `getStandardEdgeFor(...)` vermittelt dem Compiler dagegen den standardmäßig verwendeten Kantentyp, in dem Fall „SUCCESSOR“, also die Hierarchiekante. `getDefaultModuleSuperclass()` liefert den Knotentyp *Dummy*, von dem alle nicht spezifizierten Module ableiten. Die komplizierteste Methode ist aber wohl die `getDirectProperty(...)`-Methode. Immer, wenn im Quelltext auf eine Property zurückgegriffen wird, gibt diese Methode eine spezielle Instanz der Klasse „GraphProperty“ zurück. Dazu wird die Klasse bestimmt, zu der die Property gehört, und nachgeschaut, ob ein entsprechendes Attribut existiert. Im Beispiel `sphere[radius] = 5;` wird ein entsprechender Eintrag gefunden und der dazu nötige Typ, nämlich Float, bestimmt. Entsprechend dieses Typs wird dann eine „GraphProperty“-Instanz erzeugt. Zusätzlich werden noch der Name der Property und eine eindeutige ID eingetragen. Dieses Objekt wird dann in der Klasse „Runtime“ weiter verwendet.

Die „Runtime.java“ entnimmt dann in der Methode `propertyForName(...)` aus der übergebenen ID wieder die nötigen Informationen, um die Property einer entsprechenden Klasse zuzuordnen. Dazu wird ein neues „GraphProperty“-Objekt erzeugt, welches die entscheidenden Methoden zum Setzen der Werte beinhaltet. Diese Methoden müssen bei verschiedenen Properties unterschiedlich vorgehen. Während die Attribute in den Klassen „Sphere“, „Box“ und so weiter auf gleiche Weise gesetzt werden, sind für Attribute wie für Verschiebung und Rotation andere C++-Methoden zuständig. Deshalb erfolgt eine Unterteilung in den Set-Methoden nach den unterschiedlichen Properties.

Des Weiteren sind in „Runtime.java“ noch Methoden, die Werte und Objekte in eigene Klassen verpacken können. So erzeugt die Methode `wrapInt(...)` eine neue Instanz der schon angesprochenen „IntNode“-Klasse und übergibt den Wert, damit dieser gespeichert wird.

Die genauen Funktionsweisen des Runtime- und Compiletime-Models finden sich in der XL-Spezifikation unter [23] im Kapitel 3 „Data Model Interface“.

Die Klasse „Graph.java“ übernimmt neben der schon angesprochenen Aufgabe der Abarbeitung der Turtle-Befehle noch weitere wichtige Arbeit. Sie stellt zum Beispiel zwei Listen zur Verfügung: `nodesToDelete`, in der die zu löschenden Knoten gesammelt werden, und `nodesToCommit`. Letztere steht in direktem Zusammenhang mit der Methode `supplyNodes(...)`. In `nodesToCommit` werden alle Knoten geführt, die in einem Lauf des XL-Compilers erstellt werden. Diese Knoten stehen noch nicht während der Abarbeitung von Ableitungsschritten innerhalb eines Compiler-Laufs zur Verfügung. Die Abarbeitung des Compilers erfolgt durch die Methode `supplyNodes(...)`, welche alle Knoten des Szenengraphen durchgeht; also mittels Tiefensuche traversiert, und bei Übereinstimmung des Knotens mit einem Typ diesen an den XL-Compiler liefert. Dabei werden nur die Knoten durchgegangen, die nicht erst in diesem Lauf erstellt wurden, und deren Zustand in der Variable `committed` dementsprechend nicht mehr den Wert `false` besitzt. Erst am Ende eines Laufes wird in `commitModifications()` die Variable dieser neu erstellten Knoten auf `true` gesetzt.

Eine weitere wichtige Aufgabe übernimmt der *EdgeIterator*, welcher mittels `createEdgeIterator(...)` an den XL-Compiler geliefert wird. Er läuft während seiner Arbeit über alle ein- und auslaufenden Kanten eines Knotens und setzt dabei die für den Compiler wichtigen Variablen `source`, `target` und `bits`. Dieser hat damit überhaupt erst die Fähigkeit, Kantenoperationen zwischen zwei Knoten durchzuführen. Die Aufgabe der einzelnen Methoden des *EdgeIterators* ist schon geklärt durch die Klasse, von der er ableitet. Eine Besonderheit, die erst recht spät im Entwicklungsstadium von XL4Max festgelegt wurde, ist die Tatsache, dass dieser rückwärts über alle Kanten läuft. Die Kanten zu anderen Objekten sind durchnummeriert, und die Arbeit des *EdgeIterators* beginnt bei der Kante mit der höchsten Nummer. Dies ist eine Eigenart der Implementierung, und wirkt der Tatsache entgegen, dass Kanten während der Arbeit des *Iterators* gelöscht werden könnten. Dies würde aber die Nummerierung durcheinander bringen, so dass der *EdgeIterator* ein unvorhersehbares Verhalten entwickelt. Durch die umgekehrte Abarbeitung wird dieses Problem umgangen, da gelöschte Kanten zwar immer noch die Nummerierung durcheinander bringen - aber dies nur an den Nummern und Kanten, die schon abgearbeitet wurden.

Die Methode `createPropertyModificationQueue(...)` erzeugt eine Instanz der Klasse „GraphPropertyQueue.java“.

Diese „GraphPropertyQueue“ spielt eine entscheidende Rolle bei der Benutzung von sogenannten quasi-parallelen Zuweisungen. Ein Beispiel dafür ist `„sphere[radius] := 5“`, was im Gegensatz zur normalen Anweisung nur den Doppelpunkt mehr enthält. Dieser sagt aus, dass die Zuweisung parallel zu allen anderen Zuweisungen geschieht.

Da parallele Verarbeitung aber so im Computer nicht möglich ist, sondern durch sequentielle simuliert werden muss, wird hier auf die Hilfe einer Warteschlange zurückgegriffen. Diese sammelt alle Zuweisungen und führt sie am Ende der Arbeit des XL-Compilers aus. Dadurch werden alle Abhängigkeiten zwischen mehreren Zuweisungen gelöst, da noch mit dem alten Wert der Variable gearbeitet wird.

Genau genommen enthält die „GraphPropertyQueue“ sogar zwei solcher Warteschlangen. Dies ergab sich im Laufe der Programmierung. Denn es war für das Setzen des Zustands einer „Cell“ die erste Queue `entrys` ausreichend. Später wurde jedoch festgestellt, dass es nötig ist, auch noch die Queue `lateEntryS` zu implementieren. Diese wird erst abgearbeitet, nachdem mittels `addEdgeBits(...)` und `removeEdgeBits(...)` alle Kantenoperationen erledigt worden sind. So im Beispiel des Turtle-Kommandos „RV“, welches bei der Berechnung der Rotation auf die Transformationsmatrizen von sich und von seinem Vorgänger zurückgreift. Diese müssen dafür schon die Werte besitzen, die sich durch ihre Position im Raum nach dem Lauf des Compilers ergeben. Während also die Liste `entrys` in der dafür vorgesehenen Methode `apply()` mittels Schleife alle ausstehenden Methodenaufrufe durchführt, wird dieses für die Liste `lateEntryS` erst in der dafür nicht vorgesehenen Methode `clear()` getätigt. Diese ist eigentlich dafür vorgesehen, die Warteschlange für den nächsten Transformationsschritt zu leeren.

Die „GraphPropertyQueue“ stellt mehrere Funktionen bereit, um Methoden, welche später ausgeführt werden sollen, in die beiden Listen einzutragen. Die restlichen in der Klasse vorhandenen Funktionen dienen dem Setzen von Attributen und entsprechen verschiedenen Zuweisungsoperatoren aus der Syntax von XL. So entspricht die Zuweisung mittels „:=“ den Methoden, die mit „set“ beginnen. Angeführt wird an allen Methoden noch der jeweilige Typ wie *Boolean*, *Byte* und so weiter. Die Operatoren „:&=“, „:|=“ und „:^=“ führen die booleschen Operationen „AND“, „OR“ beziehungsweise „XOR“ auf die Attribute aus. Und „:+=“, „:-=“, „:*=“ und „:/=“ rufen dementsprechend die Methoden beginnend mit „add“, „mul“ und „div“ aus, wobei die Subtraktion mittels „add“ und negativem Wert funktioniert. All diese Methoden führen erst eine entsprechende Berechnung des neuen Wertes durch und setzen dann die zugehörige Methode der „GraphProperty“ aus der „Runtime“-Klasse auf die Queue `entrys`.

Die letzte Klasse in diesem *Package* ist „MaxControlFlow.java“. Diese wurde schon im Kapitel über den Implementierungsteil von C++ aufgeführt. Sie enthält die Methode `main(...)`, welche aufgerufen wird, um die angegebene XL-Datei zu kompilieren. Dazu sind allerdings mehrere Schritte notwendig.

Da in einer XL-Datei mehr als eine Klassendefinition stehen kann, es aber nur eine Hauptklasse geben kann (deren Methoden als Buttons erzeugt werden), wird diese Klasse anhand des Dateinamens ermittelt. Des Weiteren werden nur Methoden als ausführbar übernommen, die als *Public* und *Static* deklariert sind und nicht der Konstruktor der Klasse sind. Der Konstruktor einer Klasse wird Java-intern mit dem Methodennamen „<init>“ geführt. Eine spezielle Methode namens „init“ wird auch extra geführt, da diese direkt nach dem Start des Compilers ohne Zutun des Nutzers ausgeführt wird.

Der Compiler-Aufruf ist in einen *Try-Catch*-Block eingeschlossen. Dadurch ist es möglich, einen auftretenden Compiler-Fehler mit der Catch-Anweisung `catch(IOException e)` abzufangen und mittels

```
RecognitionException fehlermeldung = (RecognitionException)
(e.getCause ());
```

in eine „RecognitionException“ umzuschreiben. Danach ist eine Ausgabe im MAXScript Listener durch `fehlermeldung.getDetailedMessage(...)` möglich. Erst nach der

Implementierung dieser Zeilen erlangte XL4Max ein hohes Maß an Komfort, da dadurch eine genaue Angabe zu Fehlern im Quelltext erfolgt. Eine genaue Fehlerbeschreibung erfolgt auch in der Methode `callMethod(...)`, welche für den Aufruf der XL-Methoden mittels `invoke()` zuständig ist. Gibt es ein Problem während eines solchen Aufrufs, so wird auch hier im *Catch*-Block ein genauer Fehlerbericht an die Methode `maxLPrintln(...)` übergeben.

5 XL4Max-Plug-in

Das XL4Max-Plug-in ist ein sogenanntes Utility-Plug-in, welches seine Aufgaben erst ausführen kann, nachdem es explizit aktiviert wurde. Es wird aber schon während des Starts von 3ds max das Plug-in geladen und bestimmte Methoden von XL4Max ausgeführt. So werden globale Variablen auf einen Startwert gesetzt. Der Versuch, die JVM in Form der „jvm.dll“ schon beim Start von 3ds max mit zu starten, scheiterte allerdings. Trotz gleicher korrekter Einstellungen stürzte 3ds max schon beim Start mit einer nicht nachvollziehbaren Fehlermeldung ab. Genau genommen ist es sogar von Vorteil, die JVM erst beim eigentlichen Ausführen des Plug-ins zu laden, da so ein geringerer Speicherbedarf im Betrieb von 3ds max ohne Nutzung von XL4Max erreicht wird. Leider ist es von Sun aus nicht vorgesehen, dass ein Programm die von ihm erzeugte JVM auch wieder beenden kann. Wird der entsprechende Befehl ausgeführt, so ist es danach nicht mehr möglich, eine andere, neue JVM zu starten. Deswegen bleibt die einmal von XL4Max gestartete JVM bis zum Beenden von 3ds max im Speicher erhalten. Alle Java-Methoden, die vom Benutzer durch das Plug-in aufgerufen werden, laufen deshalb auf ein und derselben JVM ab. Trotzdem ist es gewährleistet, dass im Falle eines Fehlers auf der Java-Seite kein Hängenbleiben der JVM zu Stande kommt. Wird zum Beispiel eine fehlerhafte XL-Datei zum Kompilieren an Java übergeben, so bricht der XL-Compiler dies mit einer *Exception* ab. Dies wird auf C++-Seite mit dem Befehl

```
pJEnv->ExceptionCheck ();
```

geprüft, es erfolgt eine Ausgabe im MAXScript Listener, und durch

```
pJEnv->ExceptionClear ();
```

erfolgt eine Leerung des Speichers, welcher sich Java-*Exceptions* merkt. Erst nach Einbau dieser beiden Befehle wurde es möglich, den Java-Compiler mehrmals hintereinander zu aktivieren, auch wenn Fehler auftraten.

5.1 Voraussetzungen von XL4Max

Nötig für den Betrieb von XL4Max ist verständlicherweise eine funktionierende Version von 3ds max. Getestet wurde das Plug-in unter Autodesk 3D Studio Max Release 6 sowie unter Release 8. Die eigentliche Programmierung fand allerdings mit dem Max-SDK für die Version 6 statt, weswegen auch nur hier ein höchstes Maß an korrekter Funktionsweise garantiert werden kann. Es ist leider nicht ausgeschlossen, dass sich auch im SDK ganze Klassen und Methoden geändert haben, deren Verschwinden erst beim Aufruf zum Vorschein kommen würde. Das dürfte auch der Grund sein, warum andere Plug-in-Entwickler ihre Plug-ins stets konkret für eine Version herausgeben, unter der sie auch nur laufen. Wird dann versucht, eine andere Version von 3ds max mit dem Plug-in zu nutzen, erscheint nur ein Hinweis, dass die Plug-in-Version nicht zur 3ds max-Version passt. Man ist so gezwungen, wahrscheinlich auch aus wirtschaftlichen Gründen, eine andere Version des mitunter kommerziell vertriebenen Plug-ins zu erwerben.

Des Weiteren ist für den Betrieb von XL4Max eine aktuelle Version der Java Runtime Environment (JRE) nötig. Die Programmierung erfolgte auf einem System mit der Version 1.4.2_11, also ist es zu empfehlen, mindestens diese Version der JRE einzusetzen. Erfolgreich getestet wurde das Plug-in auch mit der Version 5 von Java, konkret mit dem JRE Version 1.5.0_07. Da keine sehr versionsspezifischen Java-

Methoden genutzt worden sind, sollten an dieser Stelle auch mit zukünftigen Java-Versionen keine Probleme auftauchen. Leider kann diese Aussage nicht für den XL-Compiler bestätigt werden.

Der benötigte Compiler wird mit XL4Max mitgeliefert, da es an dieser Stelle schon kleine Änderungen am Compiler nötig machen könnten, die Schnittstelle zwischen Compiler und der C++-Seite von XL4Max neu anzupassen. Als Version des XL-Compiler-*Packages* wird eine SVN-Revision mit der Nummer 2333 vom 30.08.06 genutzt. Dieses Datum ist auch der Stand der anderen *Packages* wie „Graph“, „Grammar“ oder „XL“. Der Compiler ist damit auf dem Stand, wie er in der Software GroIMP Version 0.9.3 zum Einsatz kommt.

5.2 Installation von XL4Max

XL4Max wird mit einem leicht zu bedienenden Installationsprogramm geliefert. Dieses soll dem Benutzer im Prinzip alle nötigen Arbeiten abnehmen, so dass ein Eingreifen nur in speziellen Fällen nötig wird, wie im Fall, dass mehrere Versionen von 3ds max installiert sind.

Der Start der Installation erfolgt durch den Aufruf von „Setup.exe“. Vorher sollte zur Sicherheit ein gestartetes 3ds max geschlossen werden. Der Benutzer wird während des Installationsprozesses aufgefordert, das Zielverzeichnis für das Plug-in anzugeben. Standardmäßig wird dazu aus der Registry von Windows der Pfad zu einem installierten 3ds max ausgelesen. Sind mehrere Versionen von 3ds max installiert, so versucht der Installationsprozess den Pfad zu einem installierten 3ds max der Version 6 zu nehmen.

In dem Stammverzeichnis von 3ds max wird daraufhin ein Ordner „XL4Max“ angelegt, welcher auch das spätere Arbeitsverzeichnis von XL4Max ist. Hier werden in einer Datei namens „XL4Max.ini“ alle Programmeinstellungen abgespeichert. Außerdem kopiert das Installationsprogramm in diesen Pfad die Datei „XL4Max.jar“, welche den XL-Compiler sowie die Java-seitige Schnittstelle von XL4Max enthält, die Hilfedatei „XL4Max.chm“, eine Verknüpfung zum Internetauftritt von XL4Max, eine Deinstallationsroutine sowie mehrere Beispiele.

Im nächsten Installationsschritt wird versucht, aus der Registry den Pfad zu einer Java Runtime Environment auszulesen. Da es hierzu von Seiten von Sun leider keine ausreichende Dokumentation gab, war dieser Schritt mit Ausprobieren verbunden. So wurde festgestellt, dass in der Registry in dem Schlüssel

```
HKEY_LOCAL_MACHINE\SOFTWARE\JavaSoft\Java Runtime
Environment\CurrentVersion
```

die aktuell für Windows festgelegte Java-Version steht. Im Fall des Programmier-Systems steht dort der Eintrag „1.4“. Anhand dieser Zeichenkette wird dann im Registrierungsbaum eine Stufe tiefer gegangen, nämlich nach

```
HKEY_LOCAL_MACHINE\SOFTWARE\JavaSoft\Java Runtime
Environment\1.4
```

Im getesteten Fall der Java-Version 5 würde die Suche in den Zweig

```
HKEY_LOCAL_MACHINE\SOFTWARE\JavaSoft\Java Runtime
Environment\1.5
```

gehen.

So kann, welche Version auch immer genutzt wird, dort der entsprechende Eintrag `JavaHome` gefunden werden. Dieser beinhaltet den Pfad zum Java-Verzeichnis und wird genutzt, um in das dort liegende Unterverzeichnis „lib\ext“ die Datei „antlr.jar“ zu kopieren. Diese ist für den Betrieb des XL-Compilers zwingend notwendig und wird in dem entsprechenden Verzeichnis erwartet.

Über das Auslesen aus der Registry erfolgt, wie schon erwähnt, auch zur Laufzeit des Plug-ins das Finden der Datei „jvm.dll“, welche für den Start der JVM zuständig ist.

Für den Fall, dass der Installationsprozess den Java-Pfad nicht finden kann, da evtl. noch keine JRE installiert ist, ist der Benutzer gezwungen, nötig wird zu einem späteren Zeitpunkt den Installationsprozess erneut auszuführen oder die Datei „antlr.jar“ von Hand in das entsprechende Verzeichnis zu kopieren. Diese kann unter der Adresse [24] bezogen werden.

Hier finden sich zusätzlich Hinweise über die Aufgabe dieser Datei und auch zur Installation. Desweiteren kann, was allerdings nicht nötig sein sollte, eine neuere Version bezogen werden. Dabei ist aber nicht mehr gewährleistet, dass der XL-Compiler noch korrekt arbeitet, da dieser an die beiliegende Version angepasst wurde.

Im nächsten Schritt des Installationsprozesses hat der Nutzer noch die Möglichkeit, einen Eintrag in das Startmenu anlegen zu lassen. Hier finden sich die Einträge zur Datei „XL4Max.ini“, um ein schnelles manuelles Editieren dieser Datei zu ermöglichen. Dies sollte allerdings nur unter großer Vorsicht geschehen, da falsche Einträge ein nicht nachvollziehbares Verhalten von XL4Max nach sich ziehen können. Es werden noch Einträge zum Internetauftritt von XL4Max und eine Verknüpfung zum Deinstallationsprozess angelegt.

Im nächsten Schritt werden dann alle Dateien kopiert, und das Plug-in ist bereit zur Nutzung.

5.3 Nutzung des Plug-ins

Da XL4Max wie schon erwähnt ein Utility-Plug-in ist, findet man es in 3ds max im Bereich Utilities. Direkt nach der Installation ist dieses allerdings noch nicht im Bereich der Schnellstart-Buttons aufgelistet, und muss deshalb über den Button „More...“ gestartet werden. Mittels des ganz rechten Buttons „Configure Button Sets“ lässt sich auch eine Verknüpfung zum schnelleren Zugriff erstellen.

Die Bedienoberfläche von XL4Max lässt sich in die 3 folgenden Grundbereiche gliedern: XL-Editor, Einstellungsmenu und XL-Methods.

5.3.1 XL-Editor

Der XL-Editor ist ein einfacher Texteditor, angelehnt an den mit Windows standardmäßig mitgelieferten Editor, in älteren Versionen auch Notepad genannt. XL-Editor ist auch ähnlich aufgebaut, um einen leichten Umgang damit zu gewährleisten. Er enthält alle grundlegenden Dateifunktionen wie „New“, „Open“, „Save“ und „Save As“. Es wurde großer Wert darauf gelegt, dass Änderungen an Dateien nicht versehentlich verloren gehen, sondern an allen nötigen Stellen Rückfragen erscheinen.

Die Bearbeitungsfunktionen des XL-Editors sind festgelegt auf „Cut“, „Copy“, „Paste“, „Find“ und „Replace“ sowie eine „Undo“- und „Redo“-Funktion. Die Undo-Funktion ist allerdings auf einen Schritt beschränkt, da mehrere Schritte einen wesentlichen höheren Programmieraufwand bedeutet hätten.

Zur Unterstützung des Nutzers und zum schnelleren Arbeiten ist zusätzliche eine Toolbar vorhanden. Hier tauchen noch einmal die wichtigsten Funktionen des Editors in

Form von Icons auf. Zusätzlich unterstützen die Menus, wie heutzutage in allen Programmen üblich, auch die Verwendung von Tastenkürzeln.

5.3.2 Implementierung des XL-Editors

Die Implementierung war viel aufwändiger als im Vorfeld vermutet, und so mussten einige Kompromisse in Sachen Funktionalität eingegangen werden. Andere Features konnten noch auf anderem Weg gelöst werden, so dass der Editor größtmöglichen Komfort bietet, wie er in dieser Zeit zu realisieren war.

Das größte Manko des XL-Editors liegt sicherlich im Fehlen von *Hotkeys*, also Tastenkürzeln für die Funktionen „Open“, „Save“ und Ähnlichem. Dies ist darauf zurückzuführen, dass der Editor rein auf Windows-API-Ebene programmiert werden musste. Erste Versuche, mit Hilfe der MFC eine Implementierung zu ermöglichen, scheiterten. Nach dem Durchlesen mehrerer Forenartikel, in denen ausdrücklich vor dem Gebrauch des MFC gewarnt wird, fiel die Entscheidung dann gegen deren Nutzung. Der MFC-Support durch 3ds max ist zwar theoretisch vorhanden, aber ohne jegliche Unterstützung von Seiten Autodesk. Auch war in Erfahrung zu bringen, dass viele Dinge wie das Handling der Fenster nicht immer so funktionierten wie sie sollten, obwohl kein Fehler in der Programmierung vorliegt. Also blieb nur noch die Wahl der Windows-API. Diese Schnittstelle vereinigt in sich alle Funktionen des Systems, die man von einem Programm aus aufrufen kann. Außerdem legt sie die bei diesen Aufrufen verwendbaren Strukturen und Datentypen fest. Sie ist sozusagen Grundlage für jegliches Fenstermanagement. In höheren Programmiersprachen wie Delphi arbeitet man nicht mehr auf dieser Ebene, sondern hat einen festen Satz von Klassen (Formularen im Fall von Fenstern) mit vorgegebenen Methoden, auf welche der Programmierer zurückgreift. Erst der Compiler setzt diese dann in entsprechende Windows-API-Befehle um.

Der Vorteil der Windows-API liegt in ihrer Konsistenz. Sie existiert seit Windows Version 1.0, und hat sich seit dem fast nur in ihrem Umfang geändert. Größte Änderung seitdem ist natürlich die Einführung der Win32-API, was aber bei der Programmierung nur in Spezialfällen einen Unterschied ergibt.

Die Tastaturkürzel sind aus dem Grund nicht aktiv, da das XL-Editor-Fenster kein eigenes Message-Handling durchführt. Normalerweise würde folgender Code nach dem Erzeugen des Editor-Fensters mittels

```
CreateWindow(...);
```

stehen:

```
while (GetMessage (&msg, NULL, 0, 0))
{
    TranslateMessage (&msg);
    DispatchMessage (&msg);
}
```

GetMessage(...) ist hier zuständig für das Abholen von Nachrichten (genauer gesagt von Ereignissen) aus einer Warteschlange, und TranslateMessage(...) sowie DispatchMessage(...) sorgen für die Verarbeitung von Tastaturbefehlen sowie die Verteilung der Befehle auf die einzelnen Fenster-Prozeduren. Dieses bringt jedoch bei der Nutzung innerhalb von 3ds max mehrere Nachteile mit sich. Solange nämlich der XL-Editor offen ist, wurden alle Tastaturbefehle an eben dieses Fenster weitergeleitet (auch wenn der XL-Editor nicht den Fokus besaß), und 3ds max selbst verlor jegliche Fähigkeit zur Nutzung seiner Hotkeys. Außerdem ist durch die Gegebenheit der

Endlosschleife, die nur so lange laufen soll, bis ein WM_QUIT-Ereignis auftritt, das Problem aufgetreten, dass 3ds max das XL-Editor-Fenster nicht schließen konnte.

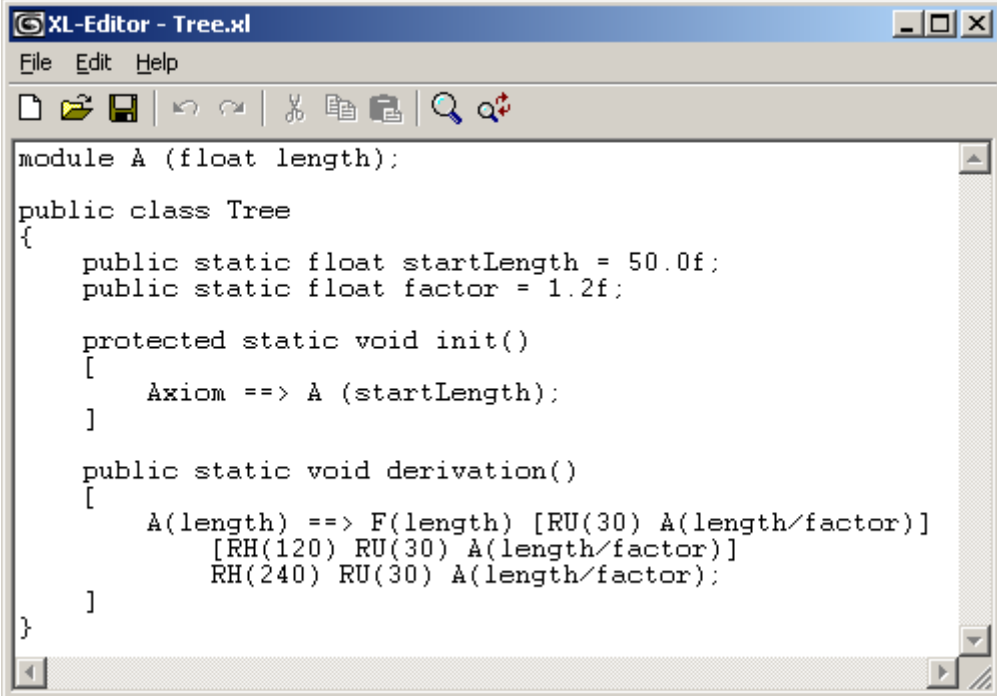
Andere Vorteile der jetzigen Lösung liegen darin, dass ein vollständiger Neuaufbau (Refresh) der Ansichten in 3ds max durchgeführt wird, wenn eine Änderung in der Szene auftritt. Dies war auch nicht gegeben, solange der XL-Editor geöffnet war. Außerdem entfällt der Umweg, das 3ds max-Hauptfenster innerhalb des Plug-ins zu *subclassen*. Die Technik des *Subclassing* bewirkt, dass man *Callbacks* umleiten kann. So wurde die Ereignisabhandlung, die 3ds max durchführte, wenn es beendet wurde, überschrieben. Diese Technik erschien allerdings zu riskant, da nie sichergestellt werden konnte, ob nicht eventuell andere wichtige Dinge auch mit überschrieben wurden beziehungsweise vergessen wurden auszuführen.

5.3.3 Gebrauch des XL-Editors

Vorweg sei zu erwähnen, dass die Programmierung eines Editors zwar Bestandteil der Bacheloraufgabe ist, aber darauf nicht der Hauptteil der Zeit verwendet wurde. So wurde der Editor nach mehr Programmierzeit als überhaupt geplant auf das wesentliche beschränkt. Er funktioniert dafür stabil, auch wenn er weniger Funktionen mit sich bringt als der Nutzer sich vielleicht erhofft.

Eine Empfehlung liegt deshalb darin, einen anderen Text-Editor zu verwenden. Als ideal hat sich während der Entwicklung der beigelegten Beispiele das Programm „Textpad“ erwiesen, zu beziehen unter [25]. Dieses Programm unterstützt Syntax-Highlighting, das gleichzeitige Bearbeiten mehrerer Dokumente sowie automatisches Einrücken von Code-Abschnitten.

In Abbildung 5 ist der XL-Editor abgebildet, und es wurde mittels der „Open“-Funktion das recht einfache Beispiel Tree.xml geladen.



```
XL-Editor - Tree.xml
File Edit Help
[Icons: Open, Save, Undo, Redo, Cut, Copy, Paste, Find, Help]
module A (float length):
public class Tree
{
    public static float startLength = 50.0f;
    public static float factor = 1.2f;

    protected static void init()
    [
        Axiom ==> A (startLength);
    ]

    public static void derivation()
    [
        A(length) ==> F(length) [RU(30) A(length/factor)]
        [RH(120) RU(30) A(length/factor)]
        RH(240) RU(30) A(length/factor);
    ]
}
```

Abbildung 5: XL-Editor mit Beispiel „Tree.xml“

Sobald eine Datei geöffnet oder ein Text unter einem bestimmten Dateinamen gespeichert wurde, erscheint der Pfad zu dieser Datei im Einstellungsmenu des Plug-ins, wie in Abbildung 6 zu erkennen.

5.3.4 Einstellungsmenu

Das Design des Einstellungsmenus ist angenähert an das Design anderer typischer Plug-ins, welche standardmäßig zu 3ds max gehören. Damit soll ein möglichst komfortabler und leicht zu erlernender Umgang mit XL4Max gesichert sein. Das komplette Rollout ist in Abbildung 6 gezeigt. Es wurde hier vorher im XL-Editor die Datei „Tree.xl“ geladen.

Das Menu (in der Benutzung spricht man auch von Rollout) ist gegliedert in die Bereiche „Editor“, „Compiler“ und „Execution“.

Der Button „XL-Editor“ im Bereich „Editor“ startet den XL-Editor oder holt ihn, falls schon gestartet, in den Vordergrund. Es kann deswegen auch immer nur ein XL-Editor-Fenster geöffnet werden.

Der Bereich „Compiler“ enthält das statische Label mit dem Dateinamen zur aktuell geladenen XL-Datei. Diese dort angegebene Datei wird beim Kompilieren an Java übergeben. Da das Label nicht sehr breit ist, gibt es die Möglichkeit, die Maus darüber zu parken. Dann erscheint ein Tooltip, welcher den kompletten Pfad anzeigt.

Der Button „Compile“ startet den XL-Compiler und übergibt dabei die angegebene Datei. Eventuell auftretende Fehler, wenn zum Beispiel keine XL-Datei geladen wurde, werden in einer Message-Box dargestellt. Auftretende Compiler-Fehler werden dagegen nur im MAXScript Listener angezeigt. Es empfiehlt sich daher, diesen auch ständig geöffnet zu haben. Zu beachten ist, dass bei jedem Start des XL-Compilers eine eventuell vorhandene Init-Methode in der XL-Datei automatisch ausgeführt wird.

Mittels „Reset“ kann die Szene wieder in den Ursprungszustand zurückgesetzt werden. Dazu werden alle in der Szene vorhandenen und für XL wichtigen Objekte gelöscht. Dies betrifft also alle Objekte unter dem Knoten *XLRoot*. Dieses ist ein *Dummy*-Knoten, welcher aus Gründen der Übersichtlichkeit nicht zu sehen ist. Existiert zum Zeitpunkt des „Reset“ noch kein *XLRoot*-Knoten, so wird ein neuer erstellt. Außerdem wird automatisch an diesen *XLRoot*-Knoten auch ein anderer *Dummy*-Knoten namens „Axiom“ gehängt. Dieser Knoten wird in vielen XL-Projekten als Hilfsknoten zum Einstieg in die Entwicklungsphasen genutzt.

Der Button „Reset & Compile“ vereint die Funktionen der Buttons „Reset“ und „Compile“ in sich, in dem erst ein Reset ausgeführt und dann der XL-Compiler aktiviert wird. Dieses dient zum schnelleren Arbeiten mit XL und ist im ständigen Umgang mit XL4Max sehr praktisch.

Der Bereich „Execution“ beinhaltet mehrere Optionen, die der Nutzer nach Belieben einstellen kann. Diese sind nur bei der Ausführung von XL-Methoden mittels „Run“ sinnvoll, wirken sich unter Umständen aber auch beim Drücken von „Step“ aus. „Run Delay“ gibt einen Wert für eine Wartezeit in Sekunden an. Bei Ausführung einer XL-Methode mittels „Run“ ist dies die Zeit zwischen den einzelnen Ausführungen. Bei schnellen Ableitungsschritten kann man so eine bessere Nachvollziehbarkeit erreichen, in dem man den Wert erhöht.

„Update Viewports“ erzwingt ein Refresh der Viewports in 3ds max zwischen den einzelnen Ausführungsschritten. Dies ist hilfreich, um Änderungen an der Szene überhaupt zu sehen, kann aber unter Umständen die Zeit für Ausführungen erhöhen.

Mittels „Animate“ können Ableitungsschritte auch animiert werden. Dies ist leider nicht immer einwandfrei, da z.B. Objekte nicht erst zu einem bestimmten Zeitpunkt erzeugt

werden können. Diese Funktion ist daher mit Bedacht anzuwenden, denn es bedarf auch einiger Anpassungen der XL-Datei. Gut zu erkennen ist dies am Beispiel „AnimAnts.xl“. Eine versehentliche Anwendung von „Animate“ kann allerdings zu unvorhersehbaren Ergebnissen führen, wie man gut am Beispiel „Koch“ aus „Fraktal.xl“ erkennen kann.

Nur wenn „Animate“ aktiviert ist, lassen sich noch zwei weitere Optionen bedienen, nämlich „Smooth Animation“ und „Smooth Creation and Deletion“.

„Key Interval“ bestimmt den Abstand, in welchem die Keys für jeden Ableitungsschritt gesetzt werden sollen.

Die genauen Bedeutungen der einzelnen Optionen, speziell der Funktionsweise von „Animate“, sind in der zum Plug-in beiliegenden Hilfe-Datei nochmals genauer erklärt.

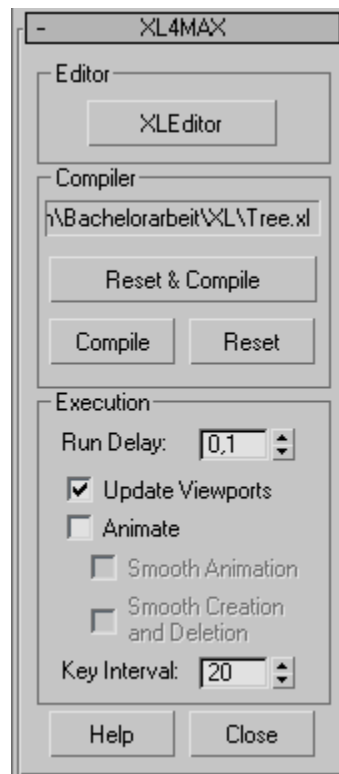


Abbildung 6: Bereich Einstellungsmenu mit Beispiel „Tree.xl“

„Help“ öffnet die im XL4Max-Verzeichnis installierte Hilfedatei. Der „Close“-Button schließt XL4Max. Dabei wird auch das Rollout „XL-Methods“ geschlossen, wobei alle erstellten Buttons zu den XL-Methoden vernichtet werden. Somit ist nach dem erneuten Öffnen des Plug-ins auf jeden Fall ein neuer Compile-Vorgang nötig.

5.3.5 Implementierung des Einstellungsmenus

Durch das Max-SDK erhält man in Visual Studio C++ die Möglichkeit, Fenster komfortabel mittels Drag&Drop zu gestalten. Das bedeutet, dass man die visuellen Elemente aus einer Liste auswählt und an den gewünschten Ort platziert. Dieses einfache Prinzip wurde jedoch nicht vollkommen ausgenutzt. So musste nach anfänglichen Fehlversuchen festgestellt werden, dass, wenn ein Button in der GUI erstellt werden soll, kein „Button“ gesetzt werden darf. Es wurde vorgegeben, stattdessen ein entsprechend großes Feld vom Typ „Custom Control“ zu erzeugen. Diesem Element muss als Attribut „Klasse“ ein „CustButton“ gesetzt werden, um tatsächlich ein Button zu erhalten. Ähnlich musste bei *Labels* und *Spinner Controls* (zum Einstellen von „Run Delay“ und „Key Interval“) verfahren werden. Dies führte in dem Moment zu

Umständlichkeit, wenn man die Buttons anordnen will, aber nicht sehen kann, welcher Button nun welche Funktion ausführen soll.

Checkboxen, wie sie für „Update Viewports“ und „Animate“ verwendet werden, sind in der Max-SDK gar nicht vorgesehen, und so blieb der Umweg über die Windows-API auch hier nicht erspart. Nur mit Hilfe der *Handles* der einzelnen Elemente war es dann möglich, überhaupt den Zustand der Checkboxen abzufragen und diese zu setzen.

5.3.6 Gebrauch des Einstellungsmenus

Da schon auf die einzelnen Funktionen eingegangen wurde, wird hier nur kurz die Vorgehensweise für den typischen Ablauf der Nutzung von XL4Max erklärt.

Nachdem im Editor eine XL-Datei geöffnet oder erstellt worden ist, wird im einfachsten Fall nur „Reset & Compile“ ausgeführt. Dies kann gerade beim ersten Start des Plug-ins eine Weile dauern, da die erste Kompilierung viel Zeit in Anspruch nimmt. Dies wird jedoch mit einem kleinen Hinweis angezeigt. Existiert in der XL-Datei eine Methode `init()`, so ist zu beachten, dass diese bei jeder Kompilierung automatisch ausgeführt wird. Dadurch kann sich die Zeit unter Umständen noch einmal erheblich erhöhen.

Solange nicht animiert werden soll, brauchen keine weiteren Einstellungen getroffen werden, und die eigentliche Arbeit kann im XL-Methods-Rollout fortgesetzt werden.

5.3.7 XL-Methods

In diesem Rollout werden nach jeder erfolgreichen Kompilierung alle als *Public* und *Static* deklarierten XL-Methoden eingetragen. Dazu wird zu jeder Methode ein Label mit dem Methodennamen erzeugt, ein Button „Step“ und ein Button „Run“.

Zu beachten ist, dass eine eventuell vorhandene Methode mit dem Namen „init“ nicht eingetragen wird, da diese Methode schon direkt nach der Kompilierung selbsttätig ausgeführt wird.

Es ist wichtig, dass entsprechende XL-Methoden nicht nur *Public* sind, damit sie in dieses Rollout eingetragen werden, sondern auch als *Static* deklariert sind. Nur dann ist es überhaupt möglich, Methoden einer Java-Klasse auszuführen, von denen kein Objekt erzeugt wurde.

Verschwindet das Rollout XL-Methods, egal aus welchen Gründen (Schließen von XL4Max, Aufruf einer anderen Utility), werden alle darin erzeugten Buttons gelöscht, und müssen durch eine neue Kompilierung wieder erzeugt werden. Dies geschieht, auch wenn es der Nutzer als lästig empfinden könnte, aus mehreren Gründen, wie zum Beispiel der Sicherheit. In diesem Fall ist davon auszugehen, dass sich die Szene geändert hat und eine Neukompilierung nötig ist, um die Szeneobjekte mit den Java-Objekten konsistent zu halten. Außerdem könnte sich die XL-Datei geändert haben. Um in dem Fall zu vermeiden, dass mit einer älteren Version gearbeitet wird, ist es erforderlich, den XL-Compiler neu zu starten. Auch implementierungstechnisch wäre es ein hoher Aufwand geworden anders zu verfahren, denn es hätte im Speicher festgehalten werden müssen, welche Buttons überhaupt erzeugt wurden, um diese dann wiederherzustellen.

5.3.8 Implementierung von XL-Methods

Die Programmierung dieses Rollouts war eine kleine Herausforderung. Denn das Rollout an sich musste schon vorher festgelegt sein, die Buttons müssen aber dynamisch erzeugt werden. Es blieb nichts anderes übrig, als ein leeres Dialogfeld mit geringer Höhe zu erstellen. Und erst nach erfolgreicher Kompilierung wird von der Java-Seite aus die

native Methode `CreateMethodButton(...)` aufgerufen, welche dann die eigentlichen Buttons mit entsprechendem Label erzeugt.

Da erst vorgesehen war, noch einen dritten Button mit „Stop“ anzulegen (diese Idee musste auf Grund unlösbarer Probleme fallen gelassen werden), finden sich im C++-Code noch immer Reste davon. Es ergaben sich nämlich Schwierigkeiten, die Methode dynamisch an die entsprechenden Buttons zu linken. Im Endeffekt war es am besten so zu implementieren, dass jedem Button eine fortlaufende Nummer zugewiesen wird („Step“ und „Run“ derselben Methode erhalten auch dieselbe Nummer), welche dann an Java beim Aufruf übergeben wird. Auf Java-Seite wird dann wiederum ausgezählt, zu welcher Methode die entsprechende Nummer gehört, und diese wird dann ausgeführt.

5.3.9 Gebrauch von XL-Methods

Abbildung 7 zeigt den Aufbau des Rollouts nach der Kompilierung des Beispiels „Tree.xl“. Da eine Methode in dieser Datei „init“ heißt, ist diese, wie schon erwähnt, nicht aufgeführt.

Das Label mit dem Text „derivation“ gibt den Namen der zweiten Methode in dieser XL-Datei an.

Der Button „Step“ lässt diese Methode genau einmal ausführen. Es erfolgt also ein Ableitungsschritt. Die Nutzung des Buttons „Run“ lässt die Methode solange ausführen, bis der Nutzer die *Escape*-Taste betätigt. Gleiches passiert beim Anklicken von „Cancel“, welches neben der jetzt angezeigten Progress-Bar im unteren Teil von 3ds max zu sehen ist. Allerdings erfolgt ein Abbruch erst nach dem gerade ausgeführten Ableitungsschritt. Dies ist leider nicht anders zu realisieren gewesen, und so kann ein Abbruch eine Weile auf sich warten lassen.



Abbildung 7: Bereich „XL-Methods“ nach Kompilierung von „Tree.xl“

6 Beispiele

Die in diesem Kapitel aufgeführten Beispiele sollen im Detail klären, wie bestimmte Funktionen von XL anzuwenden sind und was bestimmte Funktionen von XL bewirken. Dazu erfolgt eine fast zeilenweise Erklärung.

Das erste Beispiel „Simple.xml“ soll auch die grundlegenden Schritte, beginnend vom Start von 3ds max, aufzeigen, um eine XL-Datei mittels XL4Max zu erstellen. Es wird davon ausgegangen, dass sich 3ds max in seinem Auslieferungszustand befindet, ohne eigens angepasste Menus. Es folgt der Start des Compilers, um dann die Ausführung einer XL-Methode nachvollziehen zu können.

6.1 Simple.xml

Dieses sehr simple Beispiel erstellt eine Sphere (Kugel) und bewegt diese in x-Richtung.

Nach dem Start von 3ds max muss zuerst XL4Max aufgerufen werden. Zu finden ist es in der rechten Menuleiste unter dem Punkt „Utilities“, welcher durch einen Hammer (Abbildung 8) symbolisiert wird.

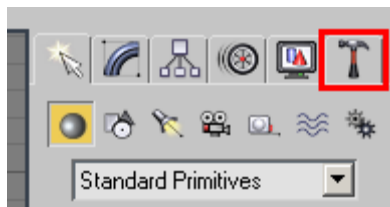


Abbildung 8: Menüpunkt zum Aufrufen der Utilities von 3ds max

In „Utilities“ existiert ein Button-Set, auf den der Nutzer beliebig Utilities platzieren kann. So kann über „Configure Button Sets“ auch XL4Max als Verknüpfung zum schnelleren Zugriff hier abgelegt werden.

Für den Start von XL4Max reicht es, „More...“ auszuwählen und in der erscheinenden Liste den Punkt „XL4Max“ auszuwählen und mittels „OK“ zu bestätigen. Daraufhin öffnet sich die Plug-in GUI. Ein Druck auf den Button „XL-Editor“ öffnet den zugehörigen Editor. Dieser zeigt nach der Installation noch keine Datei an.

Jetzt entweder über „Datei“ -> „Öffnen“ die XL-Datei „Simple.xml“ aus dem Ordner „Examples“ öffnen, oder folgenden Text von Hand eintragen:

```
public class Simple
{
    public static void move()
    [
        Axiom ==> Sphere(10);
        s:Sphere ::> s.move(10, 0, 0);
    ]
}
```

Wird der Text manuell eingegeben, muss dieser in einem beliebigen Ordner gespeichert werden. Es ist darauf zu achten, dass der Dateiname „Simple.xml“ lauten muss. Die XL-Datei und die XL-Klasse müssen den gleichen Namen besitzen, Groß- und Kleinschreibung dabei zu beachten.

Nach dem Speichern kann der Editor geschlossen werden. In der GUI sollte im Abschnitt Compiler die gespeicherte beziehungsweise geöffnete Datei zu sehen sein, wie in Abbildung 9 gezeigt.

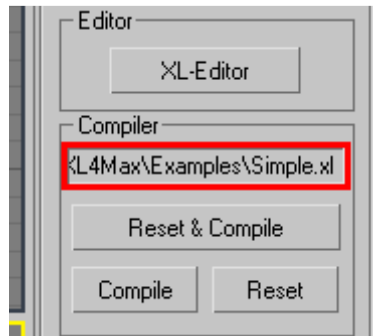


Abbildung 9: Anzeige der ausgewählten XL-Datei in der GUI

„Reset & Compile“ startet jetzt den XL-Compiler mit der angegebenen Datei. Nachdem das kleine Hinweisenfenster mit der Warteaufforderung verschwunden ist, sollte sich die GUI entsprechend Abbildung 10 verändert haben. Zu sehen ist ein Label mit dem Namen der XL-Methode „move“ und zwei Buttons „Step“ und „Run“.

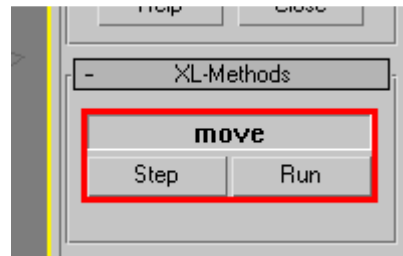


Abbildung 10: Durch den XL-Compiler erzeugte Buttons zur Methode „move“

Eine einmalige Ausführung von „Step“ lässt die Methode „move“ der XL-Datei einmal durchlaufen. Dies bedeutet, dass beide Zeilen parallel abgearbeitet werden.

```
1   Axiom ==> Sphere(10);
2   s:Sphere ::> s.move(10, 0, 0);
```

In der Szene wird nach „Axiom“ gesucht. Jedes Axiom der Szene wird durch eine *Sphere* mit der Größe 10 ersetzt. Da durch ein „Reset“ genau ein Axiom erstellt wird, entsteht also in der Szene eine *Sphere*.

Die zweite Zeile besagt, dass für jede *Sphere* der Szene eine Aktion ausgeführt wird. Allerdings existiert vor der Ausführung der Methode noch keine *Sphere*. Und auf die neu erstellte *Sphere* durch die erste Zeile hat die zweite Zeile noch keinen Einfluss, da diese parallel abgearbeitet werden.

Die zweite Zeile wirkt sich erst beim nochmaligen Drücken von „Step“ aus. Dann wird nämlich die erstellte *Sphere* angesprochen. Im Grunde wird jede *Sphere* der Szene angesprochen, jedoch existiert in dem Fall nur eine. Durch `s:Sphere` auf der linken Seite der Regel kann dann auf der rechten Seite der Regel mittels `s.move(10, 0, 0)` auf diese *Sphere* zugegriffen werden. Sie wird sozusagen in die Variable „s“ verpackt, mittels derer man dann Zugriff auf alle Methoden der Klasse „Sphere“ und ihrer Superklassen erhält. Der Befehl `move(10, 0, 0)` führt eine Bewegung um zehn Einheiten in x-Richtung aus.

Ein mehrmaliges Betätigen von Step lässt dann die *Sphere* um jeweils weitere zehn Einheiten wandern. Gleiches passiert beim Drücken von „Run“, was eine automatische Wiederholung der Methode „move“ bewirkt, bis der Nutzer auf „Cancel“ (Abbildung 11) oder *Escape* drückt.

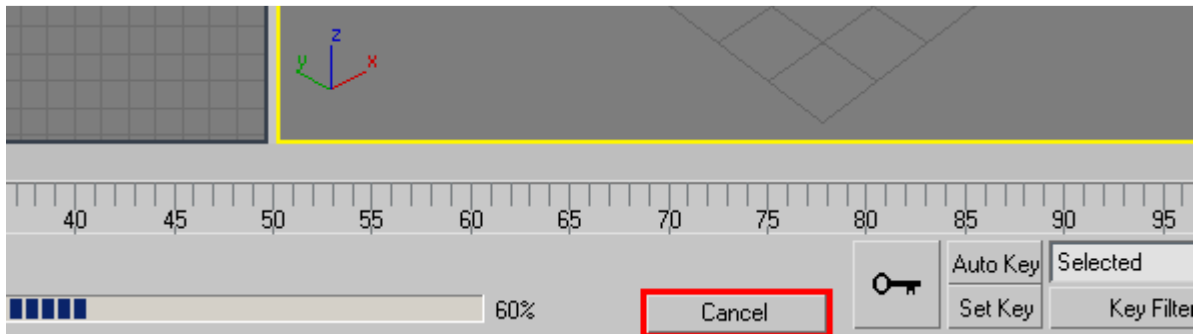


Abbildung 11: „Cancel“-Button zum Abbruch eines XL-Method-Run

6.2 GameOfLife.xl

Das Beispiel „GameOfLife.xl“ weicht in seiner Form der Implementierung in XL4Max von der in GroIMP ab. Das Prinzip ist aber dasselbe, und geht auf das von John Horton Conway entworfene System zweidimensional angeordneter zellulärer Automaten zurück. Es existiert eine Sammlung von Zellen, die in Zeilen und Spalten angeordnet sind. Als Nachbarn einer Zelle werden die acht angrenzenden Zellen angesehen. Jede Zelle ist entweder als lebendig (besitzt eine gelbe Farbe) oder als tot markiert (graue Farbe). Zu Beginn einer Simulation wird eine gewisse Anzahl von Zellen als lebendig markiert. In jedem Schritt erfolgt dann eine Überprüfung, welche Zellen zum nächsten Zeitpunkt tot oder lebendig sein werden. Dies geschieht nach folgenden zwei Regeln:

- (1) Eine lebende Zelle mit weniger als zwei lebenden Nachbarn oder mehr als drei lebenden Nachbarn ist im nächsten Simulationsschritt tot.
- (2) Eine tote Zelle mit genau drei lebenden Nachbarn ist im nächsten Simulationsschritt lebendig.

Zum „Game of Life“ gibt es unzählige Abhandlungen und Web-Seiten im Internet. Ein guter Startpunkt ist zum Beispiel die Adresse unter [26].

Eine typische Herausforderung ist die Bestimmung einer Startverteilung von lebendigen und toten Zellen, die ein oszillierendes Muster hervorbringt. Das heißt, dass nach einer gewissen Anzahl von Schritten eine schon aufgetretene Verteilung wieder vorkommt.

In der Datei „GameOfLife.xl“ finden sich mehrere Beispiele solcher Muster, die mittels „init_...“ erzeugt werden. Je nach Anzahl der Zellen ist jedoch die Ausführungsgeschwindigkeit mittels „run“ unter 3ds max unterschiedlich.

Zur Erklärung von XL wird an dieser Stelle nur auf das Beispiel „init_boxes“ eingegangen. Der dafür nötige Code ist folgender:

```

1   import static de.grogra.xl.lang.Operators.*;
2
3   public class GameOfLife {
4
5       const int neighbour = EDGE_0;
6
7       public static void init_boxes ()
8       {
9           [
10              Axiom ==>>
11              ^ for (int i : (0 : 9))
12              for (int j : (0 : 9))
13              ( [Cell(20 * i, 20 * j, 0,
14              (i >= 2) && (i < 8) && (j >= 2) && (j < 8)
15              &&((i < 5) ^ (j < 5)))]);
16              ]
17              for (apply(1)) init_neighbourhood();
18          }
19
20      private static void init_neighbourhood()
21      [
22          c1:Cell, c2:Cell,
23          ((c1 != c2) && (c1.distanceLinf (c2) < 25))
24          ==>> c1 -neighbour-> c2;
25      ]
26
27      public static void run ()
28      [
29          x:Cell(1),
30          (!(sum ((* x -neighbour-> Cell *)[state]) in (2 : 3)))
31          ==>> x(0);
32
33          x:Cell(0),
34          (sum ((* x -neighbour-> Cell *)[state]) == 3)
35          ==>> x(1);
36      ]
37  }

```

Zeile 1 enthält eine Import-Anweisung der Klasse „de.grogra.xl.lang.Operators“, womit durch die Angabe von „static“ direkt auf dessen Methoden, in dem Fall sum(...), zugegriffen werden kann.

Zeile 5 weist der Konstante neighbour den vordefinierten Kantentyp EDGE_0 zu. Statt neighbour kann auch EDGE_0 im Quelltext stehen, jedoch erhöht die Zuweisung zu Konstanten gerade bei mehreren Kanten die Übersicht.

Die Methode init_boxes() erzeugt die Zellen in der Szene.

Die Zeilen 10 und 11 sind im eigentlichen Quelltext eine Zeile und nur zur Erklärung getrennt. Das Zeichen „^“ steht für „Root“ und bedeutet, dass die Objekte an den Knoten „XLRoot“ gehangen werden. Die beiden for-Schleifen erzeugen das Schachbrett-artige Muster der Größe 10 mal 10. Durch die Schleife wird dabei jeweils eine Cell mit gegebenen Parametern der Zeile 11 an die Root gehängt. Die ersten beiden Parameter von Cell stehen für die x-Position beziehungsweise für die y-Position der Zelle. Der dritte Parameter ist die z-Position mit dem Wert 0, da alle Zellen in einer Ebene liegen sollen.

Der letzte Parameter ist die Verschachtelung von Ausdrücken, welche besagen, dass Zellen von gewisser Position ein TRUE erhalten, der Rest ein FALSE. Dieses sagt der Zelle, ob sie als lebendig oder als tot initialisiert werden soll.

Die eckigen Klammern um diese Regel besagen, dass es sich um parallel ausgeführten Code handelt, der mit spezieller XL-Syntax (wie eben der Ersetzungsregel) bestückt ist.

Die `for(apply(1))`-Anweisung der 13. Zeile steht wiederum in geschweiften Klammern der Methode, da es sich hier um sequentiellen Java-Code handelt. Durch diese Anweisung wird die folgende Methode genau ein Mal aufgerufen. Dazwischen kommt es sogar noch zu einer Überschreitung der sogenannten Transformations-Grenze. Wird diese passiert (entweder durch die `for(apply())`-Anweisung oder durch den Befehl `passTransformationBoundary()`;) werden alle angefallenen Aufgaben ausgeführt, die normalerweise erst direkt am Ende einer XL-Methode ausgeführt werden. Dies betrifft die quasi-parallelen Zuweisungen von Attributen und das Setzen von Kanten. In dem Beispiel „Game of Life“ spielt das aber keine Rolle.

Die Methode `init_neighbourhood()` der Zeile 16 setzt die Nachbarschaftsbeziehungen zwischen den Knoten. Wie schon beschrieben, hat ein Knoten zu seinen acht direkten Nachbarn eine solche Beziehung. Dazu wird in Zeile 18 mit `c1:Cell`, `c2:Cell` nach zwei Zellen gesucht, und durch die Bedingung `((c1 != c2) && (c1.distanceLinf (c2) < 25))` festgelegt, dass es sich nicht um dieselbe Zelle handeln und der Abstand der zwei Zellen 25 Einheiten nicht überschreiten darf. Treffen die Bedingungen zu, so wird zwischen diesen beiden Zellen eine Kante vom Typ `neighbour` gesetzt.

Da es sich um eine Ersetzungsregel handelt, wird diese auf alle Kombinationen von zwei Zellen angewendet. Somit kann es gerade bei einer größeren Anzahl von Zellen an dieser Stelle eine lange Zeit in Anspruch nehmen, die Nachbarschaftsbeziehungen zu setzen.

Die Methode `run()` aus Zeile 21 ist für die Simulationsschritte verantwortlich. Es werden für jede Zelle die schon erwähnten Bedingen geprüft und entsprechend die neuen Zustände gesetzt.

Dazu wird mittels `x:Cell(1)` jede Zelle abgefragt, die den Zustand 1 hat, also lebendig ist. Durch `!(sum ((* x -neighbour-> Cell *) [state]) in (2 : 3))` erfolgt die Überprüfung der Bedingung nach weniger als zwei oder mehr als drei lebenden Nachbarn. Dazu wird durch die Methode `sum(...)` eine Schleife aufgebaut, die den Zustand (`state`) jeder Zelle aufsummiert. Diese Zelle muss dabei Nachbar der Zelle `x` sein (`x -neighbour-> Cell`).

Die Klammer `(* ... *)` bedeutet, dass die eingeklammerten Objekte durch die Ersetzungsregel nicht verändert werden. Denn jedes auf der linken Regelseite vorkommende Objekt wird durch eine Ersetzungsregel beeinflusst; es sei denn, es wurde explizit mittels `(*Objekt*)` von der Ersetzung ausgeschlossen. Die Summe der Zustände wird dann mittels `in (2:3)` überprüft. Dieser Vergleichsoperator liefert TRUE, wenn die Summe im Bereich zwischen zwei und drei ist. Das „!“ am Anfang der Bedingung ist dann eine einfache Negation dieses resultierenden Ergebnisses. So wird also jede Zelle `x:Cell(1)`, für die die Bedingung gilt, durch `x(0)` ersetzt. Da auf der rechten Seite der Regel das „x“ von der linken Seite vorkommt, wird nicht eine neue Zelle erzeugt, sondern die alte Zelle mit dem Parameter „0“ aufgerufen, was bedeutet, dass ihr Zustand auf tot gesetzt wird.

Das gleiche Prinzip ist bei der zweiten Ersetzungsregel zu finden, in der die Bedingung geprüft wird, dass die Summe der Zustände der Nachbarn 3 ergibt, so dass eine tote Zelle lebendig wird.

Abbildung 12 zeigt den Startzustand des „Game Of Life“ bei der Initialisierung mittels „init_boxes“ sowie die ersten zwei Ausführungsschritte.

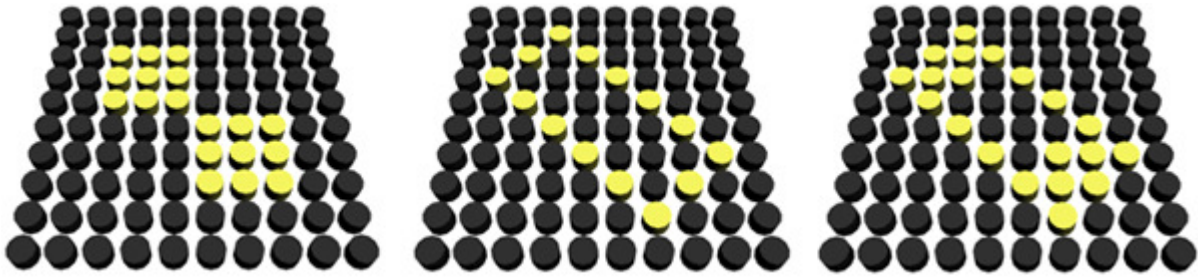


Abbildung 12: „Game Of Life“ mittels „init_boxes“ in den ersten 3 Schritten

6.3 Bush.xl

Das Beispiel „Bush“ ist wie das „Game of Life“ an der Umsetzung eines GroIMP-Beispiels angelehnt. Deshalb erfolgt hier keine tiefgehende Erklärung der Ableitungsschritte, sondern eine Erläuterung der Besonderheiten in Bezug auf XL4Max. Als erstes sei erwähnt, dass zu diesem Beispiel eine extra angefertigte Szene namens „Bush.max“ existiert. Diese sollte der Nutzer auch verwenden, um ein Resultat, wie in Abbildung 13, zu erhalten. In der vorgefertigten Szene sind neben einem Untergrund und einem Himmel auch Einstellungen zum Rendern getroffen worden. Es fällt ein Schatten des gesamten Busches, bei dem auffällt, dass die Schatten der Blätter die korrekten Umrisse anzeigen. Dies liegt am gewählten Renderer mental-ray. Dieser liegt standardmäßig 3ds max 6 dabei und erlaubt viel feinere und weitergehende Berechnungen der Szene, besonders des Lichts und seines Verhaltens. Kommt beim Rendern des Bush-Beispiels nur der Standard-Renderer von 3ds max zum Einsatz, so werden statt der korrekten Schatten nur noch die Schatten der eigentlichen Boxen erzeugt, auf denen die Blatt-Textur liegt.

Nachfolgend die Erklärung des Codes:

```

1   public class leaf extends Box
2   {
3       public leaf() {
4           this.setLength(12);
5           this.setWidth(6);
6           this.setHeight(0);
7           this.setPos(0, 6, 0);
8           this.setMaterial("Leaf");
9           // this.setMaterial("Leaf", false);
10      }
11  }
12
13  module A;
14  module S;
15
16  public class Bush
17  {
18      const float angle = 22.5f;
19      const float length = 20.0f;
20      const float dFactor = 0.6f;
21
22      public static void init ()
23      [

```

```

24             Axiom ==> D(2) A;
25         ]
26
27         public static void derive ()
28         [
29             A ==>
                Dummy
                [RL(angle) F(length) [RL(-6*angle) leaf] DMul(dFactor) A]
                RH(-5*angle)
                [RL(angle) F(length) [RL(-6*angle) leaf] DMul(dFactor) A]
                RH(-7*angle)
                [RL(angle) F(length) [RL(-6*angle) leaf] DMul(dFactor) A]
                ;
30
31             F ==> S RH(-5*angle) F(length);
32
33             S ==> F(length) [RL(-2*angle) leaf];
34         ]
35
36     }

```

In Zeile 1 beginnt die Definition einer neuen Klasse. Dies ist nicht die eigentliche Klasse der XL-Datei, zu erkennen daran, dass sie einen anderen Namen als die Datei besitzt. Die Klasse „leaf“ erbt dabei von der Klasse „Box“ und somit all seine Attribute und Methoden. Bei Erzeugung eines Objektes vom Typ „leaf“ wird in 3ds max also eine Box erzeugt. Die Darstellung eines Blattes mit dem Typ „Box“ geht darauf zurück, dass keine „Plane“ durch XL4Max erzeugt werden kann. Stattdessen erfolgt die Erzeugung einer Box mit Höhe 0, welches das gleiche Aussehen bewerkstelligt.

Im Konstruktor der Klasse „leaf“ steht zwei Mal die Anweisung `this.setMaterial(...)`, eine davon ausgeklammert. Die erste Anweisung dient zur Textur-Zuweisung aus dem Material-Editor. Dazu muss in 3ds max im besagten Material-Editor ein Material den Namen „Leaf“ besitzen, damit dieser dem Objekt vom Typ „leaf“ zugeordnet wird. Dies ist die einfachere Art, da der Inhalt des Material-Editors mit der Szene abgespeichert wird. Leider hat diese Variante den Nachteil, dass Materialien denselben Namen besitzen dürfen. Eine Zuweisung erfolgt dann nicht eindeutig. Dagegen ist die Variante aus Zeile 9 eindeutig, da in der Material-Bibliothek alle Materialien einen eindeutigen Namen besitzen müssen. Allerdings besteht der Nachteil, dass diese Bibliothek nicht in der 3ds max-Szene mit abgespeichert wird. Sie muss separat gespeichert (in dem Beispiel in der Datei „Bush.mat“) und auch separat geladen werden. Zu empfehlen ist deswegen die erste Variante.

Erwähnenswert an diesem Beispiel sind noch die Zeilen 18-20. Hier erfolgt die Zuweisung von Zahlenwerten zu Konstanten. Diese Konstanten sind dann nicht nur in normalen sequentiellen Blöcken verfügbar, sondern genauso in Blöcken mit den Ersetzungsregeln. Diese Variablen werden in dem gegebenen Beispiel auch nicht verändert (da es Konstanten sind), sondern durch den besonderen Aufbau der Regeln effektiv genutzt.

Zum Beispiel der Wert der Konstante `dFactor`: dieser wird in dem Turtle-Befehl „DMul“ angewendet. Da „DMul“ eine Multiplizierung der Dicke mit dem gegebenen Faktor bewirkt, wird in jedem Ableitungsschritt die Dicke des nächsten Astes um diesen Faktor erhöht, genauer gesagt vermindert.

Die Ableitungsschritte an sich sind recht einfach aufgebaut und bedürfen keiner genaueren Erklärung. Es sei jedoch erwähnt, dass durch die Erstellung sehr vieler Objekte in einem Ableitungsschritt die Geschwindigkeit für den nächsten Schritt rapide

abnimmt. Für das in Abbildung 13 gezeigt Beispiel brauchte das Programmiersystem fast zwei Minuten zur Erstellung.

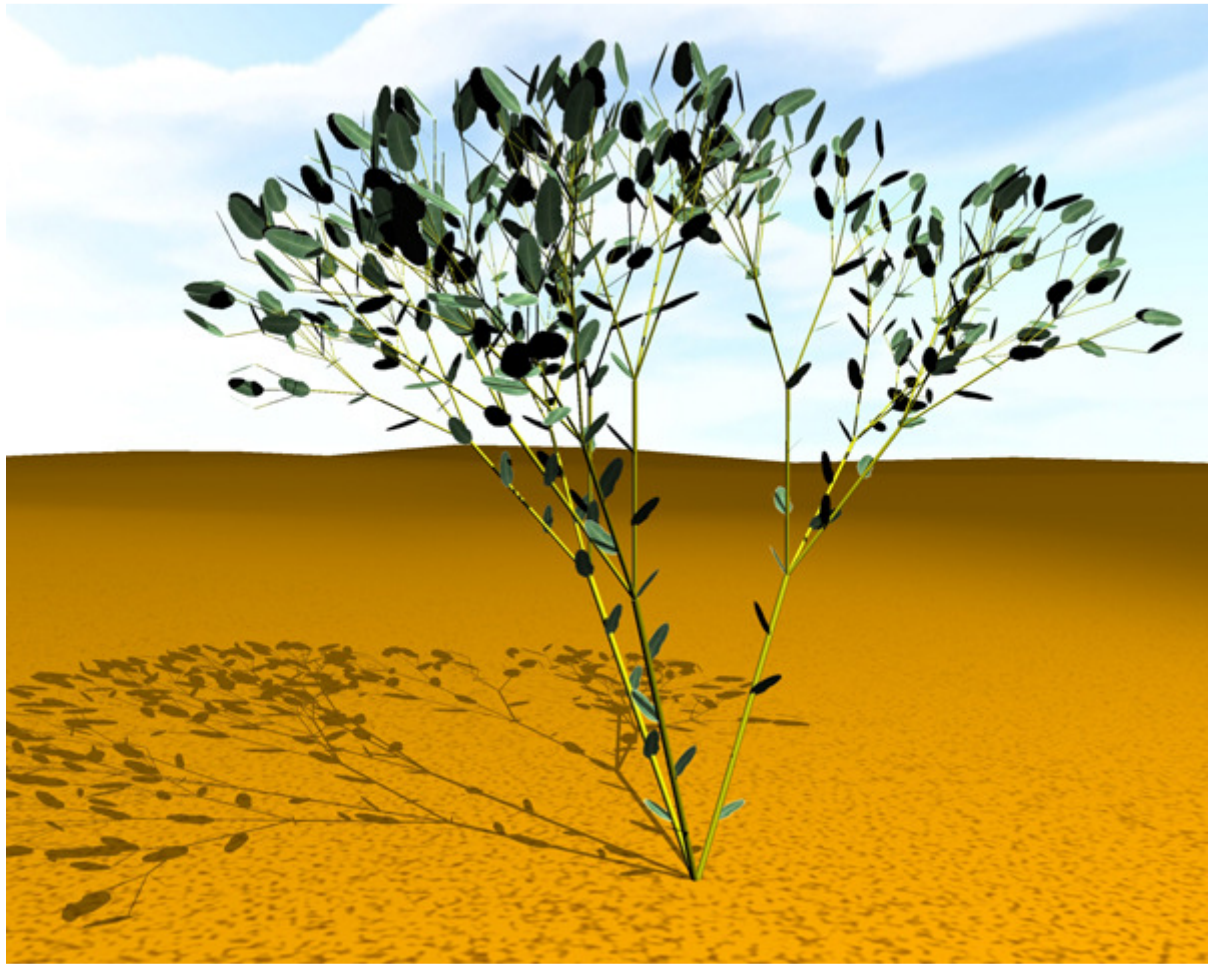


Abbildung 13: „Bush.xl“ nach fünf Produktionsschritten

7 Fazit

XL4Max hat sich trotz der verbleibenden Mankos in Sachen Stabilität und Geschwindigkeit zu einem erstaunlichen Produkt mit größerem Funktionsumfang entwickelt, als dieses vor Beginn der Arbeit abzusehen war. So war vornherein der Wunsch existent, das in GroIMP vorhandene Beispiel „Ants“ auch in der 3ds max-Umgebung simulieren zu können. Dieses Ziel wurde nicht nur erreicht in Form der „Ants.xl“, sondern sogar noch übertroffen, indem eine Speicherung der Simulationsschritte möglich ist. Die Übernahme des Beispiels erfolgte dabei ohne größere Änderungen an der Datei. Nötig waren nur geringfügige Anpassungen der Property-Bezeichner sowie am Grundgerüst der XL-Datei. Die Möglichkeit der Erstellung einer Animation („AnimAnts.xl“) bedurfte wiederum nur weniger Modifikationen.

Mit dieser Simplität bei der Erstellung komplexer Vorgänge aber auch beim Modellieren von umfangreicheren Objekten hat das Plug-in die gestellten Anforderungen erfüllt. Trotzdem ist XL4Max noch nicht als endgültiges Produkt anzusehen. So ließen sich noch viele Verbesserungen erzielen, in dem zum Beispiel der mitgelieferte XL-Editor einen größeren Komfort bieten würde. Allerdings nimmt die Entwicklung eines solch umfangreicheren Editors selbst sehr viel Zeit in Anspruch. Der effektive Nutzen bleibt zusätzlich fraglich, da es schon eine ausreichende Anzahl derartiger Programme gibt, die die gestellten Ansprüche bestens erfüllen können.

Implementierungstechnisch wäre es von Vorteil, die Umsetzung der Properties, die rein Java-seitig erfolgt, zu vereinfachen. Die Umsetzung eines weiteren 3ds max-Objektes in die Sprache XL würde an vielen Stellen im Code eine Änderung bedeuten. Wünschenswert wäre dagegen, nur die neue Klasse für das Objekt erstellen zu müssen, in der alle nötigen Informationen enthalten sind, um dass der XL-Compiler korrekt die Attribute des Objekts in Properties umsetzen kann.

Die Frage, welche Implementierungen von Objekten aus 3ds max noch erstrebenswert wären, lässt sich leicht beantworten, wenn man sich die durch XL4Max erstellten Bäume und anderen Pflanzen anschaut. Sie wirken mitunter, durch die Nutzung von Zylindern, kantig und eckig. Eine Verfeinerung der Struktur, um ein weicheres Ergebnis zu erzielen, ist auf Grund der Geschwindigkeit nicht immer möglich. Eine andere Möglichkeit wäre die Nutzung von NURBS-Oberflächen. Diese lassen sich auf einfache Weise durch Loft-Objekte erzeugen (für weiterführende Literatur siehe [14] und [27]), und bilden vom Aussehen her eine Art gebogenen Schlauch. Die so erzeugten Objekte wirken organischer trotz einer sehr einfachen Struktur. Eine Umsetzung ist im GroIMP beiliegendem Beispiel „NURBSTree“ zu finden. Eine Implementierung in XL4Max war indes nicht möglich, da das 3ds max-SDK nicht die in 3ds max vorhandene Funktion „Loft“ anbietet.

Die Umsetzung weiterer Objekte aus 3ds max - denkbar wären neben geometrischen Objekten auch Dinge wie Kameras und Lichter - könnte auf Grund der großen Anzahl fast nur noch auf Wunsch der Nutzer von XL4Max geschehen. Das gleiche gilt für Modifikatoren, welche eine bei Veränderungen des Aussehens von Objekten wichtige Rolle spielen.

Der wohl größte Nachteil der Nutzung von XL4Max im Vergleich zu XL in GroIMP liegt in der Geschwindigkeit. Trotz intensiver Mühen bei der Implementierung gelang es nicht, gleiche Beispiele annähernd so schnell ausführen zu können. Dies liegt zum einen an der Schnittstelle Java-C++, zum anderen an 3ds max. Das Programm braucht, wie Tests ergaben, eine verhältnismäßig lange Zeit, um Objekte zu erstellen. Aber auch hat 3ds max große Probleme bei der Handhabung von Szenen mit sehr vielen Knoten. So nimmt in dem Fall die Bestimmung von Hierarchiebeziehungen einen großen Zeitraum

in Anspruch. Zusätzlich wird bei vielen gleichzeitig angezeigten Objekten, was nicht unbedingt eine hohe Polygonzahl mit sich bringt, ein *Umsehen* im virtuellen Raum regelrecht zur Qual. Der Wert der angezeigten Bilder pro Sekunde (*fps*) sinkt auf ein extrem niedriges Niveau.

Hier verspricht aber die, laut Presseberichten noch im Oktober dieses Jahres erscheinende, neue Version von 3ds max Abhilfe. Bei der Entwicklung der Version 9, so wird von Seiten Autodesk's gesagt, wurde großen Wert auf diesen Bereich gelegt. Ein dazu jetzt schon verfügbares Video „Poly Speed“, zu finden unter [28], macht große Versprechungen. Sollten diese zutreffen, so wird eine baldige Portierung auf dieses System erfolgen und dem Nutzer geraten, dann mit dieser Version von 3ds max zu arbeiten.

8 Quellenverzeichnis

- [1] Turtle-Befehle (Übersicht), <http://www.uni-forst.gwdg.de/~wkurth/turtbef.html>, Stand 12.09.2006
- [2] Przemyslaw Prusinkiewicz, Aristid Lindenmayer: *The Algorithmic Beauty of Plants*. Springer, New York 1990
- [3] Winfried Kurth: *Growth Grammar Interpreter GROGRA 2.4*. Berichte des Forschungszentrums Waldökosysteme Göttingen, Ser. B, Vol. 38, Göttingen 1994
- [4] Florian Breier: *L-Systeme und andere künstliche Pflanzen*, <http://medien.informatik.uni-ulm.de/lehre/courses/ss02/Computergrafik/FlorianBreier.pdf>, Stand 12.09.2006
- [5] Grundlagen und Einsatz von parametrisierten L-Systeme, <http://olli.informatik.uni-oldenburg.de/lily/LP/flow1/page15.html>, Stand 12.09.2006
- [6] Sensitive Growth Grammars, <http://www-gs.informatik.tu-cottbus.de/grogra.de/grammars/sensitive.html>, Stand 12.09.2006
- [7] GROGRA (GROWth GRAMmar interpreter), <http://www.uni-forst.gwdg.de/~wkurth/grogra.html>, Stand 12.09.2006
- [8] Relational Growth Grammars, <http://www-gs.informatik.tu-cottbus.de/grogra.de/grammars/rgg.html>, Stand 12.09.20206
- [9] Winfried Kurth, Ole Kniemeyer, Gerhard Buck-Sorlin: *Relational Growth Grammars - A Graph Rewriting Approach to Dynamical Systems with a Dynamical Structure*. In: J.-P. Banatre, P. Fradet, J.-L. Giavitto, O. Michel (eds.), *Unconventional Programming Paradigms. International Workshop UPP2004, Le Mont Saint Michel, France, September 15-17, 2004, Revised Selected and Invited Papers, Lecture Notes in Computer Science, Vol. 3566*, Springer Verlag, Berlin, Heidelberg (2005), 56-72
- [10] Ole Kniemeyer: *Rule-based modelling with the XL/GroIMP Software*, In: Harald Schaub, Frank Detje, Ulrike Brüggemann (eds.), *The Logic of Artificial Life, Proceedings of 6th GWAL, Bamberg, 14.-16.04.2004, AKA Akademische Verlagsgesellschaft, Berlin 2004*, 56-65
- [11] XL, <http://www-gs.informatik.tu-cottbus.de/grogra.de/grammars/xl.html>, Stand 12.09.2006
- [12] GroIMP, <http://www-gs.informatik.tu-cottbus.de/grogra.de/software/groimp/index.html>, Stand 12.09.2006
- [13] Java ist auch eine Insel - 24.2 Einbinden einer C-Funktion in ein Java-Programm, http://www.galileocomputing.de/openbook/javainsel5/javainsel24_001.htm, Stand 12.09.2006
- [14] Christian Immler: *3D Studio Max 3*, Data Becker, Düsseldorf, 1999
- [15] Extrusion (Geometrie), [http://de.wikipedia.org/wiki/Extrusion_\(Geometrie\)](http://de.wikipedia.org/wiki/Extrusion_(Geometrie)), Stand 12.09.06

- [16] Java Application Profiling using TPTP, <http://www.eclipse.org/articles/Article-TPTP-Profiling-Tool/tptpProfilingArticle.html>, Stand 12.09.2006
- [17] Eclipse Test and Performance Tools Platform, <http://www.eclipse.org/tptp>, Stand 12.09.06
- [18] Code Conventions for the Java Programming Language, <http://java.sun.com/docs/codeconv>, Stand 12.09.2006
- [19] JNI Types and Data Structures, <http://java.sun.com/j2se/1.4.2/docs/guide/jni/spec/types.html#wp16432>, Stand 12.09.2006
- [20] Bug ID: 4335526 Crash when enabling remote debugging in jvm created using jni, http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4335526, Stand 12.09.2006
- [21] Java ist auch eine Insel - 1.3 Eigenschaften von Java, http://www.galileocomputing.de/openbook/javainsel5/javainsel01_002.htm#Rxx747java01002040000521F035115, Stand 12.09.2006
- [22] Java (Programmiersprache), [http://de.wikipedia.org/wiki/Java_\(Programmiersprache\)#Merkmale_der_Sprache](http://de.wikipedia.org/wiki/Java_(Programmiersprache)#Merkmale_der_Sprache), Stand 12.09.2006
- [23] The XL Language Specification (INCOMPLETE DRAFT), <http://www-ghs.informatik.tu-cottbus.de/grogra.de/xl-specification/index.html>, Stand 12.09.2006
- [24] ANTLR Parser Generator, <http://www.antlr.org>, Stand 12.09.06
- [25] TextPad - the text editor for Windows, <http://www.textpad.com>, Stand 12.09.06
- [26] Spiel des Lebens (zellulärer Automat), [http://de.wikipedia.org/wiki/Spiel_des_Lebens_\(zellulärer_Automat\)](http://de.wikipedia.org/wiki/Spiel_des_Lebens_(zellulärer_Automat)), Stand 12.09.2006
- [27] Ralf Hiete, Ralf Rendelmann, Jörg Walther: 3ds max 4, Data Becker, Düsseldorf, 2001
- [28] Autodesk - Autodesk 3ds Max - 3ds Max 9 Features, <http://usa.autodesk.com/adsk/servlet/item?siteID=123112&id=7684178>, Stand 12.09.06

9 Anhang

9.1 Aufgabenstellung

Anbindung der Programmiersprache XL an die 3D-Modelliersoftware 3ds Max über ein Plug-in

1 Aufgabenstellung

Das Softwarepaket GroIMP verfügt mit der Programmiersprache XL über weitreichende Möglichkeiten der Transformation von (Szenen-)Graphen durch Graphersetzungsgeln, u.a. für Zwecke der Vegetationsmodellierung. Um diese Möglichkeiten einem breiterem Kreis von Grafik-Modellierern zugänglich zu machen, wäre es wünschenswert, die Sprache XL in eine weit verbreitete 3D-Modelliersoftware zu integrieren. Die Software 3ds Max kann durch Plug-ins erweitert werden, es soll ein solches Plug-in zur Einbindung von XL erstellt werden. Dazu ist die Einarbeitung in die Programmierschnittstelle und die Datenstrukturen von 3ds Max (MAXScript, C++) sowie die XL-Schnittstelle zur Anbindung externer relationaler Daten (Java) erforderlich.

2 Funktionen des Plug-ins

Das Plug-in soll folgende Funktionen bieten:

- Ein in die GUI integrierter Editor erlaubt die Bearbeitung von XL-Dateien. Beim Abspeichern wird der Quelltext kompiliert, Fehlermeldungen werden gegebenenfalls in der GUI angezeigt.
- Es wird ein Mechanismus bereitgestellt, mit dem man von der GUI aus bestimmte (etwa alle parameterlosen und öffentlichen) XL-Methoden aufrufen kann.
- In XL spezifizierte Graphersetzungsgeln operieren direkt auf dem Szenengraph von 3ds Max. Dazu existiert eine für den Anwender sinnvolle Anbindung des Szenengraphen an das XL-System. Diese soll es unter anderem ermöglichen, L-System-ähnliche Regeln zu formulieren (d.h. den Turtlebefehlen F, f, RU usw. entsprechende Symbole bereitstellen).
- Es können globale Parameter vom Anwender definiert werden, die sowohl in der GUI als auch im XL-Quelltext zugänglich sind.

Für alle diese Funktionen kann GroIMP als Vorbild dienen.

3 Anbindung an XL

Die Kompilation des XL-Programmcodes erfolgt über den bestehenden XL-Compiler. Die Ausführung der Graphersetzungsgeln erfolgt über das XL-Laufzeitsystem. Da XÖ in Java implementiert ist, muss hierzu über das Java Native Interface (JNI) eine Kopplung zwischen 3ds Max und XL erstellt werden. Das 3ds Max-Plug-in instanziiert dabei eine eigene Java Virtual Machine, die Parameter für diesen Vorgang sollen im Plug-in einstellbar sein.

Auf der Java-Seite erfolgt die Anbindung an das XL-System über die Klassen und Schnittstellen im Paket `de.grogra.xl` und seinen Unterpaketen, vor allem über die Schnittstellen `de.grogra.xl.compiler.CompiletimeModel`, die die Struktur des Szenengraphen zur Zeit der Kompilation beschreibt, und `de.grogra.xl.runtime.RuntimeModel`, die die Struktur zur Laufzeit beschreibt und außerdem Möglichkeiten bereitstellt, den Szenengraph zu modifizieren. Zum `RuntimeModel` gehören noch weitere Schnittstellen im Paket `de.grogra.xl.runtime`.

Es bietet sich an, die Schnittstellen nicht direkt zu implementieren, sondern die abstrakte Basisimplementation im Paket `de.grogra.xl.impl.base` zu verwenden.

In jedem Fall muss zunächst der Szenengraph von 3ds Max genau studiert werden. Es ist zu überlegen, welche Knoten und Kanten sinnvollerweise auf der XL-Seite sichtbar und transformierbar sein sollen und wie XL-seitig erstellte Knoten 3ds Max-seitig umgesetzt werden sollen.

Nachdem die Beziehung zwischen dem Szenengraph von 3ds Max und der XL-Umsetzung geklärt ist, muss für eine sinnvolle Auswahl an geometrischen Objekten eine einfache Möglichkeit geschaffen werden, nach diesen zu suchen (auf der linken Seite von Regeln und in Suchausdrücken) sowie diese zu erzeugen. Hierzu ist einer der Mechanismen von XL zu nutzen, die in der XL-Spezifikation definiert sind, etwa `boolean`-Methoden, Klassen und `Predicate`-Unterklassen zur Suche und Methoden oder Klassen zur Erzeugung.

Neben den genannten Aspekten, die die Topologie des Szenengraphen betreffen, stellt XL auch eine Schnittstelle für Attribute zur Verfügung (diese werden im Quelltext über eckige Klammern angesprochen, etwa `sphere[radius]`). Während der Kompilation werden dazu die `XLField`-bezogenen Methoden vom `CompiletimeModel` benutzt, während der Laufzeit die `FieldSequence`-bezogenen Methoden von `RuntimeModel`. Eine sinnvolle Implementation dieses Mechanismus, der neben dem vereinfachten Zugriff auf Attribute auch quasi-parallele Zuweisungen ermöglicht, ist wünschenswert.

4 Java-Programmierschnittstelle für 3ds Max

Innerhalb von XL sollen möglichst viele Aspekte des 3ds Max-Szenengraphen ansprechbar sein. Es reicht daher nicht aus, nur die beschriebene Implementation von `CompiletimeModel` und `RuntimeModel` zur Verfügung zu stellen, sondern es muss auch eine allgemeine (nicht XL-bezogene) Java-Programmierschnittstelle geben, d.h. eine Sammlung von Java-Methoden zum Erstellen und Verändern des 3ds Max-Szenengraphen und seiner Attribute (vor allem Geometrie und Aussehen) sowie zur Benutzung allgemeiner 3ds Max-Funktionen (etwa Konsolenausgabe). Da das 3ds Max-API sehr umfangreich ist, können nicht alle Funktionen berücksichtigt werden; hier ist eine sinnvolle Auswahl zu treffen.

5 Dokumentation

Eine Quelltextdokumentation (Java, C++) soll es anderen ermöglichen, sich mit den internen Strukturen vertraut zu machen und eigene Erweiterungen zu implementieren.

Außerdem ist eine Dokumentation für den 3ds Max-Anwender zu erstellen, die folgende Punkte berücksichtigt:

- Die Installation des Plug-ins muss beschrieben werden. Hierzu gehört auch eine Beschreibung der zusätzlich benötigten Software einschließlich Installationshinweisen, z.B. für Java.
- Die Anwendung des Plug-ins muss dokumentiert werden: Um welche Funktionen wird 3ds Max erweitert, welche Optionen sind einstellbar, welche Menüeinträge neu?
- Auch wenn keine ausführliche Beschreibung des Graphersetzungs-Konzeptes verlangt wird, sollte die Dokumentation an Beispielen die Anwendung von XL für 3ds Max einführen.
- Im Zusammenhang mit dem letzten Punkt müssen die anwendernahen Klassen und Methoden beschrieben werden, die das Plug-in auf der Java/XL-Seite bereitstellt.
- Software und Dokumentation sollen auf einer Internetseite bereitgestellt werden.