Bachelor-Arbeit

Udo Bischof

13.September 2006

Thema:

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

Betreuer:

Prof. Dr. Winfried Kurth

Ole Kniemeyer

1. Aufgabenstellung

Das Softwarepaket GroIMP[1] verfügt mit der Programmiersprache XL über weit reichende Möglichkeiten der Transformation von (Szenen-)Graphen durch Graphersetzungsregeln, u.a. für Zwecke der Vegetationsmodellierung. Um diese Möglichkeiten einem breiteren Kreis von Grafik-Modellierern zugänglich zu machen, wäre es wünschenswert, die Sprache XL in eine verbreitete 3D-Modelliersoftware zu integrieren. Die Software Maya kann durch Plug-ins erweitert werden, es soll ein solches Plug-in zur Einbindung von XL erstellt werden. Dazu ist eine Einarbeitung in die Programmierschnittstelle und die Datenstrukturen von Maya[2] (MEL[3], C++[4]) sowie die XL-Schnittstelle zur Anbindung externer relationaler Daten (Java[5]) erforderlich.

2. Einleitung

Diese Arbeit befasst sich mit der Lösung der eben genannten Aufgabe und deren Anwendungsmöglichkeit. Ziel ist dabei nicht, sämtliche Kleinheiten der Programmierung zu erläutern, sondern eher, dem Anwender ein Mittel in die Hand zu geben, ohne große Vorkenntnisse (außer eventuell Java-Kenntnissen) einfache L-Systeme zu entwerfen und umzusetzen. Dabei ist der Umgang mit XL4Maya und die zur Verfügung stehenden Klassen und Methoden Hauptaugenmerk. Absichtlich wird auf komplizierte Klassendiagramme[6] und Codeerläuterungen verzichtet, da dies einem Anwender den Zugang zu XL4Maya nicht erleichtern würde.

Im Kapitel 3 wird zunächst allgemein auf die Grundlagen von L-Systemen, relationalen Wachstumsgrammatiken und XL als deren Implementierung eingegangen. Um bestimmte Hintergründe und Probleme speziell im Bezug auf die Implementierung näher zu beleuchten, werden im Kapitel 4 neben verschiedenen Erläuterungen des Maya-Szenegraphen und einigen Lösungsstrategien auch einige Klassen und MEL-Skripte vorgestellt. Hierbei wird auf eine ausführliche Darlegung und Dokumentation des Codes verzichtet, sondern nur einige Kernpunkte erläutert.

Das Kapitel 5 beschäftigt sich nun mehr mit der Anwendung von XL4Maya und deren Wirkungsweise auf Maya. Neben den L-System-Befehlen und Primitivobjekten werden auch die wichtigsten aus XL bekannten umgesetzten Konzepte vorgestellt. Es werden die notwendigen Klassen, Konstruktoren und Methoden aufgezeigt und erläutert.

Um dem Anwender einen Leitfaden für XL4Maya offen zu legen, bietet Kapitel 6 einen Einstieg. Neben der Installation ist auch die GUI Thema.

Zu guter Letzt sind in Kapitel 7 fünf Beispiele genauer erläutert und mit Quelltext aufgeführt. Sie verwenden einen Großteil der in Kapitel 5 vorgestellten Konzepte und können somit hilfreich bei der Entwicklung eigener Beispiele sein.

Noch ein paar Worte zu verwendeten Begriffen und gezeigten Quelltexten bzw. Formatierungen. In L-Systemen spricht man üblicherweise von Befehlen. In XL sind diese Befehle Knoten, um genau zu sein: Instanzen einer Klasse wie F oder M, somit

1 Vgl.: [1].
2 Grundlagen von Maya siehe [2] und [3].
3 Vgl. [4], [5] und [6].
4 Zur Programmierung in C++ siehe [7] und [8].
5 Zur Programmierung in Java siehe auch [9].
6 Eine vollständige Java-Klassenreferenz befindet sich in [22] bzw. auf der beiliegenden CD.

sind beide Begriffe gültig. Bei Quelltexten steht in der ersten Zeile des Listings in eckigen Klammern, aus welcher Programmiersprache dieser stammt. Dies gehört natürlich nicht zu dem eigentlichen Programm. Zudem sind die meisten Quelltexte hier nur ausschnittweise aufgeführt, und zumeist wurde bei XL-Ausschnitten der notwendige Rahmen (die Definition der Klassen und Methoden) weggelassen. Methodendefinitionen und Konstruktoren werden von einem Punkt (●) angeführt und stellen keinen gültigen Quellcode dar.

Der Inhalt dieser Arbeit besteht aus:

- dem schriftlichen Anteil,
- dem Plug-in als Setup-Routine für Maya 7.0 und Maya 8.0,
- einer Onlinehilfe als kompiliertes HTML, welche dem Setup-Packet beiliegt,
- dem Quellcode des Java- und C++-Teils,
- einer Java-Klassenreferenz als JavaDoc,
- den MEL-Skripten,
- fünf Beispielen mit XL-Code und zugehörigen Maya-Szenen
- einer gestalteten Webseite unter http://xl.student-by-default.de

Sämtliche Inhalte sind über die aufgeführte Webseite bzw. die der gebundenen Arbeit beiliegenden CD erreichbar.

3. Grundlagen zu XL

XL ist eine auf Java aufbauende, regelbasierte Programmiersprache. Grundlage dafür sind relationale Wachstums-Grammatiken. XL vereinigt die Stärken einer imperativen Programmiersprache mit regelbasierten Paradigmen. Die nachfolgend beschriebenen L-Systeme sind ein Spezialfall von XL.

Die anschließenden Kapitel sind nur ein schneller Überblick über XL und deren Grundlagen. Es wird an einigen Stellen etwas vorgegriffen, doch es werden nur Techniken aufgezeigt, die XL4Maya auch beherrscht. Des Weiteren ist dieses Vorwissen hilfreich, damit die in den späteren Kapiteln aufgeführten Beispiele verständlicher sind.

3.1. L-Systeme

Aristid Lindenmayer (1925 – 1985) entwickelte ein Verfahren, um biologisches Pflanzenwachstum algorithmisch zu beschreiben[1]. Grundlage dafür bildet eine Grammatik mit einem Startsymbol und mindestens einer Ableitungsregel. Grundlegende Befehle und zugleich Elemente des Alphabets der Grammatik sind:

F - zeichne einen Strich
\+ - Drehung nach links
\- - Drehung nach rechts
[- Speichere Position
] - Rufe gespeicherte Position ab

1 Vgl.: [10].

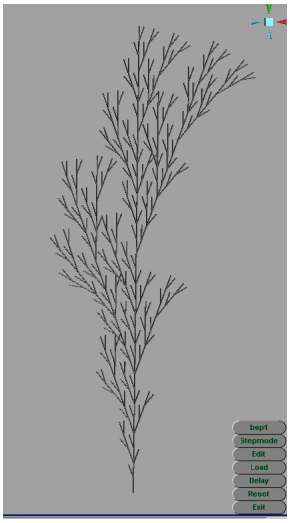
Die Winkel für die Drehung sind implementierungsabhängig. Somit ist die Kochkurve im Falle von Drehungen um 45 bzw. -45 Grad beschreibbar durch:

```
Start: F
Regel: F->F + F - - F + F
```

Mit diesen wenigen Symbolen, die auch Turtlebefehle genannt werden, kann man bereits recht komplexe pflanzliche Strukturen erzeugen. Über Erweiterungen der oben genannten L-System-Befehle können weitaus komplexere Strukturen modelliert werden. Eine Auswahl dieser Befehle sind aufgelistet unter

http://www.uni-forst.gwdg.de/~wkurth/turtbef.html

Folgend ein paar Beispiele für L-Systeme[1] umgesetzt in XL. Jeweils neben der Abbildung befindet sich der zugehörige Code:



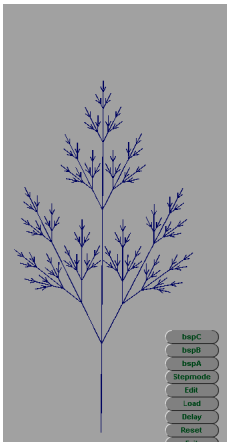

```
public static void bspA()[
Axiom ==> F;
F ==> F[RU(20)F]F[RU(-20)F][F];
]
```


```
module x extends Transform;
(…)
public static void bspB()[

Axiom ==> x;
x ==> F RU(-22.5) [ [x] RU(22.5) x ]
      RU(22.5) F [ RU(22.5) F x ]
      RU(-22.5) x;
F ==> F F;

]
```

[1] Algorithmen entnommen aus: [11], S.25.




```
module x extends Transform;
(…)
public static void bspC()[

Axiom ==> x;
x ==> F [RU(25.7) x] [RU(-25.7) x] F
x;
F ==> F F;

]
```

3.2. XL als Implementierung relationaler Wachstums-Grammatiken

Relationale Wachstums-Grammatiken wenden das formale Prinzip der Ersetzungsregeln und Grammatiken auf Graphen an. Hierbei sind Knoten Objekte der darunter liegenden Programmiersprache, im Falle von XL ist es Java. Es können Suchanfragen und Veränderungsoperationen auf den Graphen angewendet werden. Innerhalb dieser Grammatiken ist es möglich, imperative Code-Blöcke einzufügen, um eine Mischung aus regelbasierter und imperativer Programmierung zu ermöglichen. Die weiter oben beschriebenen L-Systeme sind ein Spezialfall der relationalen Wachstumsgrammatiken.

XL ist die Implementierung einer solchen relationalen Wachstumsgrammatik. Innerhalb der Regeln kann „normaler“ Java-Code eingefügt werden, dessen lokale Variablen in bestimmten Regeln sichtbar sind und für bestimmte Suchanfragen oder Eigenschaften von Knoten verwendet werden können. Um die Funktionalität von XL zu vergrößern, können weitere Java-Bibliotheken gelinkt und verwendet werden.

Die bereits weiter oben beschriebene Koch-Kurve kann in XL mit folgendem Code erzeugt werden:

```
[XL]
Axiom ==> F(1) RU(120) F(1) RU(120) F(1);
F(x)  ==> F(x/3) RU(-60) F(x/3) RU(120) F(x/3) RU(-60) F(x/3);
```

Das Axiom ist üblicherweise der Startknoten. Hierbei sind F, Axiom und RU Klassen von Java, deren Instanz in XL einen Knoten darstellt. Die Variable x ist ein Feld der Klasse F.[1]

Die Regeln in XL werden quasi-gleichzeitig abgearbeitet, dass heißt, Veränderungen werden erst dann wirksam, wenn alle Regeln abgearbeitet wurden.

3.3. Aufbau einer XL-Datei

Grundlegend entspricht die Syntax und Semantik von XL der von Java, mit einigen Veränderungen. Doch ist die Struktur einer XL-Datei ähnlich der einer Java-Datei. Im Kopf müssen benötigte *imports* eingefügt werden. XL4Maya importiert bereits

[1] Beispiel entnommen aus: [12].

eine Zahl von *packages*, sodass für den Beginn kaum eigene *imports* notwendig sind. An dieser Stelle können aber prinzipiell alle Java-Klassenbibliotheken oder eigene Klassen verwendet werden.

Danach folgt die Definition der Hauptklasse und der zusätzlichen Klassen. Wichtig ist zu wissen, dass XL eine Klasse braucht, um zu arbeiten. In dieser sucht der Compiler nach *public*-deklarierten Methoden, um diese auf Wunsch des Anwenders auszuführen. Es muss also in dieser Klasse, die den gleichen Namen wie die Datei tragen muss (im Bezug auf XL4Maya) mindestens eine *public*-deklarierte Methode existieren. Zudem müssen alle Methoden dieser Klasse *static* deklariert sein.

In den selbst definierten zusätzlichen Klassen kann enthalten sein, was der Anwender möchte. Nachfolgend ein kurzes Beispiel:

```
[XL]
import ... // an dieser Stelle imports einfügen
public class myClass{
      public static void method1(){
      }
}
```

Dies sieht aus wie eine normale Java-Klasse. Unterschiede ergeben sich erst, wenn man Ableitungsregeln benutzen möchte. Dann müssen für die Methode statt geschweiften Klammern eckige verwendet werden.

```
[XL]
…
public static void method1()[
      ... //an dieser Stelle Regeln definieren
]
```

Innerhalb dieser eckigen Klammern können dann wiederum imperative Blöcke über geschweifte Klammern eingefügt werden.

3.4. Imperative Blöcke

In einem imperativen Block kann der Anwender beliebige Java-Anweisungen einsetzen, Variablen deklarieren oder Klassen instanzieren. Die Semantik und Syntax entspricht der von Java. Ein kleines Beispiel:

```
[XL]
…
public static void method1()[
      { double x = 5; }
      Axiom ==> F(x);
]
…
```

Die in dem imperativen Block festgelegten Variablen bleiben über die Ableitungsregel erhalten, sie können dort sogar neu festgelegt werden.

3.5. Knoten, Klassen und Module

Es ist ohne weiteres möglich, neben der Hauptklasse weitere Klassen zu definieren, die dann instanziert werden. Wenn man ohne Instanzierung arbeiten möchte, müssen entsprechend die aufzurufenden Methoden *static* deklariert sein.

Diese eigenen Klassen können auch, wie das Vererbungsprinzip in Java es ermöglicht, von vorhanden beliebigen oder XL4Maya-Klassen ableiten. Auf diese Art kann man eigene Knotentypen entwickeln, d.h. zusätzliche Berechnungen ausführen oder Werte speichern. Im folgenden kurzen Beispiel wird dies gezeigt:

```
[XL]
public class eigenerKnoten extends Transform{
      public eigenerKnoten(double drehung){
            super();
            this.rotateBy(drehung, 0, 0);
      }
}

public class Beispiel{
      public static void methode1()[
            Axiom ==> eigenerKnoten(45.0) F(3);
      ]
}
```

In diesem einfachen Beispiel wird ein eigener Knoten definiert, der nach seiner Erzeugung eine Drehung um die übergebene Variable *drehung* durchführt. Da der Konstruktor der übergeordneten Klasse, nämlich *Transform*, aufgerufen wird, ist auch in Maya ein *Transform*-Knoten erzeugt worden. Leitet man nur von *Node* ab, kann man an dieser Stelle die *rotateBy*-Methode nicht anwenden, da kein physikalischer Knoten in Maya existiert und in der Klasse *Node* die dafür notwendigen Methoden nicht implementiert sind.

In XL ist es möglich, so genannte Module zu deklarieren. Diese können Werte speichern und wie Knoten eingesetzt werden. Weitere Erläuterungen dazu befinden sich im Kapitel 5.1.6.

3.6. Ableitungen

In XL gibt es mehrere Arten von Ableitungen. Bereits in den vorigen Beispielen verwendet, seien sie hier noch einmal erklärt:

==> Ersetzt die linke Seite durch die rechte und hängt alle Kinder der linken Seite an die Knoten der rechten. Dies wird auch Einbettung genannt. Alle ein- und auslaufenden Kanten der linken Seite werden dann auf die rechte Seite übertragen.

==>> Ersetzt die linke Seite durch die rechte, hängt aber nicht die Kinder der linken Seite an die Knoten der rechten. Wenn diese Knoten nicht anderweitig in den Graphen eingefügt werden, werden sie gelöscht.

::> Dies ist keine Ersetzung im eigentlichen Sinne, sondern eine Ausführung der rechten Seite für jede gefundene linke. Dies müssen imperative Anweisungen sein, eingefügt in geschweifte Klammern.

In XL gibt es noch weitere Operatoren, aber für XL4Maya sind nur diese relevant.

3.7. Suchmuster und Bedingungen

Ein Suchmuster ist die linke Seite einer Regel. Es können damit Knoten oder bestimmte Muster, also Anordnungen von Knoten im Graphen gesucht werden und somit Operationen auf eine Gruppe von Knoten eingeschränkt werden.

Ein paar Beispiele für Suchmuster:

```
x:Sphere, y:Cube
```

sucht eine Kugel und einen Quader, die im Graphen nicht miteinander verbunden sind.

```
x:Sphere y:Cube
```

Diesmal muss der Quader im Graphen direkt unter der Kugel hängen. Dasselbe bedeutet in einer anderen Schreibweise:

```
x:Sphere –successor-> y:Cube
```

Das x: bedeutet, dass der Knoten in der Variable x festgehalten wird und auf einer rechten Seite wieder verwendet werden kann.

```
[XL]
x:Sphere ==> x Cube;
```

Dieses Beispiel zeigt die Anwendung des oben Erwähnten: Die jeweils gefundene Kugel wird in x gespeichert (die Instanz) und auf der rechten Seite wieder eingefügt. Somit wird die gefundene Kugel durch sich selbst ersetzt und im Graphen darunter ein Cube erzeugt.

In manchen Situationen hilfreich ist das Konstrukt `(*Knoten*)`. Es sorgt dafür, dass der gefundene Knoten nicht ersetzt wird, sondern nur für das Suchmuster gilt.

An dieser Stelle ein kleiner Hinweis speziell zu XL4Maya: Da Maya in seiner Standardszene bereits einige Objekte hat, sollte mit anfänglichen Suchmustern wie

```
[XL]
x:Transform ==> ;
```

sehr vorsichtig umgegangen werden, da sonst Objekte gelöscht werden können, die Maya noch braucht oder blockiert.

3.8. Anwendung von Methoden

In den späteren Kapiteln wird eine große Anzahl von verschiedenen Methoden aufgezeigt, welche dem Anwender zur Verfügung steht. Hier soll kurz gezeigt werden, wie diese Methoden angewendet werden:

```
[XL]
Sphere ==> Cube.(setShader(„materialSG“));
```

Für den Fall, dass mehrere Methoden auf einen Knoten bei seiner Erzeugung angewendet werden sollen, können die Methoden einfach nacheinander innerhalb der Klammern, jeweils getrennt durch ein Komma aufgeführt werden:

```
[XL]
Axiom ==> y:Cube.(setShader(…), setRadius(…), …);
```

Diese beiden Regeln zeigen zwei Möglichkeiten, wie Methoden auf einen Knoten angewendet werden können. Hierbei sind es Methoden definiert in den Knoten. Nun ist es möglich, dass recht viele Methoden bei der Erzeugung eines Knotens

angewendet werden sollen. In diesem Fall ist es ratsam, einen neuen Knotentyp zu definieren, wie in Kapitel 3.5 bereits beschrieben. Die verschiedenen Parameter können über den Konstruktor an die Methoden weitergegeben werden. Dies bläht den Quellcode einer Regel nicht unnötig auf und hilft, die Übersicht zu bewahren.

Eine andere Variante ist, die Methodenanwendungen in einem imperativen Block am Schluss der Regel durchzuführen. In diesem Block ist die zugewiesene Knotenvariable noch immer sichtbar:

```
[XL]
Axiom ==> y:Cube { y.setShader(…); };
```

Wichtig ist, dass die Regel durch ein Semikolon abgeschlossen werden muss. In dem imperativen Block können nun so viele Berechnungen durchgeführt und Methoden angewendet werden wie notwendig.

4. Maya: Grundlagen und Implementierung

Da XL in Java implementiert wurde und Maya eine API für C++ einsetzt, ist es notwendig, aus C++-Code heraus eine Java-Virtual-Machine zu starten, um den XL-Compiler darauf laufen zu lassen.

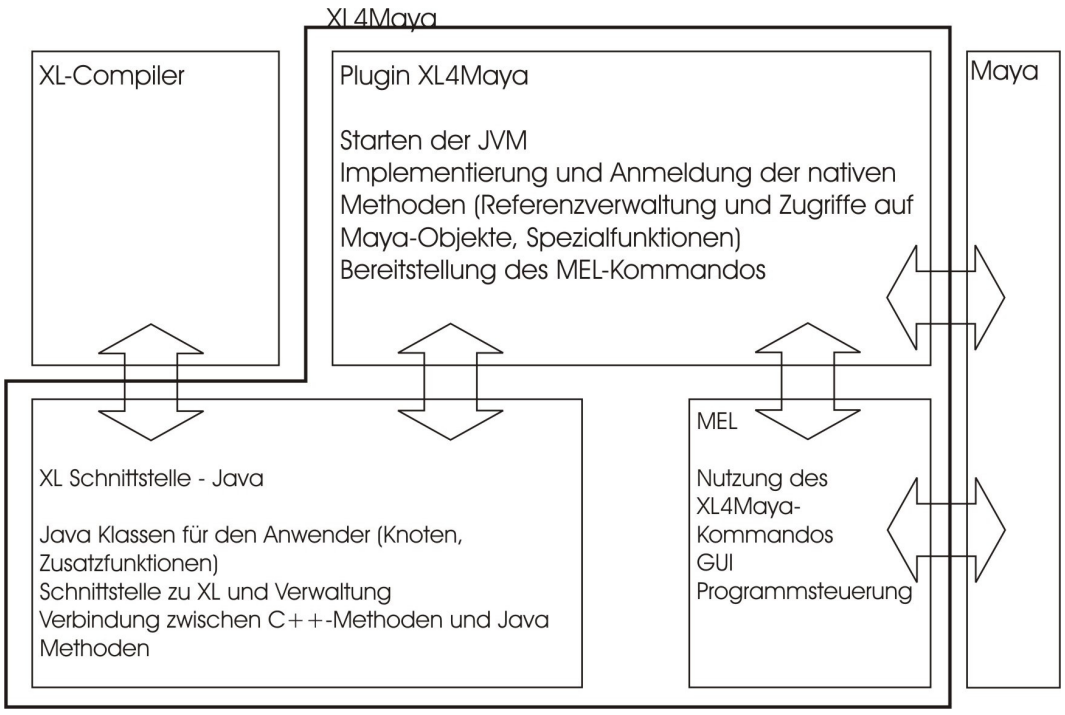


Abbildung 1: Struktur und Einbindung von XL4Maya

Abbildung 1 zeigt den Aufbau von XL4Maya als Bindeglied zwischen dem XL-Compiler und Maya. Im Folgenden werden alle Teile aufgezeigt und erläutert, zusammen mit wichtigen Teilen des Programmcodes. Der Schwerpunkt wird hier auf den Klassen und Methoden liegen, die das Verstehen der Interna von XL4Maya ermöglichen. Darüber hinaus liegt dem Programm eine vollständige Klassen- und Methodenreferenz bei, welche alle verwendbaren Klassen und Methoden erläutert.

4.1. Maya

Die erste Version von Maya wurde 1998 von der Firma Alias|Wavefront angeboten. Es wird hauptsächlich in der Film- und Effektbranche eingesetzt und besitzt umfassende Möglichkeiten, realistische Szenen und Animationen[1] zu erstellen. Die Hauptfähigkeit liegt dabei auf Charakteranimationen, die schwierigste aller Disziplinen des CGI (computer generated imaging), aber auch die gute Möglichkeit der Erweiterung und Anpassung an bestimmte Anforderungen. Somit ist dieses Softwarepaket, welches in den Versionen Maya Complete, Maya Unlimited[2] und Maya PLE angeboten wird, eines der weltweit führenden und meist eingesetzten.

Inzwischen ist Maya von der Firma Autodesk aufgekauft und in dessen Portfolio übertragen worden. In Zukunft soll so die Zusammenarbeit mit anderen 3D-Paketen wie 3D-Studio Max und diversen Postproductions-Software gestärkt werden.

Bevor auf die eigentliche Einbettung und Verwendung von XL eingegangen wird, sind im Folgenden noch ein paar wichtige Themen zu behandeln, welche die Motivation einiger Ansätze untermauern. Sie stellen nicht die Funktionsweise von XL bzw. XL4Maya dar, sondern beziehen sich allgemein auf Techniken in Maya und seinen Szenegraphen, welche die Grundlage von XL4Maya darstellen.

4.1.1. Der Maya-Szenegraph

Die Szeneinformationen, also der Szenegraph, ist in Maya im Vergleich zu anderen 3D-Programmen recht komplex. Er ist unterteilt in zwei unabhängige Sichten, den Dependency-Graph (DG) und den Directed-Acyclic-Graph (DAG). Der DG ist eine Obermenge des DAG, d.h. jeder DAG-Knoten ist zugleich ein DG-Knoten, aber nicht umgekehrt. Beide Graphen können unabhängig voneinander betrachtet und editiert werden, allerdings wirken sich Änderungen des einen Graphen unter Umständen auf den anderen aus. Die Unterschiede dieser beiden Graphen liegen im Folgenden:

Der DAG stellt die hierarchische Beziehung zwischen DAG-Knoten dar, d.h. *parent-child*-Beziehungen; die Form (d.h. die geometrischen Informationen wie Polygone, Kanten und Vertices) durch die *Shape*-Knoten und die Position im Raum über *Transform*-Knoten. Ein *Transform*-Knoten ist im Prinzip nichts weiter als eine Transformationsmatrix, welche Rotation, Translation und Skalierung relativ zu seinem *parent* enthält. In einem *Shape*-Knoten sind die Geometrie-Informationen im lokalen Raum gespeichert. Ein *Shape*-Knoten ohne einen übergeordneten *Transform*-Knoten ist nicht erlaubt, da Maya sonst nicht weiß, wie die Geometrie im Raum zu platzieren ist.

[1] Weiteres zu Animationen mit Maya siehe auch: [13].
[2] Informationen zu Maya Unlimited: [14].

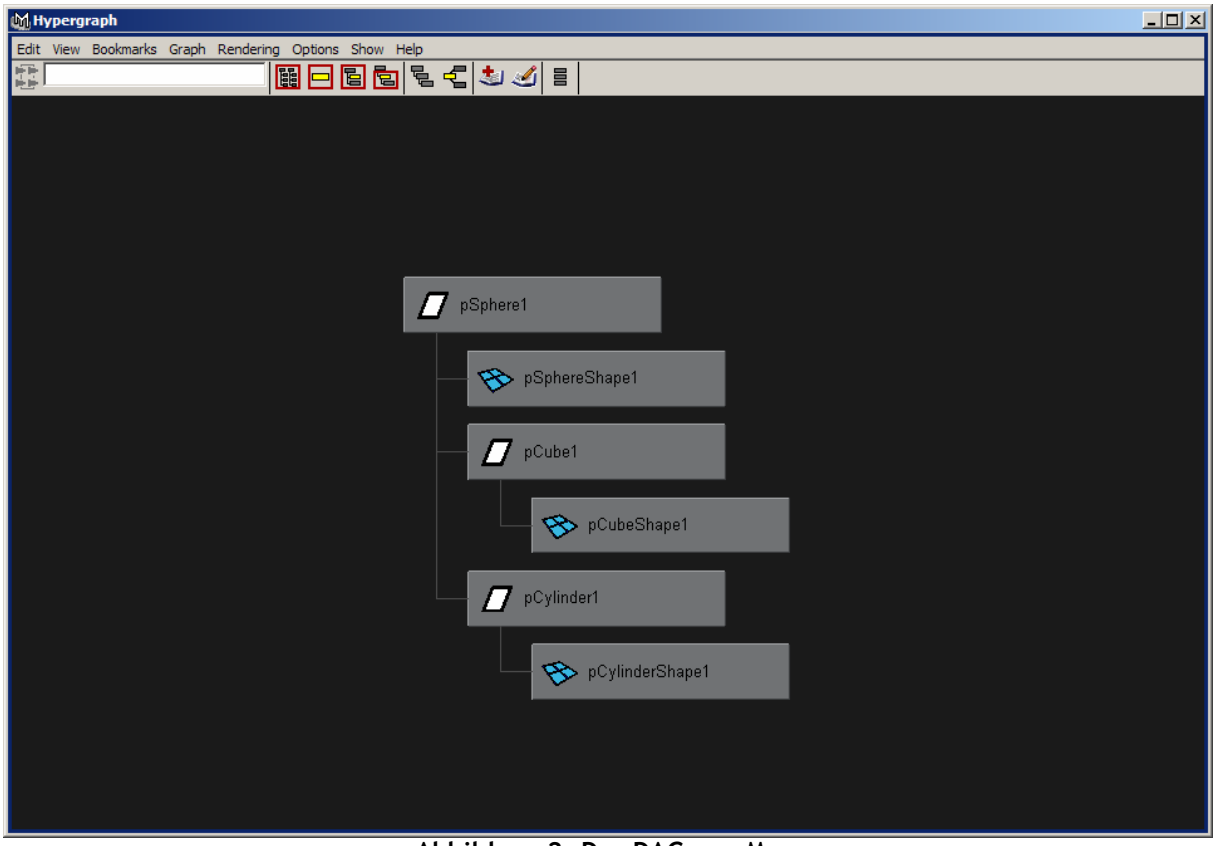


Abbildung 2: Der DAG von Maya

Abbildung 2 zeigt einen DAG, in dem ein Quader und ein Zylinder unter einer Kugel eingeordnet sind. Hierarchische Beziehungen zwischen verschiedenen Objekten finden über die entsprechende Einordnung der *Transform*-Knoten in den DAG statt.

Hin und wieder ist es notwendig, mehrere Objekte unter einem *Transform*-Knoten zusammen zu fassen, ohne ein extra Objekt erstellen zu müssen (z.B. einen *locator*). Hierfür bietet Maya *null*-Objekte, welche im DAG nur ein *Transform*-Knoten ohne *Shape*-Knoten sind. Deshalb sind sie im Betrachtungsfenster nicht sichtbar und können auch nicht gerendert werden.

Da in diesem Beispiel *pCube1* und *pCylinder1* in der Hierachie unter *pSphere1* sind, wirkt sich die Transformationsmatrix von *pSphere1* direkt auf seine Kinder aus. Dies ist mathematisch betrachtet nichts weiter als eine Matrix-Multiplikation von rechts. Um die Weltkoordinaten des Quaders zu bestimmen, muss die Transformationsmatrix der Kugel mit einbezogen werden.

Der DG hingegen ist die Anwendungsreihenfolge von Berechnungen auf *Shape*- oder *Transform*-Knoten und modelliert den Fluss von Daten. Im Gegensatz zum DAG kann dieser Graph Zyklen enthalten, welche sich aber auf die Berechnungsreihenfolge auswirken und somit unerwartete Folgen haben können.

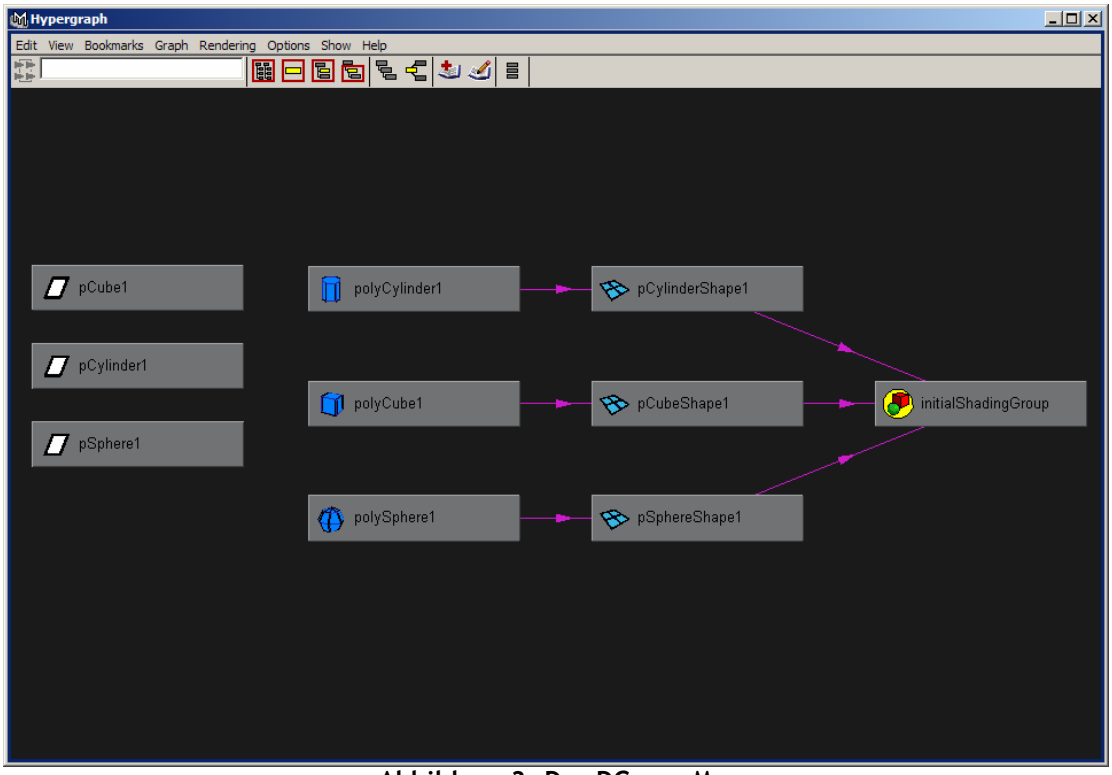


Abbildung 3: Der DG von Maya

Abbildung 3 zeigt den zu Abbildung 2 gehörenden Dependency-Graph. Die hierarchische Information spielt keine Rolle. Nun sind hier 4 zusätzliche, reine DG-Knoten zu sehen. *polyCylinder1*, *polyCube1* und *polySphere1* sind Knoten, welche Parameter wie den Radius bereitstellen und über die Datenflusskante z.B. zu *pCubeShape1* die entsprechende Geometrie erzeugen. Mehr zu Datenflusskanten weiter unten im Kapitel.

Diese Knoten werden aus Sicht des *Shape*-Knotens auch History genannt. Das sind also Knoten, welche im Sinne des Datenflusses vor dem *Shape*-Knoten liegen und zusammengenommen dessen Form beeinflussen. *initialShadingGroup* ist ein Knoten, der für das Material der Objekte zuständig ist, welche mit ihm verbunden sind. Jedes Objekt wird nach seiner Erstellung initial mit diesem Knoten verbunden. Mit jedem neuen Material, welches der Anwender erstellt, werden neue *Shading-Groups* erzeugt. Knoten, welche im Sinne des Datenflusses nach einem *Shape*-Knoten kommen, werden (in Bezug auf diesen *Shape*-Knoten) Future genannt.

Der Anwender hat die Möglichkeit, die History zu löschen und somit Knoten und Berechnungsschritte bei der Auswertung der Szene zu sparen. In diesem Fall werden die bis dahin eingestellten Parameter dazu verwendet, „feste“ Geometrie zu erzeugen und im *Shape*-Knoten zu speichern. Danach ist es nicht mehr möglich, den Radius der Kugel direkt über einen Parameter zu ändern, da dieser vorher durch den Knoten *polySphere1* bereitgestellt, aber durch das Löschen der History entfernt wurde.

Über die bereits verbunden DG-Knoten und deren Kanten hinaus ist es möglich, im Connection-Editor eigene Kanten zwischen Attributen zu ziehen. Dies bedeutet dann, dass der Wert des übergeordneten Attributs, also jenes, welches im Sinne des Datenflusses davor liegt, den Wert des untergeordneten Attributs steuert. Sofern die Datentypen der Attribute kompatibel sind, können auf diese Weise komplexe Mechanismen für interessante Effekte entwickelt werden. Als Beispiel

liegt die Entwicklung von Shadern für Maya vor, hier muss im Allgemeinen auf diese Technik zurückgegriffen werden, um z.B. Texturknoten mit Materialknoten zu verbinden. Größtenteils passiert dies in Maya automatisch, aber mit steigenden Anforderungen sind immer mehr Eingriffe „von Hand“ in den DG notwendig.

Der DAG und DG zusammen können bei großen Szenen schnell sehr hohe Komplexität erreichen, sodass ein Anwender ohne vorsichtigen Umgang schnell an die Grenzen der Ressourcen stoßen kann.

4.1.2. Vereinfachung des Szenegraphen für XL4Maya

Nun musste entschieden werden, welche Knoten XL bekannt gemacht werden, also eine Repräsentation als Instanz einer Java-Klasse erhalten. Doch warum nicht den gesamten Dependency-Graph, also jeden Knoten aus Maya XL bekannt machen? Diese Entscheidung führte zu weiteren Überlegungen, in deren Schlussfolgerung kaum Vorteile zu erkennen waren. Zum Einen hätte der Anwender genaue Kenntnisse über die Struktur des Maya-Szenegraphen haben müssen. Der Programmierer müsste jegliche Konsequenzen des Befehls „*Sphere*“ kennen, denn die Struktur des Maya-Szenegraphen verändert sich relativ stark, schon bei der Erzeugung eines einzelnen Objektes. Wenn dann noch diverse Kantenoperationen hinzukommen, ist die Übersicht nicht mehr gegeben.

Zum Anderen würde der Anwender durch die komplexe Struktur der Kanten, die von einem Knoten zum Nächsten führen, einen weitaus höheren Programmieraufwand haben. Es wäre ungemein schwierig, ein passendes Muster auf der linken Seite der Regel von Hand zu erstellen, um den gewünschten Effekt zu erzielen.

Ein weiterer Grund, und das war der Entscheidende: Was sollte der Anwender mit diesen DG-Knoten anfangen? Ersetzungsregeln wären kaum ausführbar, da es zwischen nicht-DAG-Knoten keine „Standardkante“ gibt. Eine Codezeile wie

```
[XL]
x:Transform y:Shape -DGEdge-> z:Sphere ==> x y;
```

wäre nicht durchführbar, sie würde in Maya keinen Sinn ergeben. Ebenso würde es selten notwendig sein, die Datenflusskanten manuell zu verändern, da der Aufwand für die Programmierung den Nutzen bei weitem übersteigen würde. Für solche Fälle wurde die Möglichkeit implementiert, MEL-Code innerhalb von XL aufzurufen. Damit sind solche Spezialfälle ebenso abgedeckt. Weiteres dazu befindet sich im Kapitel 5.6.

Die nicht-DAG-Knoten repräsentieren keine Objekte, sondern sind Algorithmen, welche Daten erhalten (z.B. Geometriedaten), diese verändern und weiterleiten an andere Knoten. Sie bieten keine strukturelle oder inhaltliche Information der Szene. Einzig und allein die Parameter dieser Knoten sind für den Anwender interessant und im XL-Code überblickbar. Um diese Parameter zur Verfügung zu stellen, ohne einen direkten Zugriff auf die DG-Knoten zu erlauben, wurde der Suchalgorithmus der Methode zum Setzen und Abfragen von Attributen, welche implizit bei der Anwendung des []-Operators auf mayaspezifische Klassen aufgerufen wird, um die Historyknoten erweitert. Weiteres dazu im Kapitel 5.4.

4.1.3. Knotenklassifikation

In XL ist der Anwender es gewohnt, mit Begriffen wie „Sphere“ oder „Cube“ umzugehen. Wenn man aber die Struktur einer Kugel im Szenegraph von Maya betrachtet, ist nicht sofort klar, welcher Knoten eigentlich die Kugel repräsentiert, die man am Bildschirm sehen kann. Wird in Maya eine Kugel erzeugt, etwa durch den MEL-Befehl

```
[MEL]
polySphere;
```

entstehen in Maya drei neue Knoten: Ein *Transform*-, ein *Shape*- und ein *Sphere*-Knoten. Der *Transform*- und der *Shape*-Knoten sind DAG-Knoten, *Sphere* ist nur dem DG bekannt. Die Schwierigkeit liegt nun darin, dass nur die Kombination aller drei Knoten als eine Kugel aufzufassen ist. Doch der XL-Anwender möchte vorzugsweise mit einem einzigen Knoten arbeiten. Zwar befinden sich alle notwendigen Geometrie-Daten im Shapeknoten, doch sind diese parametrisiert durch den damit verbundenen DG-Knoten, in diesem Fall ist es der *Sphere*-Knoten.

Auf Grund der Entscheidung, den vollständigen DG nicht mit in den XL-Graphen aufzunehmen, besitzt XL keine Instanz dieses *Sphere*-Knotens. Somit muss entweder der *Shape*- oder der *Transform*-Knoten die *Sphere* in XL repräsentieren. Die Entscheidung fiel zuletzt auf den *Transform*-Knoten, da der *Shape*-Knoten ohne einen zugehörigen *Transform*-Knoten nicht existieren kann und der *Transform*-Knoten zugleich sämtliche Transformationsmatrizen enthält.

Erzeugt nun XL eine neue *Sphere*, wird intern der entsprechende MEL-Befehl ausgeführt. Eine direkte Implementation über die C++-API hätte zu viel Aufwand nach sich gezogen. Über Callbacks, welche bei neu erzeugten Knoten in Maya bestimmten Code ausführen, können diese neuen Knoten im Maya-Graph identifiziert werden. Damit XL der gesamte DAG bekannt ist, wird der neue *Transform*- und *Shape*-Knoten angemeldet, also entsprechende Instanzen der Java-Klassen *Transform* und *Shape* erzeugt.

Für den speziellen Fall der Sphere, aber auch für andere Primitivobjekte existieren für XL verschiedene Klassen, welche von der *Transform*-Klasse ableiten. Somit wird bei der Anmeldung dieser neuen *Sphere* in Java der Konstruktor der *Sphere*-Klasse aufgerufen, die Referenz zum entsprechenden realen Maya-Objekt gespeichert und die Instanz in die interne Verwaltung von XL eingetragen. Damit ist bei allen weiteren Vorgängen von XL, z.B. Suchmustern oder Ersetzungen, dieser reale Knoten aus Maya vom Typ *Transform* als *Sphere* in XL bekannt.

Ein besonderer Fall tritt ein, sollte der Anwender XL-Code auf eine bereits existierende Maya-Szene anwenden. Zu diesem Zeitpunkt sind der Knoten-Verwaltung von XL noch keinerlei Knoten bekannt. Beim Traversieren des Maya-DAGs trifft der Iterator also nur auf *Transform*- bzw *Shape*-Knoten. Da die DG-Knoten unsichtbar sind, musste ein Prinzip implementiert werden, welches primitive Knotentypen auf Maya-Seite bei initialer Traversierung der Szene erkennen kann.

Maya bietet zum Traversieren des DAGs bzw DGs zwei Klassen: *MItDag* und *MItDependencyGraph*, welche alle notwendigen Methoden bereit stellen[1]. Zum Traversieren des Maya-DAGs wird der Dag-Iterator verwendet. Trifft dieser auf

[1] Vgl.: [15].

einen *Transform*-Knoten, dessen Referenz XL noch nicht bekannt ist, muss geprüft werden, ob es sich eventuell um ein Primitiv-Objekt handelt. Dafür wird zunächst der passende *Shape*-Knoten ausfindig gemacht, welcher immer in der Hierarchie direkt unter dem *Transform*-Knoten eingeordnet ist. Sollte einer existieren (z.B. bei *null*-Objekten ist dies nicht zwingend der Fall), muss der DG-Iterator die History ausgehend von diesem Knoten prüfen, also entgegen des Datenflusses die DG-Knoten traversieren. Befindet sich irgendwo innerhalb der History ein Knoten mit dem Typ *kPolySphere*, ist der *Transform*-Knoten in XL als *Sphere* anzumelden, sonst wird er als *Transform*-Knoten in XL angemeldet.

Diese Prinzip ist recht einfach, kann allerdings einige Probleme aufwerfen. So ist es nicht immer gegeben, dass der DG so sauber strukturiert ist, es kann z.B. ein DG-Knoten mit mehreren *Transform*-Knoten verbunden sein und Zyklen können existieren. In diesem Fall würden alle verbundenen *Transform*-Knoten als Sphere angemeldet werden, was nicht immer richtig wäre. Dies ist jedoch recht selten so, und ein Abfangen jeglicher Ungewissenheiten hätte ungemein viel Zeit in Anspruch genommen.

Zusammengefasst: XL4Maya kann bei einer bereits bestehenden Szene bestimmte Primitiv-Objekte von selbst erkennen, sodass Suchen darauf ausgeführt werden können und passende Knoten automatisch als Primitiv-Objekte in XL angemeldet werden. Wird ein Primitiv-Objekt in XL erstellt, wird dieses sofort angemeldet und ist somit von vornherein als ein solches bekannt, obwohl der entsprechende Maya-Knoten eigentlich ein *Transform*-Knoten ist. Sobald der Anwender seine Szene nach der Anwendung von XL4Maya speichert und dabei XL-Knoten erzeugt hat, sind diese nach dem Neuladen und Reinitialisieren des Compilers nicht mehr als solche Knoten erkennbar (z.B. F, M usw.).

4.1.4. Das Pointer-Problem

"Maya owns all the data, you own none of it!"[1]

Jeder Knoten in XL, also jede Instanz der Klasse *Node* (und deren abgeleiteten Klassen) wird durch eine eindeutige Referenz adressiert. Dies sind einfache Integer-Werte. Um jedem XL-Knoten einen Maya-Knoten zuordnen zu können, muss diese Referenz auf beiden Seiten des Plug-ins gleich sein. Ein Beispiel:

```
[XL]
x:Node ==> x Sphere;
```

Diese XL-Zeile sucht alle Knoten im XL-Graph, erzeugt für jeden eine Kugel und hängt diese in der Hierarchie unter den gefundenen Knoten. Nun müssen Maya und XL gleichermaßen mit dem Begriff „Sphere" umgehen können, das heißt eine eindeutige Zuordnung der Sphere auf Maya-Seite zu einer Instanz der Klasse *Sphere* auf XL-Seite muss hergestellt werden. Die ursprüngliche Idee war, den Pointer, also die reale Speicheradresse auf den entsprechenden Knoten in Maya als Integer auszulesen und an XL zu übergeben. Somit hätte XL bei der Arbeit mit diesem Knoten nur den gespeicherten Pointer übergeben müssen und Maya hätte diesen als

[1] [4], S. 285.

ein Objekt (also einen Knoten) im Speicher identifizieren können. Doch leider verbietet Maya solch einen Umgang mit Objekten im Speicher.

In der Maya-API gibt es die Klasse *MObject*, diese kapselt so genannte Function-Sets. Ein Function-Set ist eine C++ Klasse, welche direkt an dem (im Speicher vorhandenen) Objekt in Maya arbeitet. Somit hat der API-Programmierer nur indirekten Zugriff auf die Maya-Objekte, aber ansonsten alle Freiheit, die für seine Arbeit notwendig ist. Das heißt, er kann sie an Funktionen übergeben und ineinander umwandeln, sofern die Function-Sets kompatibel sind. Allerdings gilt die Einschränkung, dass der Pointer, der zu dem realen Objekt im Speicher führt, nicht dem Programmierer zur Verfügung gestellt wird. Den Pointer auf das *MObject* kann man zwar auslesen, aber dieser ändert sich im Laufe der Arbeit mit Maya ständig und nicht nachvollziehbar.

Somit galt es das Problem zu lösen, dass XL eine statische Referenz zu Objekten des Maya-Szene-Graphs benötigt. Pointer können nicht ausgelesen werden, also musste ein anderes Prinzip entwickelt werden.

Zur Lösung des Problems musste eine Liste aller Objekte, die XL zur Verfügung gestellt werden sollten, erstellt und konsistent gehalten werden. Dies ergab einige neue Probleme, die es zu lösen galt. In der Maya-API wird darauf hingewiesen, dass man *MObject*-Instanzen nie länger als notwendig gespeichert halten sollte, da die Gültigkeit dieser Objekte offensichtlich nicht über längere Zeit aufrecht erhalten werden kann. Doch zunächst galt es herauszufinden, ob diese Einschränkung auch die Verwendung zur Lösung des Pointer-Problems betraf.

Die Maya-API bietet eine vorimplementierte Liste, die Klasse *MObjectArray*. In dieser kann man *MObjects* speichern und verwalten. Um die Verwaltung dieser Liste in Bezug auf XL zu vereinfachen, wurde die C++-Klasse *ObjectList* implementiert. In dieser Klasse befinden sich alle notwendigen Methoden zum Einfügen und Entfernen von Knoten in die Liste mit Zugriffen auf die Indizes, um den Pointer zu einem bestimmten Maya-Objekt zu bestimmen. Dabei wird ein Index innerhalb der Liste zur Neubelegung freigegeben, wenn ein Objekt in Maya aus dem DAG gelöscht wird.

Möchte nun XL einen bestimmten Knoten ansprechen, wird die gespeicherte Referenz der entsprechenden Java-Klasse an die C++-Methode *getObject* der Instanz der Klasse *ObjectList* übergeben. Diese Methode liefert ein *MObject* zurück, also einen Verweis auf ein reales Mayaobjekt.

Umgekehrt wiederum muss C++ Objekte XL bekannt machen, also deren Referenz anhand eines übergebenen *MObjects* bestimmen und übergeben. Hierfür existiert die Methode *getIndex*, welche den entsprechenden Wert zurückliefert.

Von all diesen Vorgängen bekommt der Anwender nichts mit, außer dass sich diese indirekte Adressierung auf die Geschwindigkeit auswirkt. Bei jeder Anfrage von XL muss diese Liste vollständig durchlaufen werden, um das Objekt anhand der Referenz ansprechen zu können. In der Auswertung des Profilings liegen diese beiden Methoden in Bezug auf Aufrufhäufigkeit an der Spitze.

Bei der Implementierung dauerte es einige Zeit, bis dieses Prinzip fehlerfrei funktionierte. Es gibt viele Einflüsse auf den Maya-DAG, die man auch auf diese Liste von *MObjects* übertragen muss. Sobald XL Knoten löscht oder hinzufügt, ob direkt oder indirekt, muss dies eins zu eins auf die Liste angewendet werden, um die Konsistenz zu bewahren. Erst recht spät konnte dieses Ziel erreicht werden. Allerdings hatte die Verwendung des *MObjectArrays* auch seinen Preis. Solange der Anwender während der Benutzung von XL (d.h. zwischen zwei Ausführungen von XL-Methoden) die Szene nicht verändert, schlägt die Einschränkung, dass *MObjects*

nicht über einen längeren Zeitraum gespeichert werden sollten, nicht zu. Sollten aber dennoch Änderungen vorgenommen werden, kommt es beim erneuten Ausführen einer Methode von XL zum Absturz. Ausgenommen davon sind Kameradrehungen, aber Aktionen wie Objektverschiebungen oder Parameteränderungen können zu diesem Fehler führen.

4.2. MEL

Maya unterscheidet sich von anderen 3D-Programmen durch eine weitere Besonderheit. Die Entwickler begannen bei Maya zunächst, eine Programmiersprache zu entwickeln, welche allen Anforderungen an die Animations- und Grafikwelt genügen würde. Erst danach wurde die GUI entworfen und ausschließlich mit den Mitteln von MEL umgesetzt. Somit ist MEL (Maya Embedded Language) fähig, fast alles in und um Maya zu steuern. Jede Funktion, die der Anwender aufruft oder anklickt, ist im Prinzip ein MEL-Skript. Durch Anpassung dieser Skripte ist Maya komplett individualisierbar und kann an fast jedes Problem angepasst werden.

Entwickelt man nun ein Plug-in, könnte man dies fast ausschließlich in MEL statt in der C++-API tun. Bis auf wenige Funktionen, z.B. eigene Knotentypen, stehen viele Methoden zur Verfügung, die man für den Umgang mit Maya braucht. Einige Dinge, z.B. die GUI, kann man nur über MEL ansteuern. Damit kommt ein Programmierer, der mehr als nur ein Kommandozeilen-Plug-in schreiben möchte, nicht um MEL herum.

Es gibt aber einen wichtigen Grund dafür, bestimmte Dinge in C++ zu entwerfen: Die Geschwindigkeit. MEL wird von Maya interpretiert und ist somit recht langsam. Ebenso hat man nur wenige Debugmöglichkeiten, welche bei komplexeren Aufgaben schnell die Grenzen des Programmierers ausreizen. Es liegt also beim Programmierer, eine ausgewogene Mischung aus MEL und C++ zu finden, denn selbst wenn der Entwickler in C++ einen Algorithmus geschrieben hat und diesen zur Anwendung bringen möchte, muss er diesen als MEL-Befehl oder als Knoten implementieren, damit der Anwender Zugriff hat. Auch ein selbst entwickelter Knoten benötigt ein MEL-Kommando, um überhaupt erstellt werden zu können.

4.2.1. Das MEL-Kommando XL4Maya

XL4Maya implementiert das MEL-Kommando

```
[MEL]
XL4Maya
```

mit den Flags

```
-i/-init [String]
-r/-runMethod [Integer]
-s/-stepMethod [Integer]
-d/-delay [Integer]
```

und dem Rückgabetyp

```
string[]
```

Zunächst muss, um XL zu initialisieren, der Befehl

```
[MEL]
XL4Maya –i „Pfad“;
```

ausgeführt werden. An das Plug-in wird nun die XL-Datei weitergegeben und der Compiler in Java initialisiert. Wurde die XL-Datei nicht gefunden (z.B. durch Schreibfehler) oder der Compiler meldet einen Fehler, wird der Prozess abgebrochen und eine Fehlermeldung ausgegeben. Wurde die Datei fehlerfrei kompiliert, gibt das Plug-in an MEL eine Liste aller gefundenen, als *public* deklarierten Methodennamen zurück. Nun muss dem Compiler mitgeteilt werden, welche Methode ausgeführt werden soll:

```
[MEL]
XL4Maya –s x;
```

X steht hier für den 1-basierten Index der Methode, im Code ist dies die xte *public*-Methode. Das Kommando führt die Methode einmal aus. Alternativ dazu führt folgender Code die Methode solange aus, bis der Anwender auf „Esc“ drückt:

```
[MEL]
XL4Maya –r x;
```

Zusätzlich zu den hier beschriebenen Parametern kann der Anwender noch das *delay* verändern:

```
[MEL]
XL4Maya –r x –d y;
```

X ist wieder der Index der Methode, y das gewünschte *delay* zwischen den Ausführungsschritten. Dies kann sinnvoll sein, wenn dynamische Prozesse zu schnell ablaufen, sodass der Anwender sie nicht verfolgen kann. Wird kein Parameter angegeben, ist der Standard 800 ms.

Dies ist die gesamte Steuerung des Plug-ins, mehr ist nicht notwendig. Wenn eine andere Datei geladen oder der Compiler erneut auf veränderten Code angewendet werden soll, muss einfach erneut initialisiert werden. Um die Steuerung zu vereinfachen, wurde auch eine GUI implementiert, die im Kapitel 6.2 erklärt wird.

4.2.2. Implementierung eines MEL-Kommandos in C++

Die Implementierung eines MEL-Kommandos für Maya 7 beginnt mit einem Wizard des Maya-SDK für Visual Studio .NET 2003. Dieser erstellt zwei Klassen, die PluginMain.cpp und XL4MayaCmd.cpp. Die PluginMain.cpp realisiert zwei Methoden:

```
[C++]
MStatus initializePlug-in( MObject obj );
MStatus uninitializePlug-in( MObject obj );
```

In ihnen wird Code untergebracht, welcher beim Initialisieren und Deinitialisieren des Plug-ins ausgeführt werden soll. Die Initialisierung findet statt, wenn sich zum einen das kompilierte Plug-in im Verzeichnis /Maya/bin/Plugins/ befindet und zum anderen in den Voreinstellungen das Plug-in geladen wird. Deinitialisiert wird das Plug-in, wenn man es in den Voreinstellungen beendet oder Maya beendet wird.

In der Initialisierung wird das Kommando XL4Maya beim Plug-in angemeldet, sodass dieses als MEL-Kommando zur Verfügung steht, sobald das Plug-in geladen ist. Folgende Zeile implementiert dieses:

```
[C++]
status = Plugin.registerCommand ("XL4Maya", XL4MayaCmd::creator,
XL4MayaCmd::m_Syntax);
```

Es ist möglich, weitere Kommandos anzumelden, um dem Plug-in weitere Funktionalität zu geben.

Zusätzlich wird in der PluginMain.cpp die Java-Virtual-Machine erzeugt, also eine Instanz der Klasse JVM.

Die Klasse XL4MayaCmd implementiert das eigentliche Kommando und erbt von der Maya-API-Klasse *MPxCommand*. Einstiegspunkt beim Aufruf des Kommandos aus der Kommandozeile oder über das Skript (beschrieben im Kapitel 4.2.3) ist die Methode

```
[C++]
MStatus XL4MayaCmd::doIt( const MArgList &args )
{ … }
```

An sie wird die Liste der Parameter übergeben. Diese können ausgewertet und für Berechnungen verwendet werden. Der folgende Code ist ein Ausschnitt der Klasse.

```
[C++]
MStatus XL4MayaCmd::doIt( const MArgList &args )
{
bool init = false;

…

      MSyntax syntax = this->m_Syntax();
      MArgDatabase argData(syntax, args);

      if( argData.isFlagSet(XL4MayaInit))
```

```
            argData.getFlagArgument( XL4MayaInit, 0, init );
      …
}

MSyntax XL4MayaCmd::m_Syntax()
{
      MSyntax syntax;
      syntax.addFlag(XL4MayaInit,        XL4MayaInitLong,
      MSyntax::kBoolean);
      …

      return syntax;
}
```

Dies ist ein Beispiel, wie ein Parameter angemeldet und ausgewertet wird. Die Methode *XL4MayaCmd::m_Syntax()* ist für die Anmeldung zuständig. Es wird ein MSyntax-Objekt erzeugt und zurück gegeben. In der *XL4Maya::doIt*, die bei der Ausführung des Kommandos aufgerufen wird, werden die übergebenen Parameter durch das Objekt *argData* ausgewertet und an interne Variablen zur weiteren Verwendung übergeben.

4.2.3. Das Skript zum Kommando

Damit der Anwender nicht allein mit dem Kommando arbeiten muss, wurde via MEL eine GUI implementiert, die mit dem Plug-in interagiert. Prinzipiell kann der Anwender XL einzig und allein über die Konsole steuern, doch es ist deutlich komfortabler mit einer entsprechenden GUI. Eine genauere Erklärung der einzelnen Funktionen der GUI befindet sich im Kapitel 6.2, hier wird hauptsächlich auf die programmiertechnischen Hürden eingegangen.

MEL bietet eine Vielzahl von GUI-Elementen, wie Menüs, Fenster und Steuerelemente. In Maya erstellt man ein Fenster mit einem einfachen Befehl:

```
[MEL]
window Fenstername;
```

Nun kann ein Layout hinzugefügt werden. Ein Layout definiert die Art und Weise, wie Steuerelemente angeordnet werden. Ein Layout kann mehrere andere Layouts beinhalten:

```
[MEL]
scollLayout;
columnLayout;
```

Hierbei wird in das Fenster, welches der aktuelle *Parent* ist, ein *scrollLayout* eingefügt. Dieses stellt eine *scrollBar* zur Verfügung, um mehr Elemente, als auf den Bildschirm passen, zu ermöglichen. Darin wird ein *columnLayout* erstellt, welches sämtliche Unterelemente in einer einzigen Spalte darstellt.

```
[MEL]
button –label „exit“ –command „exit();“;
```

Nun wird in das *columnLayout* ein Button eingefügt, welcher bei einem Klick des Anwenders die Methode *exit()* ausführt. Natürlich muss diese Methode in einem Skript, das vorher mit dem Befehl

```
[MEL]
source Skriptname;
```

hinzugefügt wurde, vorhanden sein, um ausgeführt werden zu können. Um das Fenster nun anzuzeigen, fehlt noch eine Zeile:

```
[MEL]
showWindow Fenstername;
```

Dies sind die Grundlagen für eine einfache GUI in MEL. Es gibt noch eine Vielzahl von weiteren Elementen, welche verwendet werden können.

Im ersten Entwurf für die GUI von XL4Maya wurde ebenfalls solch eine Fenstertechnik eingesetzt. Doch im Laufe der Beispielentwicklung fiel auf, dass jedes zusätzliche Fenster enorm viel Platz wegnahm und störend auf die Entwicklung von XL-Programmen wirkt. Der Workflow

Maya-Szene erstellen -> Code für XL erstellen -> Code anwenden -> Code editieren -> Code anwenden -> …

wurde ständig durch zusätzliche Klicks und die Notwendigkeit, das Plug-in-Fenster aus dem Weg zu schieben, unterbrochen. Eine andere Lösung musste entwickelt werden, welche wenig Platz einnimmt und dennoch komfortabel ist.

Maya bietet die Möglichkeit, Buttons und Informationen direkt in der 3D-Ansicht (dem HUD) einzublenden. Hierfür ist der sichtbare Bereich in Sektionen und diese ihrerseits in Blöcke unterteilt. Genauere Informationen dazu befinden sich in der Maya-Dokumentation. Der Vorteil dieser Vorgehensweise ist, dass die Buttons immer sichtbar sind und nur wenig Platz einnehmen. Für den Editor besteht noch immer die Möglichkeit, nach Klick auf den Button *edit* ein weiteres Fenster zu öffnen und einen Editor anzubieten. Dieser wurde in der Datei *xlEditor.mel* implementiert.

Hierbei sei erwähnt, dass der Editor seinen Zweck erfüllt, allerdings wenig komfortable Möglichkeiten bietet. Dazu sind die MEL-Strukturen recht kompliziert und bieten bei weitem nicht alle Möglichkeiten z.B. der Windows-API. Dazu gehören Code-Highlighting, Anzeige der Zeilennummern, Drucken usw. Die Empfehlung des Autors liegt in diesem Fall darin, einen externen Editor zu verwenden, wie z.B. TextPad (www.textpad.com). Dieser liefert sämtlichen Komfort eines Code-Editors mit und ist zudem kostenlos. Der Anwender kann in diesem Fall bequem die XL-Datei erstellen, speichert diese und wechselt hinüber zu Maya, welches im Hintergrund weiterlaufen kann, während man den Editor benutzt. Anschließend muss nur dem Plug-in die Datei via *load* mitgeteilt und der Compiler mit *start* angewendet werden. Sollten dann Korrekturen notwendig werden, kann der Anwender zurück zum Editor wechseln, dort die notwendigen Veränderungen machen und nach dem Speichern zurück in Maya *Reset* drücken und erneut den Compiler anwenden.

Eine Besonderheit ist diese Reset-Funktion. Da es kaum möglich ist, sämtliche Veränderungen von XL an der Maya-Szene mitzuführen, um sie gegebenenfalls rückgängig zu machen, musste für die Verwendung von *Reset* eine Konvention eingeführt werden. Bevor der Anwender XL erstmalig auf eine Maya-Szene anwendet, sollte diese gespeichert worden sein, damit der ursprüngliche Zustand erhalten bleibt. Klickt der Anwender auf *Reset*, fragt MEL den Pfad und den Namen der aktuell geladenen Szene ab (sollte bis dahin nicht gespeichert worden sein, meldet *Reset* einen Fehler) und lädt diese einfach neu. Es obliegt hier dem Anwender, an der richtigen Stelle zu speichern.

Neben der GUI wurde auch der Zugriff auf die INI-Datei komplett in MEL programmiert. Hierfür wurde das Skript *xlINI.mel* angelegt, welches verschiedene Methoden der GUI zur Verfügung stellt. Sobald der Anwender eine neue Datei anlegt oder lädt, wird der Dateiname samt Pfad in der INI-Datei gespeichert. Ebenso wird das zuletzt gewählte *delay* mit unter-gebracht.

Der Anwender hat, um die Skripte anzuwenden, nur zwei Zeilen Code auszuführen:

```
[MEL]
source xl.mel;
runxl();
```

Hierbei wird das Skript *xl.mel* in den Speicher von Maya geladen und zugleich überprüft. Anschließend erscheint mit der Ausführung der Prozedur *runxl()* die GUI. Alle weiteren MEL-Prozeduren werden über die GUI angesteuert.[1]

4.3. Die Verwaltungsklassen in Java

Im Folgenden sollen nicht die einzelnen Details der Verwaltungsklassen aufgeführt, sondern nur ein Überblick geboten werden. Das gesamte Projekt unterteilt sich ein drei Packages: *mayaL*, *model* und *objects*. Die Klassen zur Kommunikation zwischen dem XL-Compiler und dem Plug-in in C++ befinden sich im Package *model*, während die verschiedenen Knoten in den anderen beiden untergebracht sind.

Die erste vom Maya-Plug-in angesteuerte Klasse ist *ControlFlow*. Die darin enthaltene *main*-Methode wird ausgeführt, sobald der Anwender in der GUI auf *Start* klickt oder den entsprechenden MEL-Befehl

```
[MEL]
XL4Maya –i FILENAME
```

aufruft. In ihr werden zunächst sämtliche Initialisierungen vorgenommen, damit bei der späteren Ausführung keine Konflikte entstehen. Anschließend wird die XL-Datei eingelesen und an den Compiler weitergegeben. Nach dem erfolgreichen Kompilieren werden an C++ und somit an MEL die Namen aller als *public* deklarierten Methoden zurückgegeben.

Bei der Programmierung von XL-Dateien liegt aufgrund bestimmter Umstände eine zusätzliche Konvention vor: Die in XL definierte Hauptklasse sollte den Dateinamen tragen, damit der XL-Compiler automatisch weiß, welche die Primärklasse ist. Ansonsten kann es passieren, dass der Compiler die auszuführenden Methoden in der falschen Klasse sucht, sollten noch andere Klassen in der XL-Datei definiert sein

1 Weitere Informationen über die Funktionen von MEL befindet sich in [16].

und diese sich vor der Hauptklasse befinden. Dies ist für den erfahrenen Java-Programmierer kaum ein Umstand, da die Java-Code-Conventions ähnliches vorsehen.

Innerhalb der Klasse *ControlFlow* befindet sich auch die Methode *step_method(int index)* zur Ausführung einer XL-Methode.

Neben der *ControlFlow*-Klasse sind im Package *model* auch die vom Compiler benötigten Klassen *Compiletime* und *Graph*, welche z.B. für die Kantenoperationen wie *addEdgeBits* zuständig sind. In der Klasse *Graph* ist der *EdgeIterator* implementiert, welcher XL die Möglichkeit gibt, sich durch den Maya-Graphen zu bewegen und bestimmte Knoten zu suchen. Die Klassen *MEL* und *User* sind für den Anwender interessant und werden in den Kapiteln 5.6 bzw. 5.9 genauer erklärt.

4.3.1. Die Kommunikation zwischen Java und C++

Die eigentliche Schnittstelle zu Maya ist aber die Java-Klasse *MayaAPI*. Hier befinden sich alle Methoden, welche C++ zur Ausführung aus Java heraus zur Verfügung stellt. Java macht keinen Unterschied daraus, dass beim Aufruf einer dieser Methoden eigentlich C++-Code aufgerufen wird. Jede Aktion des XL-Compilers, welche Auswirkungen auf Maya haben soll, findet über diese Klasse statt. Hier befinden sich alle Methoden zur Erzeugung von Objekten, Veränderung des Maya-Graphen und Umsetzung von besonderen Funktionen. Im Laufe der Programmierarbeiten haben sich ganze 83 native Methoden angesammelt. Dies war kaum zu verhindern, da C++-seitig kein echtes Überladen der einzelnen Methoden möglich war. Damit hätte sich etwa die Hälfte der Methoden einsparen lassen.

Nun ein paar Erklärungen zur C++-Seite. Das Gegenstück zur *MayaAPI*-Klasse in Java ist die Klasse *JVM* in C++. Hier werden alle Methoden, welche Java ausführen kann, bekannt gemacht. Hierbei wird ein Array vom Typ *JNINativeMethod* angelegt und mit folgenden Informationen, hier beispielhaft nur einer Methode gefüllt:

```
[C++]
nm[0].name = "m_println";
nm[0].signature = "(Ljava/lang/String;)V";
nm[0].fnPtr = m_println;
```

In der ersten Zeile wird der Name vergeben, welcher später in Java bekannt werden soll. Wichtiger ist die zweite Zeile, hier wird die Signatur der Methode gespeichert, also der Typ der Rückgabe und die Typen der Übergabeparameter. Nur über die korrekte Signatur kann nach dem Starten der Java-Virtual-Machine aus Java heraus die Methode auch gefunden werden. Nur allzu oft hielten Programmierarbeiten sich daran auf, den Fehler in einer der Signaturen zu finden. C++ kann nämlich nur feststellen, dass irgendetwas nicht stimmt und somit die JVM nicht gestartet werden kann, doch den Fehler zu finden obliegt der Erfahrung des Programmierers.

Anschließend müssen die Methoden in Java registriert werden mit den Zeilen:

```
[C++]
jclass cls = pJEnv->FindClass("de/grogra/ext/maya/model/MayaAPI");
int state = pJEnv->RegisterNatives(cls,nm,count_of_methods);
```

```
if (state != 0) {MGlobal::displayError("Can´t find MayaAPI");};
```

Die dritte Zeile spiegelt den eben beschriebenen Fehler wieder. Dies ist zugleich ein Aufruf einer Java-Methode aus C++ heraus. Jenes geschieht ebenso beim Starten des XL-Compilers und dem Ausführen verschiedener XL-Methoden, beides ist in der C++-Klasse *XL4MayaCmd*, die das MEL-Kommano implementiert, untergebracht.

Sehr viel mehr findet in der Klasse JVM nicht statt, außer dass neben dem Setzen von Parametern die JVM mit eben diesen gestartet wird. Eine große Hilfe bei der Entwicklung war der Parameter

```
[C++]
options[6].optionString = strdup(
"-Xrunjdwp:transport=dt_socket,address=8000,server=y,suspend=n");
```

welcher das komfortable Debuggen des Java-Anteils aus der Java-Entwicklungsumgebung Eclipse heraus erlaubte. Weiterhin werden in den Parametern auch die Speichereinstellungen der JVM unternommen, welche bei der Ausführung des Compilers in der JVM auf unterschiedlichen Rechner-Systemen hilfreich sein können. In dieser Implementierung werden bis auf wenige Parameter die Standards belassen.

Die eigentliche Implementierung der nativen Methoden befindet sich in der Datei *methodencontainer.cpp* bzw *methodencontainer.h*. Hier findet der Zugriff auf die API von Maya statt und führt die Aktionen, die vom XL-Compiler angefordert werden, aus. In dieser Datei befindet sich keine Klasse, sondern nur eine Sammlung all dieser Methoden und deren Implementierung.

Die Java-Virtual-Machine, gestartet von C++ aus, stellt eine Hürde, die nicht überwunden werden kann: Im Laufe der Arbeit mit XL4Maya passierte es hin und wieder, dass die Java-Virtual-Machine abstürzt. Dies passiert bei besonderen, oftmals nicht nachvollziehbaren Fehlern, aber dennoch recht selten. Das stellt eigentlich kein Problem dar, denn die JVM sollte mit dem Neustart des Plug-ins reinitialisiert werden und der Anwender ist nicht gezwungen, Maya neu zu starten. Doch ein Bug in J2SDK sorgt dafür, dass die JVM, solange der Vaterprozess läuft, also in diesem Falle Maya, sich nicht neu starten lässt. Dies ist ein von Sun anerkannter (und scheinbar akzeptierter) Bug und ist mit einfachen Mitteln nicht zu beheben. Sollte es also geschehen, dass die JVM abstürzt, reicht es nicht, das Plug-in aus Maya zu entfernen und wieder hinzuzufügen, um ein Neustarten der JVM zu erzwingen. Maya muss vollständig neu gestartet werden.

5. XL4Maya – Knoten, Klassen und Methoden

Im Folgenden wird eine geeignete Auswahl zur Verfügung stehender Klassen, Methoden und Konzepte vorgestellt und beschrieben, um welche XL zur Verwendung mit Maya erweitert wurde und die dem Anwender zur Verfügung stehen. Im Gegensatz zu Kapitel 4 ist hier nicht die eigentliche Implementierung Thema, sondern die richtige Anwendung und Funktionalität der vorhandenen Klassen und Methoden. Kern ist die Java-Klasse *User*, die einige nicht-knotenspezifische Methoden enthält, die Klasse MEL, welche es erlaubt, MEL-

Befehle aus XL heraus auszuführen, und die verschiedenen Knotenklassen, wie *F* oder *Transform*. Die beiden Klassen *User* und *MEL* werden in XL4Maya automatisch eingebunden und müssen vom Anwender nicht explizit angegeben werden. Die darin befindlichen Methoden können in XL direkt aufgerufen werden.

5.1. L-System-Befehle

XL enthält in seiner Anpassung für GroImp eine große Anzahl von L-System-Befehlen. Im Rahmen dieser Arbeit musste eine sinnvolle Auswahl verschiedener L-System-Befehle getroffen werden. Kriterium war zum Einen der Zeitrahmen der Implementierung und zum Anderen die Gewährleistung einer gewissen Funktionalität. Die Entscheidung fiel auf F, M, D, DMul, RL, RU, RH und RV. Des Weiteren wurden diese Klassen über ihre übliche Funktion hinaus mit verschiedenen zusätzlichen Möglichkeiten ausgestattet, um besondere Effekte zu erzielen.

Bei der Beschreibung jedes Knotentyps wird auf die verschiedenen Konstruktoren und Methoden eingegangen. Eine vollständige Referenz befindet sich in der beiliegenden HTML-Referenz.

5.1.1. F

Der Knoten F setzt unter den Turtle-Befehlen Bewegung in Wuchsrichtung mit gleichzeitigem Zeichnen um. Angewendet auf Maya sind diese Knoten Zylinder. Da die Anzahl dieser Knoten recht hoch sein kann, sind bei der Erzeugung die Polygonzahl auf 5 beschränkt (drei entlang der Achse, eines jeweils auf der Ober- und Unterseite). Somit senken auch tausende dieser Objekte die Performance der interaktiven Darstellung aufgrund der geringeren Polygonzahl kaum. Sollte der Anwender dennoch mehr Polygone benötigen (z.B. bei Nahansichten), kann das entsprechende Attribut auch nachträglich über die Methode der Klasse *User* *setValue* geändert werden.

In der Definition der L-System-Befehle ist F nur mit maximal einem Parameter ausgestattet, welcher die Länge bestimmt:

- `F(double length)`

Hierbei ist der Radius mit 0.1 ein Standardwert. Zusätzlich gibt es noch den Konstruktor

- `F()`

der ein F mit der Standardlänge von 2 und dem Standardradius von 0.1 erstellt. Wenn F in Kombination mit D oder DMul-Knoten eingesetzt wird, sollte einer der beiden vorangegangenen Konstruktoren eingesetzt werden. Wird nämlich an F ein Radius übergeben, hat D und DMul keinen Einfluss auf den Radius dieses Knotens.

Es wurden noch weitere Konstruktoren hinzugefügt, um bei der Erzeugung noch mehr Parameter zu setzen. Dazu gehört der Radius

- `F(double length, double radius)`

und die Polygonanzahl der Unterteilungen durch

- `F(double length, double radius, double subdivisionsAxis)`

- `F(double length, double radius, double subdivisionsAxis, double subdivisionsHeight)`
- `F(double length, double radius, double subdivisionsAxis, double subdivisionsHeight, double subdivisionsCabs)`

Dies erspart nämlich den übermäßigen Einsatz von D-Knoten und nachträglichen *setValue*-Aufrufen.

SubdivisionsAxis, subdivisionsHeight und subdivisionsCabs sind Attribute des Maya-Objektes und beschreiben die Anzahl der verwendeten Polygone um die Höhenachse, entlang der Höhenachse und der Verschlüsse auf der Ober- bzw. Unterseite.

F ist darüber hinaus mit einigen Zugriffsmethoden ausgestattet, um nachträgliche Änderungen durchführen zu können. Hierbei macht sich dieser Knoten das Prinzip der zwei Pivots zu Nutze, um den untersten und den ober-sten Punkt zu markieren.

In Maya hat jedes Objekt, repräsentiert durch seinen *Transform*-Knoten zwei verschiedene Pivots. Diese können unabhängig voneinander platziert werden und bestimmen, wie der Name schon verrät, den Angriffspunkt der Skalierung und der Rotation.

Der Scale-Pivot, also der Angriffspunkt der Skalierung markiert bei F den untersten Punkt. Wenn der XL-Comiler Kantenoperationen durchführt und z.B. zwei F in der Hierarchie aneinander hängt, wird das *child*-F mit seinem Scale-Pivot an den Rotate-Pivot des *parent*-F bewegt. Der Rotate-Pivot ist der Angriffspunkt der Rotation des Objektes.

Die Methode

- `void setlength(double value)`

führt alle notwendigen Operationen aus, um alle nachfolgenden Kinder nach der Änderung der Länge wieder an den Rotate-Pivot zu bewegen. Somit kann bei langen Ketten von Fs mitten drin ein F seine Länge ändern, ohne dass dabei die Struktur verloren geht. Keine zusätzlichen Berechnungen muss

- `void setradius(double value)`

durchführen, da der Radius sich nicht auf die Struktur auswirkt. Zur Abfrage der Werte stehen

- `double getradius()`
- `double getlength()`

zur Verfügung.

5.1.2. M

Die Klasse M verhält sich in L-Systemen genauso wie F, nur dass dabei kein Zeichnen stattfindet. In Maya ist dies ein simpler *Transform*-Knoten. Auch hier ist üblicherweise nur maximal ein Parameter vorgesehen, nämlich die Länge der Bewegung in Wuchsrichtung durch den Konstruktor

- `M(double length)`

Wie bei F existiert auch der einfache Konstruktor

- `M()`

welcher eine Bewegung um die Länge 2 realisiert.

Doch M eignet sich noch für einige zusätzliche Funktionen, welche in Zusammenhang mit z.B. RV interessante Effekte erzielen.

Als eine dieser Funktionen hat der Anwender die Möglichkeit, an M als Parameter einen anderen polygonalen Knoten zu übergeben. Der Konstruktor

- `M(Transform node)`

setzt dann die aktuelle Position auf den nächst möglichen Punkt auf der Oberfläche des übergebenen Knotens. Damit kann z.B. die aktuelle Position immer auf einer Oberfläche eines polygonalen Objektes gehalten werden, und es können somit komplexe Vorgänge modelliert werden. Diese Technik funktioniert nur bei polygonalen Objekten.

Eine Erweiterung dieser Funktion ist der Konstruktor

- `M(Node intersectionobject, double x, double y, double z)`

woran nicht nur ein anderer Knoten übergeben wird, sondern auch zusätzlich ein Vektor (x, y, z). Nun kann der Anwender explizit angeben, in welcher Richtung ein Schnittpunkt gesucht werden soll. Hierzu sendet M einen Strahl entlang des eingegeben Vektors aus und ermittelt einen Schnittpunkt mit dem übergebenen Objekt. Wenn einer gefunden wird, bewegt sich M an diese Stelle, ansonsten passiert nichts. Diese Funktion ist z.B. geeignet, Objekte entlang einer Oberfläche zu verteilen und dabei die Abstände dieser Objekte konform zu halten. Im Kapitel 7.5 wird das Beispiel „Citygenerator“ genauer erklärt. Dort kommt dieser Konstruktor bei der Platzierung der Gebäude auf der Oberfläche zum Einsatz.

5.1.3. D und DMul

In L-Systemen setzt D (Diameter) den Durchmesser aller nachfolgenden F-Objekte. DMul multipliziert den aktuellen Radius mit dem übergebenen Parameter und setzt in allen nachfolgenden F-Objekten diesen Radius. In der Implementierung für Maya wird genau dieses umgesetzt. Um das zu erreichen, wird in jedem Knoten eine Instanz der Klasse *TurtleStates* mitgeführt. Diese speichert den aktuellen Radius des Knotens, auch wenn der Radius nur auf F einen Effekt hat. Am Ende einer Regel wird der komplette Maya-Graph traversiert in einer rekursiven Tiefensuche, und die Turtlestates werden abgearbeitet. Dabei gibt jeder Knoten, beginnend bei der Wurzel, seinen Turtlestate an seine nachfolgenden Knoten weiter. Speziell D und DMul überschreiben diesen Turtlestate mit ihrem neuen Radius. Somit bekommen alle in der Hierarchie darunter liegenden Knoten diesen neuen Radius übergeben und F setzt ihn um, sofern der Anwender nicht einen anderen Radius übergeben hat.

5.1.4. RL, RU, RH

Die drei Knotentypen RL, RU und RH realisieren Drehungen im dreidimensionalen Raum. Jeder Knoten dreht um eine lokale Achse, ist also abhängig von der aktuellen Lage im globalen Koordinatenraum.

Alle drei besitzen einen einfachen Konstruktor

- `RX(double value)`

wobei X durch L, U oder H zu ersetzen ist.

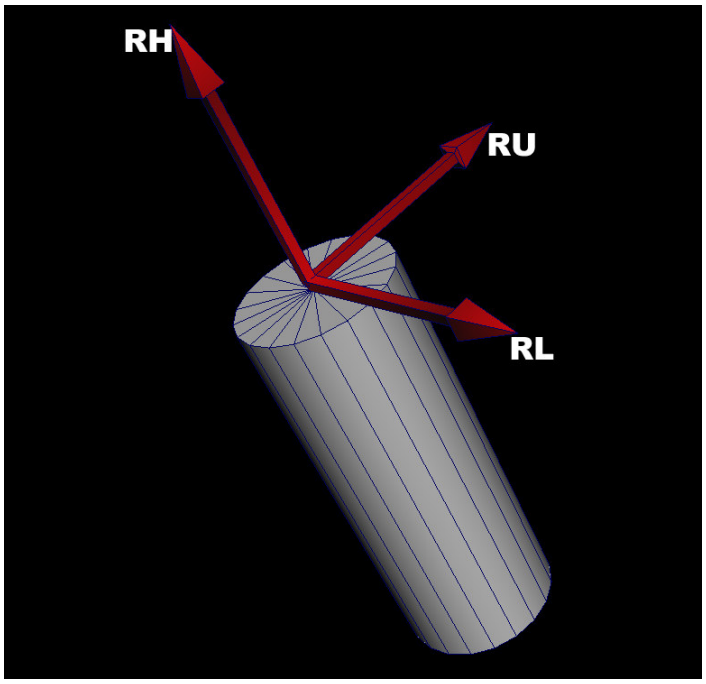


Abbildung 4: Rotationsachsen von RH, RU und RL

Abbildung 4 zeigt die Rotationsachsen in Maya. RH dreht um die Achse, die in Wuchsrichtung zeigt, RU um jene Achse, die den lokalen Up-Vektor darstellt, und RL um die nach links wegzeigende Achse.

Da diese Knoten, ebenso wie die anderen L-System-Befehle (bis auf F) nur einen einfachen *Transform*-Knoten in Maya erzeugen, ist es für viele Modellierungen kein Problem, mehrere dieser Rotationsknoten nacheinander zu verwenden. Dabei ist aber zu beachten: Je mehr Knoten, desto tiefer die Hierarchie. Sollte diese zu tief werden, dauert die Ausführung jeder Regel immer länger.

Es ist also empfehlenswert, bei vielen Rotationen einfach ein Modul zu erstellen und die Rotationen über die Methode *setRotate* der Klasse *Transform* durchzuführen. Dies verringert die Anzahl der verwendeten Knoten erheblich und äußert sich oftmals merklich in der Geschwindigkeit.

5.1.5. RV

RV ist ein Knoten, mit dem man einen Trend für das Wachstum nach oben oder unten im globalen Koordinatensystem geben kann. Der Standardkon-struktor

- `RV(double value)`

hat nur einen Parameter, dieser ist eingeschränkt zwischen -1 und 1, wobei -1 ein Wachstum nach oben darstellt und 1 entsprechend nach unten.

Wählt der Anwender Werte dazwischen, wird die aktuelle Wuchsrichtung nicht vollständig auf (0, -1, 0) bzw. (0, 1, 0) gedreht, sondern auf einen Vektor zwischen der aktuellen Richtung und einem dieser beiden Vektoren, je nachdem ob der Parameter positiv oder negativ ist. Bei einem Parameter von 0 passiert nichts, der Knoten wird aber dennoch erstellt. Abbildung 5 zeigt die Wirkung von RV. In diesem Fall wurde ein Wert von -0.2 eingesetzt, die F-Knoten haben eine Länge von 8.

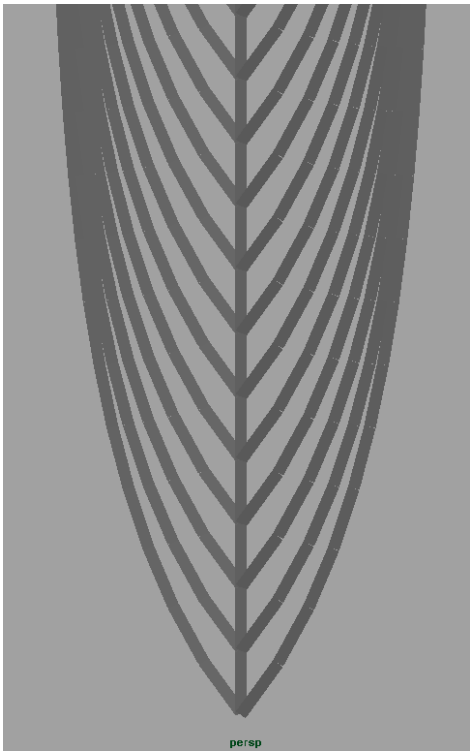


Abbildung 5: RV in Aktion

Dieser Standardkonstruktor entspricht nicht exakt dem der Implementierung von RV in GroImp. In dieser Variante ist bei einem Wert von 1 bzw. -1 ein striktes Wachstum in Richtung (0,-1,0) bzw. (0, 1, 0) zu verzeichnen, wäh-rend in GroImp dies erst nach mehreren Schritten erreicht wird. Um ein we-niger steiles Konvergenzverhalten zu modellieren, müssen in dem Fall kleine-re Werte als -1 bzw. 1 gewählt werden.

Dieser Knoten eignet sich jedoch noch für weitere Funktionen, die im Folgenden beschrieben werden. Zunächst ist es von Vorteil, wenn die Pflanze nicht strikt den Trend nach oben oder unten einhalten muss. Wenn man an Phototropismus denkt, kann es durchaus auch eine andere Richtung sein. Der Konstruktor

- `RV(double value, double x, double y, double z)`

implementiert dieses Verhalten. Der Betrag des Vektors wird nicht beachtet, einzig und allein die Richtung ist entscheidend. Eine Vorwärtsbewegung in die berechnete Richtung muss danach z.B. mit einem M- oder F-Knoten realisiert werden. Abbildung 6 zeigt die Wirkung von RV mit einem Faktor von -0.2 und mit dem Vektor (1, 1, 1) als Parameter.

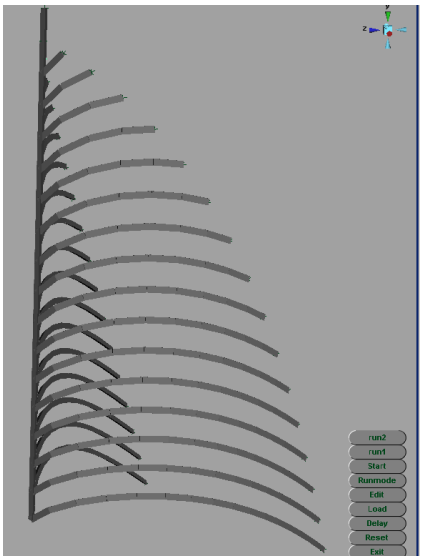


Abbildung 6: RV mit Vektorangabe

Ein weiteres wichtiges Verhalten von Pflanzen ist das Wachstum entlang einer Oberfläche. Hierfür existieren zwei Konstruktoren:

- `RV(double value, Node m)`

und

- `RV(double value, Node m, double x, double y, double z)`

Der Erstere sucht als Wuchsrichtung den nächstgelegenen Punkt auf der Oberfläche des Knotens m. Ist die Oberfläche erreicht, z.B. durch die Bewegung über M-Knoten, ist das Verhalten kaum vorhersehbar, je nachdem, wie die Oberfläche des Knotens m aussieht. In diesem Fall sind zusätzliche Mechanismen notwendig, um das Verhalten kontrollieren zu können.

Der Zweite verhält sich äquivalent zum Ersten, allerdings kann man das Verhalten über den Vektor (x, y, z) steuern. Dieser Vektor gibt einen Trend vor, in welche Richtung sich das Wachstum entwickeln soll. Hierbei entscheidet der Betrag des Vektors über die Stärke des Trends.

Intern nutzen beide Konstruktoren die Maya-Api-Methode *getClosestPoint*, um den Zielpunkt zu berechnen. Dies alles findet im globalen Koordinatenraum statt, somit sind die berechneten Drehungen unabhängig von der aktuellen Lage im Raum.

Insgesamt ist RV ein mächtiges Werkzeug, um natürliches Verhalten von Pflanzen zu modellieren, ist aber auch für viele andere Zwecke sinnvoll.

5.1.6. Module

XL bietet die Möglichkeit, über das Stichwort *module* im Quelltext eigene Knoten zu deklarieren. Diese Module werden noch vor der Klassendefinition angeführt und können im Quelltest wie normale Knoten verwendet werden. Implementiert wurden diese so, dass sie von *Locator* ableiten, es entsteht also in Maya physisch ein *Transform*-Knoten, auf den alle Operationen der *Transform*-Klasse angewendet werden können. Module können Werte speichern, indem bei der Deklaration deren Typen bekannt gegeben werden und bei der Erstellung des Knotens entsprechende

Werte übergeben werden. Es ist dem Anwender möglich, ein Modul auch von anderen Klassen ableiten zu lassen.

```
[XL]
module meristem(int step) extends Transform;
public class mod{
      public static void method1()[
            Axiom ==> F(5) meristem(5);
            meristem(p), (p>0) ==> F(p) meristem(--p);
      ]
}
```

In diesem Beispiel speichert das Modul *meristem*, hier abgeleitet von *Transform*, einen Integer. In der ersten Ableitungsregel wird zunächst das Modul initialisiert und der Wert 5 übergeben. In der zweiten Regel wird der Wert aus dem Modul in die Variable p eingelesen, der Datentyp wird automatisch erkannt. Die nachfolgende Bedingung (p>0) gibt an, dass die Regel nur solange ausgeführt wird, wie die Bedingung erfüllt ist. Das Modul wird auf der rechten Seite durch ein F der Länge p und einem neuen Modul ersetzt. Diesmal wird p um einen Wert gesenkt, bevor dieser an das Modul übergeben wird.

Mit dieser Technik ist es möglich, Ableitungen auf bestimmte Schrittzahlen zu beschränken. Hierbei kann die aktuelle Schrittzahl für weitere Berechnungen eingesetzt werden.

5.2. Weitere XL-Knoten

Neben den L-System-Knoten gibt es noch weitere, nützliche Knoten. *Node* und *Transform* bilden hierbei eine Ausnahme: Sie sind die Grundlage aller anderen Knoten, da diese davon erben. *Node* erzeugt keinen eigenen Knoten, der in Maya durch ein Objekt repräsentiert wird, sondern ist nur in XL gespeichert.

5.2.1. Node

Node ist die Basisklasse für alle zur Verfügung stehenden Knoten. Sie enthält alle Methoden für die Verwaltung von XL-Knoten und ebenso einige Methoden für den Anwender. Alle hier beschriebenen Methoden sind ebenso für alle anderen Knotenklassen gültig. Nicht beschrieben werden die Methoden zum Abfragen und Setzen von Attributen. Dies befindet sich im Kapitel 5.4.

Der Konstruktor

- `public Node()`

erzeugt kein reales Objekt in Maya und wird beim Ableiten automatisch aufgerufen. Für den Fall, dass doch ein physikalisches Objekt erzeugt werden soll, ist es besser, die eigene Klasse von einem Primitivobjekt wie *Transform* abzuleiten und einen der expliziten Konstruktoren über *super()* aufzurufen. Wenn man aber nur einen Behälter für Methoden braucht, die eventuell verspätet auszuführen sind, kann man durchaus von *Node* ableiten. Dabei ist aber zu beachten, dass diese Klasse nicht in Suchmustern oder auf der rechten Seite eingesetzt wird. In den imperativen Blöcken von XL kann aber durchaus via *new* ein neues Objekt der selbst definierten Klasse erstellt und verwendet werden.

Die Klasse *Node* stellt darüber hinaus noch einige andere Methoden bereit:

- `String getDAGPath()`

Diese Methode liefert den DAG-Path aus Maya als String zurück. Der DAG-Path ist ein gutes Mittel, um Knoten in Maya eindeutig im DAG zu identifizieren. Viele MEL-Methoden erwarten als Parameter einen DAG-Path. Zusammen mit der MEL-Klasse ist diese Methode ein wichtiges Werkzeug.

- `String getName()`

gibt den Namen des Knotens zurück. Manche MEL-Methoden brauchen als Parameter diesen Namen. Ebenso kann man anhand des Namens auch Suchen durchführen.

```
[XL]
public static void findname()[
(* h:Node *) ::>
      {
            println(h.getName());
            println(h.getDAGPath());
      }

]
```

Dieses Beispiel führt eben eine solche Suche durch und gibt den Namen aller gefundenen Knoten und deren DAG-Path aus. Die Ausgabe in einer Szene mit nur einer Kugel ist:

```
[MEL]
// pSphere1
// |pSphere1
```

- `Node getParentNode()`

sucht den Vater im DAG das Knotens und gibt diesen als Knoten zurück. Diese Methode ist oftmals dann sinnvoll, wenn man nicht für einfache *parent-child*-Beziehungen ein Suchmuster anwenden möchte. Dies ist eine reine Effizienzfrage, denn der direkte Zugriff auf den Vater ist bei weitem schneller. Es wird davon ausgegangen, dass im DAG ein Knoten nur einen Vater haben kann. Ausnahme sind Instanzen von Maya-Objekten, denn hier besitzt ein Shapeknoten mehrere *Transform*-Knoten darüber. In diesem Fall wird der erste Vater zurückgegeben. Genauso funktioniert

- `Node getChildNode(int index)`

nur dass hier ein Index mit übergeben werden muss. Es ist typisch für einen DAG, dass ein Knoten mehrere Kinder hat. Um über alle Kinder iterieren zu können, liefert

- `int getChildCount()`

die entsprechende Anzahl von Kindknoten zurück. Im Folgenden ein passen-des Beispiel, angewendet auf die Szene mit nur einer Kugel mit dem Namen „pSphere1“:

```
[XL]
(* h:Node *), (h.getName().equals("pSphere1")) ::>
{
      Node m = h.getParentNode();
      if (m != null)
      {
            println(m.getName());
      }
      int count = h.getChildCount();
      for (int i=0; i<count; i++)
      {
            m = h.getChildNode(i);
            println(m.getName());
      }
}
```

Die entsprechende Ausgabe dazu ist:

```
[MEL]
// world
// pSphereShape1
```

- `boolean setShader(string ShadingGroup)`

ist für die Vergabe von Shadern zuständig. Konnte kein Shader mit dem Namen gefunden werden, wird *false* zurückgegeben, ansonsten *true*. Der Shader sollte zuvor in Maya vorbereitet worden sein, da ein Shading-Network[1], also ein komplexes Netzwerk von DG-Knoten zur Erzeugung von Materialien, zu erstellen, die Möglichkeiten von XL übersteigt. Alternativ dazu wäre es möglich, ein MEL-Skript zu schreiben, dessen Parameter die Art des Shaders steuert, und dieses dann mittels MEL-Klasse auszuführen. Den dann erstellten Shader kann man mit dieser Methode anschließend auf einen Knoten anwenden.

```
[XL]
(* h:Node *), (h.getName().equals("pSphere1")) ::>
{h.setShader("blinn1SG");}
```

In diesem Beispiel wird ein bereits existierender Shader auf einen Knoten namens „pSphere1“ angewendet.

- `void setVisibility(int onoff)`

schaltet die Sichtbarkeit eines Knotens ein oder aus. Dabei ist zu beachten, dass dieses Attribut von Maya auf die Kinder übertragen wird. Deshalb ist es nicht möglich, einzelne Objekte innerhalb einer Hierarchie-Kette auszublenden.

Die restlichen Methoden zum Setzen und Lesen von Attributen werden im Kapitel 5.4 genauer erläutert.

5.2.2. Transform

Die Klasse *Transform* leitet von *Node* ab und sämtliche Primitivobjekte leiten wiederum von *Transform* ab. Sie stellt alle notwendigen Methoden zum Zugriff auf die Transformationsmatrix, deren einzelne Komponenten und Pivots bereit, ebenso einige Methoden für zusätzliche knotenbezogene Berechnungen.

Die Klasse *Transform* besitzt einen einfachen Konstruktor:

[1] Zu Shading-Networks siehe auch: [17], S. 130ff.

- `public Transform()`

Dieser erzeugt in Maya ein *null*-Objekt. Das ist ein einzelner *Transform*-Knoten, der in der Hierarchie eingeordnet Auswirkungen auf die Position, Skalierung und Lage seiner Kinder hat. Sämtliche L-System-Knoten und Primitvobjekte sind solch ein *null*-Objekt und haben somit ebenfalls Zugriff auf die hier beschriebenen Methoden.

Um auf die Transformationsmatrix zugreifen zu können, stehen einige weitere Methoden zur Verfügung. In erster Linie unterscheiden die meisten Methoden in der Transformation im lokalen und globalen Koordinatensystem. Dies schlägt sich im Parameter *boolean global* nieder. Die Erklärung deshalb zu Beginn, damit nicht bei jeder Methode dies wiederholt werden muss.

Die aktuelle Position des Objektes im Raum kann über

- `Point3d getTranslate()`
- `double[] getPosition()`
- `double[] getPosition(boolean global)`
- `Point3d getPositionAsPoint()`
- `Point3d getPositionAsPoint(boolean global)`

bestimmt werden. Bei der ersten Methode werden die Werte ausgelesen, wie sie der Anwender auch im Attributeditor auslesen kann, also die relative Translation zu seinem *parent*. Die anderen Methoden lesen die absolute Position aus. Die Position eines Objektes ist abhängig davon, welchen Pivot man betrachtet. In diesem Fall wird der *Scale*-Pivot ausgelesen und zurückgegeben. Man hat die Wahl zwischen einem Double-Array oder einem Point3d als Rückgabetyp. Point3d steht in dem Package *javax.vecmath* zur Verfügung, welches standardmäßig importiert wird und genutzt werden kann.

Die Rotation und Skalierung kann im Unterschied zu den Translationen nur im lokalen Raum abgerufen werden:

- `double[] getRotation()`
- `double[] getScale()`

Entsprechend zu den *get*-Methoden für Translation, Scale und Rotation gibt es die passenden *set*-Methoden:

- `void translate(double x, double y, double z)`
- `void translate(double x, double y, double z, boolean global)`
- `setRotation(double x, double y, double z, boolean global)`
- `void setScale(double x, double y, double z)`

Die bisher beschriebenen Methoden führen absolute Translationen und Rotationen durch. Für den Fall der relativen Veränderung gibt es:

- `void translateBy(double x, double y, double z)`
- `void translateBy(double x, double y, double z, boolean global)`
- `void rotateBy(double x, double y, double z, boolean global)`

Dies sind die wichtigsten Methoden zum Verändern von Objekten im Raum. Es ist jedoch auch möglich, direkt auf die Transformationsmatrix zuzugreifen:

- `double[] getTransformationMatrix(boolean exclusive)`
- `void setTransformationMatrix(double[] Matrix)`

Hierbei ist für die double-Arrays eine bestimmte Anordnung der Werte zu beachten:

$$T = \begin{vmatrix} 1 & 0 & 0 & 0 \end{vmatrix}$$

| 0 1 0 0 |
| 0 0 1 0 |
| tx ty tz 1 |

Zusätzlich zu den aufgeführten Methoden ist es ebenso möglich, auf die Pivots zuzugreifen mit:

- `void setRotatePivot(boolean global, boolean balance, double x, double y, double z)`
- `void setRotatePivot(double x, double y, double z)`
- `void setScalePivot(boolean global, boolean balance, double x, double y, double z)`
- `void setScalePivot(double x, double y, double z)`

Balance ist ein Parameter, der, falls er auf *true* gesetzt ist, zu der Translation des *scale*-Pivots eine kompensierende Transformation hinzufügt, damit die Transformationsmatrix sich nicht ändert. Auch hier hat der Anwender die Möglichkeit, zwischen globaler und lokaler Transformation zu entscheiden. Die Methoden ohne den Parameter bleiben im lokalen Raum.

Über diese Methoden zum Zugriff auf die Transformationsmatrix hinaus enthält die Klasse *Transform* noch ein paar weitere hilfreiche Methoden:

- `double distance(Transform m)`
- `double distanceLinf(Transform m)`

Diese beiden Methoden berechnen die Distanz zwischen zwei *Transform*-Knoten. *distanceLinf* berechnet die l-infinite Distanz mit der Definition:

```
distanceLinf = MAX[abs(x1-x2), abs(y1-y2)]
```

wobei *abs* den Absolutwert bestimmt und *MAX* den größten Wert unter den Argumenten zurückgibt.

Dies ist nicht die „echte“ euklidische, ist aber wesentlich schneller und nützlich bei vielen Vergleichsoperationen, bei denen es nur darauf ankommt, welcher Knoten näher dran oder weiter entfernt ist.

Eine weitere interessante Methode ist

- `boolean isinVolume(Node m)`

Hierbei wird berechnet, ob sich der Schwerpunkt des Knotens innerhalb des polygonalen Volumens des Knotens m befindet. Dazu werden zwei Strahlen entlang der globalen y-Achse in Maya ausgesandt und die Schnittpunkte mit dem Objekt gezählt. Somit sollte auch bei konkaven Objekten ein zuverlässiges Ergebnis zur Verfügung stehen. Mittels dieser Methode ist es möglich, weit reichende geometrische Zusatzbedingungen in ein XL-Programm einzubauen. Anhand eines vormodellierten Objektes lässt sich z.B. die Ausbreitung eines Wachstums einschränken, denn sobald das Objekt erreicht wurde, wird die Regel einfach nicht mehr ausgeführt. Ein kleines Beispiel:

```
[XL]
x:Sphere, (*h:Transform*), (h.getName().equals(„Testobjekt“)),
(!x.isinVolume(h)) ::> println(x.getName());
```

Diese Regel sucht alle Kugeln, die sich nicht im Volumen von „Testobjekt“ befinden, und gibt deren Namen aus.

- `Node getShapeNode()`

- `Node[] getShapeNodes()`

sind Methoden zum Ermitteln der darunter liegenden *Shape*-Knoten. getShapeNode() ermittelt den ersten Knoten. Da *Transform*-Knoten in der Regel nur einen *Shape*-Knoten direkt unter sich haben, ist diese Methode der direkte Zugriff, ohne ein Array zu erzeugen, während getShapeNodes() alle *Shape*-Knoten zurückliefert. Beide Methoden geben *null* zurück, sollte kein *Shape*-Knoten direkt unter diesem Knoten in der Hierarchie liegen.

Zuletzt befindet sich in der Klasse noch die Methode

- `void makeFreeze(boolean positiononly)`

Maya-Anwendern müsste diese Methode geläufig sein. Sie verändert die Geometriedaten in dem *Shape*-Knoten so, dass die Attribute im Attributeditor, also die relativen Verschiebungen und Rotationen auf 0 zurückgeführt werden, ohne dass das Objekt seine Lage im Raum verändert. Die Transformationsmatrix wird somit auf die Einheitsmatrix zurückgesetzt. Der Parameter *positionsonly* gibt an, ob sich diese Veränderung allein auf die Positionskomponente der Transformationsmatrix auswirken soll. Die Rotation bleibt somit erhalten.

Alles in Allem ist *Transform* der wichtigste Knotentyp und bietet alle Möglichkeiten zum Umgang mit Objekten im dreidimensionalen Raum. Unter Zuhilfenahme des Konstruktors lassen sich schnell anwenderdefinierte Hilfsobjekte konstruieren. Das Fehlen eines *Shape*-Knotens sorgt dafür, dass die Anzahl der Knoten, erzeugt durch Hilfsobjekte, halbiert wird. Da alle anderen Knoten (bis auf Node) von ihm erben, gelten alle hier beschriebenen Methoden auch für alle anderen Knoten.

5.2.3. Axiom

Wenn der Anwender eine neue Szene beginnt und ohne irgendwelche vorbereiteten Objekte arbeiten möchte, muss ein ersetzbarer Anfangsknoten existieren, von dem aus das Wachstum beginnen kann. Somit wird nach dem Start des Compilers zunächst dieser Knoten sowohl in Maya als auch in XL erzeugt. In Maya ist er am Namen „Axiom“ zu erkennen und ist ein einfacher *Transform*-Knoten.

Die Klasse *Axiom* besitzt keinen öffentlichen Konstruktor, kann also nicht auf der rechten Seite einer Regel eingesetzt werden.

5.2.4. Primitivobjekte

In XL4Maya steht eine Anzahl von Primitivobjekten zur Verfügung, welche zur Modellierung von polygonalen Strukturen verwendet werden können. Nurbsobjekte werden bis auf die Nurbs-Extrusion entlang eines Pfades, welche im Kapitel 5.7 erläutert werden, nicht unterstützt. Grund dafür ist, dass schon ein einfacher Kubus aus 6 Nurbs-Objekten besteht, nämlich für jede Seite eine Nurbs-Fläche. Der Anwender müsste also statt mit einem Knoten mit sechs Knoten umgehen.

Dennoch steht eine Anzahl von Primitvobjekten zur Verfügung: Cube, Sphere, Cylinder, Helix, Cone, Pipe, Pyramid, Torus, Plane und Prism. In der Erstellung verhalten sie sich alle gleich und besitzen jeweils nur einen parameterlosen Konstruktor.

5.2.5. Weitere Knotentypen

Um dem Anwender eine Vereinfachung zu bieten, steht der Knotentyp *Leaf* zur Verfügung. An sich wird nur eine *Plane* erzeugt, allerdings mit einer Besonderheit: Wie im Knotentyp F befindet sich der Scalepivot am unteren und der Rotatepivot am oberen Rand. Somit ist bei der Erzeugung gewährleistet, dass das Blatt korrekt z.B. an den Ästen oder Zweigen platziert wird. Ein Material wird nicht automatisch erzeugt, allerdings steht in der Beispielszene *Bush* ein Material samt Textur zur Verfügung, welches nur noch via *setShader* zugewiesen werden muss.

Ein weiterer wichtiger Knotentyp ist der *Locator*. In Maya sind dies Hilfsobjekte mit visuellem Feedback, es existiert also ein *Shape*-Knoten, welcher im Ansichtsfenster von Maya ein kleines grünes Kreuz darstellt. Im Renderer ist dies nicht zu sehen.

In der Anfangsphase der Programmierung leiteten alle L-System-Befehle und Module von *Locator* ab. Schnell stand fest, dass zum einen der zusätzliche *Shape*-Knoten auf Grund der größeren Baumtiefe die Ableitungsgeschwindigkeit verringerte und zum anderen die Darstellungsgeschwindigkeit im Maya-Ansichtsfenster verlangsamte. Dem Anwender steht nach wie vor offen, Module von *Transform* ableiten zu lassen, indem er bei der Deklaration explizit *extend Transform* angibt. Dies ist hilfreich bei der Fehlersuche, da anhand dieser Objekte relativ genau klar wird, welcher Knoten wohin zeigt.

Locator hat nur einen parameterlosen Konstruktor, verhält sich ansonsten genauso wie *Transform*.

5.2.6. Prädikatklassen

Prädikatklassen sind eigentlich keine Klassen, sondern Methoden, definiert in einer besonderen Klasse. Doch sie dienen zur Klassifikation von Objekten, deshalb wäre die Bezeichnung dennoch zutreffend. In erster Linie besteht solch eine Klassifikation aus zwei Methoden. Die eine Methode bekommt vom Compiler einen Knoten geliefert, und die Methode entscheidet, ob dieser Knoten zu dieser Klasse gehört. Die andere Methode hingegen erzeugt neue Objekte oder ein neues Objekt, je nach Implementierung. Dieser neue Knoten wird an den Compiler zurückgeliefert.

Diese Methoden verhalten sich in XL wie Knotentypen, sie können also in Suchmustern und rechten Seiten eingesetzt werden, je nachdem, ob beide Methoden oder nur eine von beiden implementiert wurden. In XL4Maya gibt es genau zwei Prädikatklassen, welche dem Anwender eine zusätzliche Möglichkeit bieten.

Zunächst wäre da die Duplicate-Methode. Vom ersten Moment der Implementierung war klar, dass es dem Anwender gestattet sein sollte, auf vorhandene Objekte in Maya über XL zugreifen zu können. Nun ist es aber nur selten so, dass eine Szene nur aus den erkennbaren Primitvobjekten besteht (wir erinnern uns an Kapitel 4.1.3). Ein kleines Anwendungsbeispiel:

Der Anwender hat in seiner Szene einige Varianten eines Gebäudes modelliert und möchte diese im Raum verteilen. Besteht dieses Gebäude nur aus einem *Transform*- und *Shape*-Knoten, d.h. er hat die gesamte Geometrie zusammen

geführt, kann der Anwender über ein einfaches Suchmuster auf alle Objekte zugreifen.

Doch was ist bei dem Fall, dass man ein Gebäude genau fünf Mal verteilen möchte? Hier liegt vielleicht folgende XL-Zeile nahe:

```
[XL]
x:Transform, (x.getName().equals("Haus1")) ==> x x x x x;
```

Dieser Code endet mit einem Fehler. Duplizieren auf diese Art ist nicht erlaubt. An dieser Stelle hilft die Duplicate-Methode aus:

```
[XL]
x:Transform, (x.getName().equals("Haus1")) ==> x y:Duplicate(„Haus1“,
true);
```

Die Klasse Duplicate hat zwei Parameter und sieht folgendermaßen aus:

- `Node Duplicate(String name, boolean withchildren)`

Der Parameter *name* ist der Name des zu duplizierenden Objektes, der zweite Parameter entscheidet, ob nur dieser eine Knoten oder alle in der Hierarchie darunter liegenden mit dupliziert werden. Die dabei neu entstehenden Knoten werden automatisch in XL angemeldet und können in Suchmustern verwendet werden. Um den Rückgabetyp muss sich der Anwender nicht kümmern, dies ist allein für XL von Interesse. Der Parameter *withchildren* bietet noch einen weiteren Vorteil: Hat der Anwender dieses Haus nicht nur aus einem Stück modelliert oder zu einem Knoten zusammengeführt, sondern eine ganze Palette an Objekten erstellt, brauchen diese nur vom Anwender z.B. unter einem *Transform*-Knoten in die Hierarchie einsortiert werden, um über das Duplizieren dieses *Transform*-Knotens komplette Objektgruppen zu klonen und zu verteilen. Die Möglichkeiten sind schier unbegrenzt, da der Anwender jetzt nicht mehr auf XL-Primitiv-Objekte angewiesen ist.

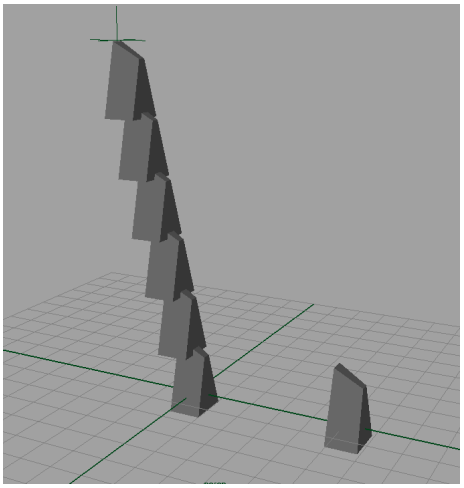


Abbildung 7: Wirkung von Duplicate

An dieser Stelle noch ein zusätzlicher Tipp. Selbstmodellierte Objekte[1] können wie der F-Knoten eingesetzt werden. Der Anwender muss, um dies zu erreichen, den *Scale*-Pivot an den Punkt, an dem das Objekt ansetzen soll, und den *Rotate*-Pivot an den Ort, an den Kind-Objekte wiederum mit ihrem eigenen *Scale*-Pivot bewegt werden sollen, verschieben. Abbildung 7 zeigt dieses Verhalten an einem einfachen Beispiel.

Der zu der Abbildung gehörende Code ist recht einfach:

```
[XL]
module meristem;
public class Dup{
            public static void runDup() [
                  Axiom ==> meristem;
                  meristem ==> x:Duplicate("einobjekt", true) meristem;
            ]
}
```

Das in Abbildung 7 abseits liegende Objekt ist das selbstmodellierte und kann nach Anwendung von XL versteckt oder gelöscht werden.

Ein anderes Anwendungsbeispiel sind eigene Äste, die dann in Nahaufnahmen die weniger detaillierten Fs ersetzen, aber genauso funktionieren. Der Anwender kann sich sogar die Konstruktions-History (siehe S. 43) zu Nutze machen, um eine vielfältige Veränderung dieser Objekte zu erreichen. Hierbei muss nach dem Duplizieren nur der entsprechende Parameter über das Anwenden der *setValue*-Methode angepasst werden, z.B. an einen zufälligen Wert.

Obwohl *Duplicate* eine Methode ist, wird sie verwendet wie eine Klasse. Doch kann sie nicht in Suchmustern untergebracht werden.

- `boolean Selection(Node n)`

prüft alle Objekte, ob sie vom Anwender selektiert wurden. Somit lassen sich komplexe, anwenderspezifische Suchmuster erzeugen.

5.3. Verspätetes Ausführen von Code

Für das Erstellen von Regeln ist es wichtig, die Möglichkeit zu haben, Regeln parallel auszuführen. Hierbei kann man natürlich nicht von echter paralleler Ausführung sprechen, sondern von Quasi-Parallelität. Diese wird benötigt, wenn der Compiler für jedes Mal, wenn das Suchmuster auf der linken Seite der gerade abzuarbeitenden Regel auf eine Situation im Maya-Graphen passt, auf den gleichen Zustand des Systems zurückgreift, wie bei den vorhergehenden Schritten. Zu beachten ist, dass sich die Parallelität auf die Ausführung einer einzigen Regel bezieht. Im Beispiel „Game of Life“ des Kapitels 7.2 wird auf dieses Prinzip zurückgegriffen und es genauer erklärt.

Zum verspäteten Ausführen von Code befinden sich in der Klasse *User* vier wichtige Methoden. Ebenso haben einige Methoden, wie *setKeyframe* oder *setValue*, ein Pendant für das verspätete Ausführen, in diesen Fällen *setKeyframeLate* oder *setValueLate*. Hierbei müssen die folgend beschriebenen Methoden nicht mehr angewendet werden.

[1] Hinweise zur Nutzung verschiedener Modelliertechniken in Maya in [18] und [19].

Allen vier Methoden liegt das gleiche Prinzip zugrunde, allerdings erfüllt jede Methode in einer anderen Situation ihren Zweck. Der erste Parameter ist zugleich Unterscheidungskriterium, doch darauf wird etwas später eingegangen. Die restlichen Parameter verhalten sich bei allen Methoden gleich.

Zunächst muss der Name der Methode übergeben werden, welche verspätet ausgeführt werden soll, ein einfacher String erfüllt dieses. Als zweites muss ein *Object-Array* die zu übergebenen Parameter enthalten. Basistypen wie *int* oder *double* müssen in Objekte über *new Double(double value)* umgewandelt werden. Zuletzt müssen noch die Typen der Argumente in einem *Class-Array* festgehalten und übergeben werden.

Nun zu dem ersten Parameter, welcher das Unterscheidungsmerkmal aller vier Methoden ist. Zunächst gibt es die vorimplementierten Knoten aus XL4Maya, wie F oder RV. Die darin enthaltenen Methoden müssen mit der Methode

- `void invokeMethodLate(Node m, String methodName, Object[] arguments, Class[] argumentTypes)`

ausgeführt werden, wobei der erste Parameter ein normaler Knoten ist.

Die zweite Kategorie sind Methoden, welche aus anderen vorimplementierten Klassen wie *MEL* oder *User* stammen. Hierfür ist die Methode

- `void invokeMethodLate(Class m, String methodName, Object[] arguments, Class[] argumentTypes)`

zuständig. Der Unterschied besteht darin, dass Knoten erst zur Laufzeit von XL bekannt sind und diese Methoden bereits zur Kompilierzeit. Der XL-Compiler behandelt beide Kategorien unterschiedlich, somit ist eine Unterscheidung notwendig.

Nun ist es dem Anwender möglich, eigene Knoten durch Ableitung von *Node* in XL zu definieren. Für Methoden innerhalb dieser Knoten ist

- `void invokeXLNodeMethodLate(Node m, String methodName, Object[] arguments, Class[] argumentTypes)`

die passende Methode. Nun ist es auch möglich, Klassen ohne Ableitung von *Node* in XL zu verwenden. In diesem Fall gilt aber die Beschränkung, dass die Klassen zumindest von *XObjectImpl* ableiten. Für die darin definierten Methoden sollte

- `void invokeXLClassMethodLate(XObjectImpl m, String methodName, Object[] arguments, Class[] argumentTypes)`

verwendet werden.

Besonders wichtig ist das verspätete Ausführen bei der Ausführung von MEL aus XL heraus, natürlich nicht in allen Fällen. Es ist aber möglich, dass MEL den Zustand der Szene derart verändert, dass die Suchmuster in der Regel eventuell nicht mehr richtig arbeiten und somit unerwartete Ergebnisse entstehen. MEL via XL wird im Kapitel 5.6 genauer behandelt.

5.4. Properties / Attributes

Der Anwender kann in XL auf Eigenschaften von Knoten, so genannte Properties, zugreifen und diese verändern. Hierbei sind diese Eigenschaften Attribute des entsprechenden Maya-Objektes, welches der XL-Knoten repräsentiert. Nun stellt die Einschränkung von XL auf den Maya-DAG an dieser Stelle ein Problem dar: Wenn der Anwender den Radius einer Kugel verändern möchte, befindet sich dieses

Attribut im *kPolySphere*-Knoten, welcher ein reiner DG-Knoten und mit dem *Shape*-Knoten der Kugel verbunden ist. Nun ist es an XL4Maya, dieses entsprechende Attribut dennoch zu finden.

```
[XL]
x:Sphere ::> x[radius] = new Double(4.0);
```

Dieses Stück Code sucht in einer Szene alle Kugeln und setzt deren Radius auf den Wert 4.0. An dieser Stelle muss der Wert über *new* übergeben werden. Der Grund dafür ist recht komplex. Zunächst ein paar Erläuterungen dazu, wie XL4Maya Attribute zu einem Knoten sucht.

XL4Maya bietet mit dieser Technik die Möglichkeit, alle Attribute der *History* eines Knotens zur Verfügung zu stellen. Zur Erinnerung: Die *History* sind alle DG-Knoten, die im Sinne des Datenflusses vor dem *Shape*-Knoten eines Objektes liegen. Da eine Kugel nicht nur aus einem *Transform*- und *Shape*-Knoten besteht, sondern auch aus einem DG-Knoten (*kPolySphere*), muss um das Attribut „Radius“ zu finden die gesamte *History* danach abgesucht werden. Dabei gilt keine Beschränkung, dass nur Attribute dieses einen Knotens gefunden werden, sondern durchaus auch Attribute eines jeden anderen Knotens der *History*. Hat der Anwender z.B. ein paar Deformationen auf das Objekt angewandt, außerhalb von XL mit den Maya-Tools oder innerhalb über MEL, ist es durchaus in manchen Fällen wünschenswert, diese Attribute auch zur Verfügung zu haben. So können besondere Effekte erzielt werden.

Nun müssen XL diese Attribute, besser gesagt deren Typen und Namen bereits zur Kompilierzeit bekannt sein. Durch die Reduktion des DGs auf den DAG ist allerdings zur Kompilierzeit nicht klar, welche Attribute welcher Knoten in sich trägt (auch wenn diese Attribute eigentlich zum DG gehören, wird XL vorgegaukelt, sie befänden sich im *Transform*-Knoten). Somit sind die Anzahl, sowie deren Namen und Typen zur Kompilierzeit unbekannt, da der DG sich beliebig verändern kann und somit Attribute dynamisch hinzukommen oder wegfallen. Maya besitzt weit mehr als 1000 Knotentypen, welche in relativ beliebigen Kombinationen im DG auftauchen und somit in der History eines jeden Knotens von XL landen können. Um nun XL jedes Attribut zur Verfügung zu stellen, bekommt XL nur die Information, dass jedes Attribut vom Typ *Object* ist. Somit kann prinzipiell mit jedem Datentyp umgegangen werden.

Dies ist der Grund für diesen Umgang mit Attributen über *new*. Um dem Anwender jedoch noch eine andere Variante zu ermöglichen, gibt es die Methoden:

- `void setValue(String property, double value)`
- `void set3Value(String property, double value1, double value2, double value3)`
- `void setValueLate(String property, double value)`
- `void set3ValueLate(String property, double value1, double value2, double value3)`

Es ist empfehlenswert, diese Methoden zu verwenden. Die Methode *setValue* erwartet als Parameter den Namen des Attributs. Dieser ist für den Anwender im Connection-Editor ablesbar. Die Schreibweise muss exakt sein, damit das Attribut gefunden werden kann. Der zweite Parameter ist der Wert, auf den das Attribut gesetzt werden soll.

Maya hat neben den normalen Attributen auch Zahlentripel als Typ. Die Methode *set3Value* setzt alle drei Werte zugleich.

Zu diesen Methoden gibt es noch die Varianten für das verspätete Ausführen, um quasiparallele Ausführung der Ableitungsregeln zu gewährleisten. Damit wird der Zustand der Mayaszene erst nach Abschluss aller Regeln verändert und eventuelle Parameterabfragen beziehen sich noch auf die originalen, unveränderten Werte.

Im Gegenstück zu diesen Methoden gibt es

- `double getValue(String property)`
- `double[] get3Value(String property)`

zum Abfragen der entsprechenden Attributwerte.

5.5. XL-Edges

Eine weitere interessante Funktion von XL ist das Setzen von gewichteten Kanten zwischen Knoten. Dies ist eine gute Möglichkeit, Beziehungen zwischen Knoten zu modellieren und somit den Wachstumsprozess oftmals deutlich komplexer zu modellieren. Im Beispiel „Game of Life“ (Kapitel 7.2) wird diese Technik angewendet. Statt die Entfernung zwischen zwei Knoten bei jedem Durchlauf neu zu berechnen, wird dies zu Beginn einmal getan und die binäre Beziehung (nah; fern) über (Kante gesetzt; Kante nicht gesetzt) modelliert. Auf diese Information kann bei den nachfolgenden Durchläufen zurückgegriffen werden, z.B. über Suchmuster, und damit auf eine erneute Berechnung der Entfernung verzichtet werden.

In Maya gibt es die Möglichkeit, zwischen Attributen von verschiedenen Knoten, seien sie reine DG- oder DAG-Knoten, Datenflusskanten zu ziehen. Die Richtung der Kante entscheidet, von wo nach wo die Daten fließen. XL4Maya macht sich dieses Prinzip zunutze, um anwenderdefinierte XL-Kanten physikalisch zu verankern. Alternativ dazu hätte auch eine rein in Java entwickelte Lösung implementiert werden können, allerdings wäre dadurch der Verwaltungsaufwand erheblich gesteigert und die Bewahrung der Konsistenz eines der großen Probleme geworden.

Als Beispiel folgende Code-Zeile:

```
[XL]
Axiom ==> x:Cube -EDGE0-> y:Sphere;
```

In diesem Beispiel wird aus dem Axiom ein Quader und eine Kugel erzeugt. Anwenderdefinierte Attribute für Knoten sind eine der großen Stärken von Maya, aber im Rahmen dieser Arbeit auch problematisch, wie im Kapitel 5.4 bereits erwähnt. XL4Maya erstellt nun in diesem Fall mehrere neue Attribute in den Knoten und verbindet diese mit einer DG-Kante. Da der Quader Vater der Kugel werden soll, erhält dieser ein P[index]-Attribut. Der Index ist wichtig, da jeder Knoten Vater beliebig vieler anderer Knoten sein kann. Die Kugel hingegen erhält ein C[index]-Attribut, welches dafür steht, dass der Knoten als Kind der Beziehung gemeint ist. Die Attributtypen sind auf beiden Seiten *Integer* und enthalten den Wert der Kante. Der Anwender hat neun unterschiedliche Kanten zur Verfügung, hinzu kommt noch die SUCCESSOR-Edge, welche der normalen *parent-child*-Beziehung in Maya entspricht. EDGE_0 entspricht 0x4000, also 16384, EDGE_1 0x8000, das heißt 32768. Wenn der Anwender zwischen zwei Knoten sowohl eine EDGE_0 als auch eine EDGE_1 zieht, werden diese beiden Werte über ein logisches AND verknüpft. Heraus kommt 0xC000, also 49152. Da jede Kantengewichtung als Binärwert betrachtet nur ein gesetztes Bit besitzt, entspricht dieses logische AND der Summierung beider Werte. Nicht verankert in diesen Attributwerten ist die

SUCCESSOR-Edge. Doch bei Übergabe der Kantengewichtung an den Compiler wird dieser Wert intern weitergegeben.

Durch diese Technik kann der Compiler in XL anhand der Kantengewichtung über eine Bitmaske erkennen, welche unterschiedlichen Kanten die eine physikalische Kante darstellt. Doch dies sind Interna, welche den Anwender nicht in seiner Arbeit beeinflussen. Für ihn ist wichtig, dass es neben der SUCCESSOR-Edge noch 9 weitere anwenderdefinierte Kanten zwischen Knoten gibt. Wichtig ist ebenso, diese Kanten vom herkömmlichen „Parameter-wiring“ zu unterscheiden, da keine echten Daten fließen, sondern nur die DG-Kanten und Attribute als Hilfsmittel benutzt wurden.

5.6. MEL via XL

In einigen Situationen kann es vorkommen, dass die zur Verfügung gestellte Implementierung von Maya-Objekten und Methoden nicht ausreicht, um einen bestimmten Effekt zu erzielen. Ganz besonders kann dies auftreten, wenn Veränderungen im DG ausgeführt werden sollen. Da XL nur Attributwerte im DG verändern kann, aber z.B. keine neuen Datenflusskanten ziehen oder DG-Knoten erzeugen kann, ist die Klasse MEL das notwendige Hilfsmittel. Die Methoden darin sind *static* deklariert, der Anwender muss also keine Instanz vor der Anwendung erzeugen.

Eine zentrale Rolle spielen die Methoden

- `void execute(String command, boolean echo)`
- `void execute(String command)`

Sie sind für das eigentliche Ausführen zuständig. Bei der Ersteren hat der Anwender die Wahl, ob in Maya ein Ergebnis im Skript-Editor ausgegeben werden soll oder nicht. Diese Option ist mit Vorsicht zu genießen und sollte nur für Debug-Zwecke verwendet werden, da jede Ausgabe die einzelnen Ausführungsschritte von XL ungleich langsamer machen kann. Das eigentliche Kommando ist nur ein einfacher String.

Es ist nicht empfehlenswert, komplette MEL-Skripte mit allen Feinheiten in einem XL-Programm zusammen zu bauen und zu übergeben. Dies würde die Übersicht einschränken und besitzt einige zusätzliche Fehlerquellen, etwa Sonderzeichen. Deshalb ist es bei weitem einfacher, ein MEL-Skript mit einem externen Editor zu entwickeln, im Skript-Verzeichnis abzulegen und aus XL heraus nur aufzurufen:

```
[MEL]
source meinskript.mel;
startprozedur(parameter);
```

Diese Zeilen müssen natürlich in dem String stehen, der als Parameter an die *execute*-Methode übergeben wird.

Somit umgeht man sämtliche Probleme und hält das XL-Programm möglichst schmal. Als Parameter kann der Anwender beliebige Informationen wählen, welche XL auslesen kann, wie Attribute bestimmter Knoten, interne Variablen oder den DAG-Path eines Knotens (die Methode *getDAGPath*() der Klasse *Node* stellt diese Möglichkeit). Zu beachten ist, dass alles in einen String umgewandelt werden muss, damit es als Parameter für die *execute*-Methode fungieren kann.

Nun ist es wichtig, dass der Anwender sich auch Ergebnisse seines MEL-Skripts ausgeben lassen kann. Hierfür existieren die Methoden

- `get-TYPE-MelResult()`

wobei der Rückgabetyp dem -TYPE- entspricht. Folgende Typen werden unterstützt: int, intArray, double, doubleArray und String. Sollte das MEL-Skript oder der Befehl eine Matrix zurückliefern, wird diese in ein doubleArray linearisiert. Hierbei muss der Anwender vor der Benutzung in XL prüfen, welchen Rückgabetyp sein Skript liefert, und eventuell entsprechende Maßnahmen ergreifen. Die Einschränkung, warum ein *StringArray* nicht unterstützt wird, liegt in der Maya-API. In diesem Fall muss der Anwender das Array in einen String mit Trennzeichen umwandeln und dieses dann in XL rückgängig machen. Eine andere Möglichkeit besteht darin, die einzelnen Ergebnisse indexweise zurück zu geben. Nach jeder Anwendung der Methode *executeMEL*() werden die Ergebnisvariablen überschrieben. Somit ist es notwendig, möglichst unmittelbar danach die Ergebnisse abzurufen.

Auf eine weitere Einschränkung muss an dieser Stelle eingegangen werden: Die Erzeugung von DAG-Knoten via MEL. Es gibt viele MEL-Befehle, die Knoten erzeugen (wie bereits erörtert: jede GUI-Aktion ist zugleich ein MEL-Skript). Dies sollte mit Vorsicht eingesetzt werden, da es passieren kann, dass XL durcheinander kommt. Prinzipiell meldet ein Callback bei Knotenerzeugung diese gleich in XL an, und ein sofortiger Zugriff ist möglich. Andererseits ist dies ein Eingriff in den DAG, und es ist vorher nicht festzustellen, in welchem internen Zustand des Compilers möglicherweise der DAG so ungünstig verändert wird, dass plötzlich die aktuell erfassten Daten durch den Compiler ungültig werden (z.B. Anzahl der Kinder o. ä.). Sicherer ist an dieser Stelle die Verwendung der Methoden:

- `void executeLate(String command, boolean echo)`
- `void executeLate(String command)`

welche das verspätete Ausführen realisieren (vgl. Kapitel 5.3). Allerdings ist in diesem Fall der Zugriff auf die Rückgabewerte nur dann möglich, wenn die get-Type-Result-Methoden mittels der Hilfsmethode

- `void invokeMethodLate(Class m, String methodName, Object[] arguments, Class[] argumentTypes)`

der Klasse *User* ausgeführt werden.

Die Anwendungsmöglichkeiten dieser Technik sind vielfältig. Im einfachsten Fall kann solch ein kleines Skript jede Menge XL-Code sparen. Das Beispiel Citygenerator im Kapitel 7.5 greift darauf zurück.

5.7. Nurbs

Die Nurbsfunktionalität von XL ist eingeschränkt auf Pfadextrusionen. Damit lässt sich im Gegensatz zu aneinander gereihten Fs „weiche“, interpolierte Geometrie erzeugen. Notwendig sind die Knotentypen *Circle*, *Curve* und *Vertex*, um Nurbsextrusionen verwenden zu können.

Gleich zu Beginn dieses Kapitels ein Stück Bespielcode, anhand dessen die Funktionalität von Nurbs sich am besten erklären lässt:

```
[XL]
module meristem;
…
Axiom ==> y:Circle(0.5) Curve(y) Vertex.(translateBy(0, 1.0, 0)) meristem;
meristem ==> Vertex.(translate(0, 0.5, 0));
```

In diesem Beispiel wird zunächst ein Kreis mit dem Radius 0.5 erzeugt. Dieser dient als Profil für die Nurbsextrusion. Der unmittelbar dahinter erzeugte Knoten *Curve* repräsentiert den Pfad, der in Maya erstellt wird. Um nun die Pfadextrusion zu steuern, müssen weitere Knoten, nämlich Vertices erzeugt werden. Das durch die Extrusion erzeugte Nurbsobjekt ist zunächst XL nicht als Knoten bekannt. Dennoch kann nach dem *Transform*- oder *Shape*-Knoten gesucht werden, sofern es notwendig ist. Diese Nurbsobjekte werden nach ihrer Erzeugung unter den *World*-Knoten gehängt.

Intern geht XL bei der Erzeugung eines *Vertices* die Hierarchie stromaufwärts entlang und sucht den nächsten *Vertex*- oder *Curve*-Knoten. Jeder *Vertex* führt als Information mit sich, zu welcher *Curve* er gehört. Trifft dieser neue *Vertex* nun auf einen anderen *Vertex*, meldet er sich bei der gleichen *Curve* an und wird in Maya zu der Kurve als realer *Vertex* hinzu gefügt. Trifft er auf eine *Curve*, speichert er seine Zugehörigkeit und wird als *Vertex* in Maya angefügt.

Allerdings birgt dieses Verfahren eine wichtige Einschränkung: Die „echten“ *Vertices*, also die Knotenpunkte der Nurbskurve in Maya, sind nicht mit den „virtuellen“ *Vertices* aus XL verbunden. Es ist also nicht möglich, den Verlauf der Kurve nach Erzeugung des Vertices zu verändern, indem XL einfach die Position der Vertex-Objekte verschiebt. Deshalb muss bei der Erzeugung des Knotens in XL sofort die Transformation auf den *Vertices* via `.(translate(x, y, z))` angewendet werden oder davor Rotations- bzw Bewegungsknoten eingebaut werden. Nachträgliche Änderungen der Rotations- bzw. Positionswerte haben dann keine Auswirkungen mehr auf den Verlauf der Kurve.

5.8. Keyframes

Die Erweiterung von XL auf die Möglichkeit des Setzens von Keyframes in Maya lässt XL in die Welt der Animation vorstoßen. Nun ist es möglich, gewisse dynamische Prozesse zunächst komplett zu berechnen, dabei entsprechende Keyframes zu setzen und später die Animation einfach abzuspielen. Ein Beispiel, welches dieses einsetzt, ist das „Game of Life“, das im Kapitel 7.2 genauer erklärt wird. Beim Einsatz von Keyframes aus XL heraus müssen einige Sachverhalte beachtet werden, welche bei „normaler“ XL-Programmierung kaum ins Gewicht fallen. Doch zunächst einige Grundlagen.

In der Klasse *User* befinden sich die notwendigen Methoden zum Setzen von Keyframes und der Zeit.

- `void setTime(int value)`
- `void setTimeLate(int value)`

sind zum Setzen der Zeit in Maya zuständig. Hierbei wird der Timeslider auf den entsprechenden Wert gesetzt. Alle darauf folgenden Keyframes der Methoden

- `void setKeyframe(Node m, String property)`
- `void setKeyframe(Node m, String property, double value)`
- `void setKeyframeLate(Node m, String property)`

- `void setKeyframeLate(Node m, String property, double value)`

werden dann an diesem Zeitpunkt gesetzt. Jede dieser Methoden erwartet als Parameter einen Knoten und ein Attribut. Das Attribut muss ein Einzelattribut sein und *double* als Datentyp haben (fast alle Attribute in Maya sind kompatibel zum Typ *double*, z.B. die Typen *enum* und *integer*). Bei den Methoden, die zusätzlich noch einen *double*-Parameter erwarten, wird das Attribut vor dem Setzen des Keyframes auf diesen Wert gesetzt.

- `void setKeyframe(Node m, String property, int time, double value)`
- `void setKeyframeLate(Node m, String property, int time, double value)`

Diese beiden Methoden ignorieren die aktuelle Zeit in Maya und setzen den Keyframe am gewünschten Zeitpunkt mit dem Wert des übergebenen Parameters „value“. In einigen Fällen erspart diese Methode zusätzliche Aufrufe der Methode *setTime*.

Die zu jeder Methode gehörende *Late*-Variante erleichtert dem Anwender das verspätete Ausführen, um Quasi-Parallelität zu erreichen. Die Art des gesetzten Keyframes aller Methoden hängt von der Einstellung der Standardkeyframes in Maya ab.

5.9. Weitere Methoden der Klasse User

- `void println (String text)`

Es ist wichtig für den Anwender, eine Möglichkeit zu haben, Strings an Maya weiterzuleiten und diese auszugeben. Insbesondere beim Debuggen ist es oftmals notwendig, Variableninhalte oder andere wichtige Informationen auszugeben.

Die übergebenen Strings können beliebiger Länge sein. Es gibt bei der Übergabe an C++ eine Einschränkung der Länge, doch der String wird dann aufgesplittet und in Maya nacheinander ausgegeben.

Eine wichtige Komponente bei der Simulation bzw. Modellierung realistischer Prozesse ist der Zufall. Ohne Zufall wäre es nicht möglich, eine Pflanze mit abweichenden Parametern wachsen zu lassen oder genetische Vorgänge zu simulieren. Überall ist der Zufall notwendig. Und auch wenn in der Natur oftmals eigentlich kein Zufall dafür sorgt, wie sich ein bestimmter Prozess verhält, kann man mit der richtigen Zufallsverteilung diesen dennoch simulieren. XL4Maya bietet zwei Methoden zur Erzeugung von gleich verteilten Zufallszahlen:

- `float random (float min, float max)`
- `int irandom (int min, int max)`

Beide Methoden erzeugen Zufallszahlen innerhalb eines bestimmten Intervalls, wobei *random Float*-Werte und *irandom Integer*-Werte erzeugt.

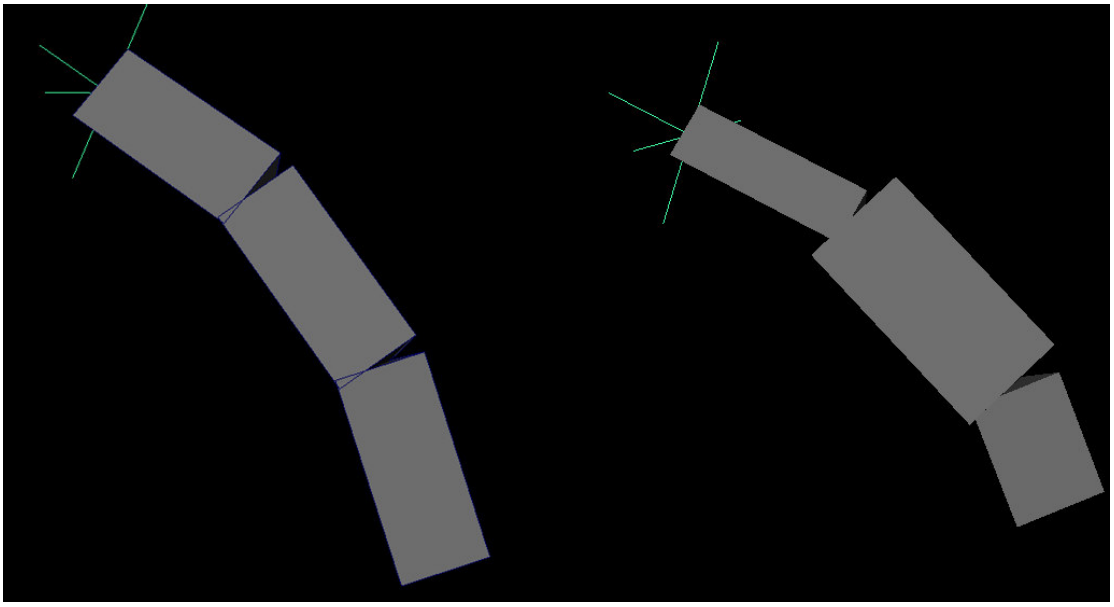


Abbildung 8: Eine Kette von F-Knoten ohne und mit Zufall

Abbildung 8 zeigt ein kleines Beispiel zur Anwendung von Zufall: Während die linke Kette ohne Zufall erzeugt wurde, besitzt die Rechte innerhalb eines Intervalls gewählte Parameter. In vielen Fällen ist bereits eine geringe Wirkung von Zufall ausreichend, um den Eindruck natürlicher Vorgänge zu erzeugen oder komplexe Vorgänge zu simulieren. Im zugehörigen Quellcode ist *run1* die Methode ohne und *run2* mit Zufallseinwirkung:

```
[XL]
module meristem;
public class zufall{
      public static void run1() [
            Axiom ==> meristem;
            meristem ==> RU(20) F(2, 0.5) meristem;
      ]

      public static void run2() [
            Axiom ==> meristem;
            meristem ==> RU(random(15.0, 25.0)) F(random(0.5, 3.0),
            random(0.2, 0.8)) meristem;
      ]
}
```

- `FinishIterator apply (final int n)`

Diese Methode implementiert das Ausführen von XL-Ableitungsmethoden aus anderen Methoden heraus und bietet die Möglichkeit, zu iterieren. Der Rückgabetyp muss vom Anwender nicht beachtet werden.

```
[XL]
      for (apply(1)) useNurbs();
```

Dieses Beispiel wendet die Methode useNurbs() einmalig an.

- `Point3d getClosestPoint(Transform node, Point3d point)`
- `Point3d getClosestPoint(Transform node,`
  `double x, double y, double z)`

Diese beiden Methoden sind in der Funktionalität äquivalent: Sie berechnen den nächstliegenden Punkt auf der Oberfläche von *node*, ausgehend von *point* bzw. dem Tripel (x, y, z). Sie liefern den berechneten Punkt als *Point3d* zurück.

Die beiden vorhergehenden Methoden liefern irgendeinen Punkt auf der Oberfläche eines Knotens zurück,

- `Point3d getRayIntersection(Point3d startPoint, double directionX, double directionY, double directionZ, Node objectToIntersect)`

ist hingegen etwas spezieller: Ausgehend von *startPoint* wird in Richtung (directionX, directionY, directionZ) ein Strahl ausgesandt und ein eventueller Schnittpunkt mit *objectToIntersect* errechnet. Sollte kein Schnittpunkt gefunden werden, wird *startPoint* zurückgegeben. Mit Hilfe dieser Methode kann der Anwender Objekte gezielt auf einer polygonalen Oberfläche verteilen.

Im Laufe der Entwicklung von Beispielen war besonders die geringe Geschwindigkeit der Ausführung von XL-Programmen auffällig. Bei kleineren Beispielen fällt das nicht allzu sehr ins Gewicht, allerdings steigt die Berechnungszeit mit der Baumtiefe des DAG dramatisch an. Um dem entgegenzuwirken, steht die Methode

- `void multiplyTransformNodes()`

zur Verfügung. Sämtliche L-System-Knoten außer F bestehen nur aus einem einzelnen *Transform*-Knoten. Wie bereits beschrieben, ist dies nichts weiter als eine Transformationsmatrix. Nun liegt es nahe, dass ein erheblicher Teil der Baumtiefe eingespart werden könnte, wenn diese Knoten ihre Transformation auf ihre Kinder weitergeben und selbst gelöscht werden. Genau dies tut *multiplyTransformNodes()*. Jeder gefundene Knoten vom Typ M, RV, RU, RL und RH wird entfernt. Dabei wird seine Transformationsmatrix ausgelesen und auf seine Kinder, welche selbst *Transform*-Knoten sind, multipliziert. Anschließend werden die Kinder in der Hierarchie an den eigenen Vater gehängt und der Knoten gelöscht.

Diese Technik sollte vom Anwender mit Umsicht eingesetzt werden, da einige Nebeneffekte entstehen können, besonders, wenn anwenderdefinierte Kanten im Einsatz sind, da das Löschen der Knoten ohne Rücksicht auf dieses geschieht. Ebenso ist zu beachten, wann diese Methode ausgeführt werden darf: Auf keinen Fall innerhalb einer Ableitungsregel. Am günstigsten ist es, diese Methode mit der Ausführung einer anderen XL-Methode zu kombinieren:

```
[XL]
      public static void letitrun () {
            for (apply(1))
                  useNurbs();
            for (apply(1))
                  multiply();
      }

      public static void multiply() {
                  multiplyTransformNodes();

      }
```

Dies ist ein Auszug aus dem Beispiel Nurbs des Kapitels 7.3. Die Methode *letitrun()* führt zunächst eine Methode mit normalen Ableitungsregeln aus, anschließend wird die Multiplikation durchgeführt.

Bei der Erstellung von XL-Programmen mit Verwendung dieser Methode sollte folgendes beachtet werden: Die Knoten sind nach der Anwendung nicht mehr vorhanden, somit können Suchmuster ihre Gültigkeit verlieren. Die Regeln müssen dem entsprechend gestaltet werden.

Nichtsdestotrotz ergibt sich eine massive Beschleunigung der Ableitungen, da die Baumtiefe wesentlich langsamer wächst.

6. Hinweise

6.1. Installation

XL4Maya wird über eine Setup-Routine installiert. Hierfür muss der Anwender die Datei „setup.exe“ starten. Über mehrere Fenster wird der Anwender durch den Installationsprozess geführt und muss nur den Anweisungen folgen. Die Setup-Routine liest bestimmte Informationen zur Unterstützung des Anwenders aus der Registry aus, und zwar den Maya-Installationsordner, die Position der jvm.dll und den Ordner der Maya-Einstellungen. Während des Ablaufes werden alle Informationen jedoch zusätzlich abgefragt und der Anwender kann diese korrigieren.

Setup erzeugt zwei Registriereinträge, welche zum einen auf die jvm.dll der vorhandenen Java-Installation zeigen und zum anderen auf die Position der XL4Maya.jar, welche alle Java-Dateien beinhaltet. Letztere ist deshalb notwendig, weil der Anwender mehrere Java-Versionen installiert haben kann und somit ein fester Bezug zur jvm.dll in der Registry existieren muss. Zudem kann es sein, dass Setup den Eintrag nicht korrekt ausliest und es somit das Plug-in zur Laufzeit auch nicht tun würde. Es muss also dem Anwender die Möglichkeit des Eingreifens gegeben werden, am besten während der Installation.

Da XL4Maya für Maya 7.0 und 8.0 zur Verfügung steht, kann es passieren, dass der Anwender versucht, die falsche Version zu installieren. In diesem Fall macht sich dies bei dem Versuch, das Plug-in in Maya zu starten, bemerkbar.

Ebenso ist es möglich, dass der Anwender beide Versionen von Maya installiert hat. In diesem Fall muss beim Setup geprüft werden, dass XL4Maya in die gewünschte Version kopiert wird. Es ist auch möglich, XL4Maya für beide Versionen zu installieren. Allerdings werden dafür keine neuen Registriereinträge gemacht, sondern nur bei der zweiten Installation die Einträge aktualisiert. Die XL4Maya.jar existiert somit zweimal und der Registriereintrag zeigt auf eine der beiden. Sollte der Anwender eine der beiden Maya-Versionen deinstallieren und das Plug-in funktioniert danach in der anderen Version nicht mehr, sollte es deinstalliert und erneut installiert werden.

Setup kopiert neben den oben genannten Einstellungen folgende Dateien ins System: „xl.mel“, „xlEditor.mel“ und „xlINI.mel“ werden in den Script-Ordner des Anwenders kopiert, „shelf_XL4Maya.mel“ in den Ordner „\prefs\shelves\“, und „XL4Maya.mll“ in „maya\bin\Plugins\“. Zusätzlich wird in den vom Anwender gewünschten Ordner die Onlinehilfe „XL4Maya.chm“, „XL4Maya.jar“, „uninstall.exe“ und ein Internet-Shortcut kopiert.

Neben diesen Dateien kann der Anwender während des Setup noch wählen, ob ein Startmenü-Eintrag erstellt wird. Dieser enthält dann Verweise auf den Internet-Shortcut, die Onlinehilfe und das Uninstall-Programm.

Eine illustrierte Beschreibung des Installationsprozesses befindet sich in der Onlinehilfe.

6.2. GUI

XL4Maya bietet eine komfortable und minimalistische Anwenderoberfläche, welche sich in Maya integriert und sich dabei nicht in den Vordergrund drängt. Nach dem Ausführen der MEL-Befehle:

```
[MEL]
source xl.mel;
runxl();
```

oder nach dem Klick auf den Button in der vom Setup-Programm erstellten Shelf „XL4Maya“ befinden sich auf der Oberfläche von Maya rechts unten einige Buttons zur Steuerung von XL4Maya. Diese Buttons passen sich je nach Situation in ihrer Beschriftung und Funktion an.

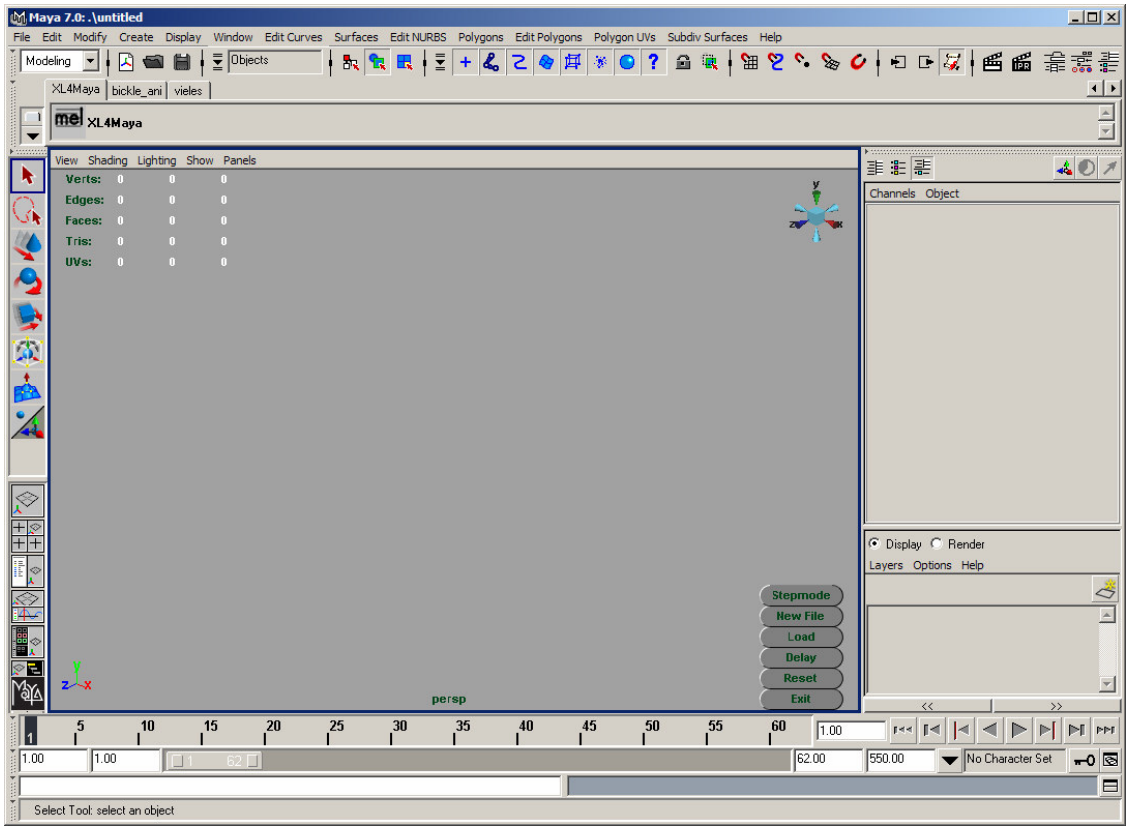


Abbildung 9: GUI nach dem Start von XL4Maya

Abbildung 9 zeigt die GUI, nachdem der Anwender auf den Shelf-Button XL4Maya geklickt hat. Es werden sechs Buttons im 3D-Fenster angezeigt.

Im Folgenden werden alle Buttons erklärt:

- `Stepmode`

schaltet zwischen dem *runmode* und dem *stepmode* um. Im *runmode* wird eine vom Anwender ausgewählte Methode iterativ ausgeführt, bis er auf „Esc“ drückt. Während der Ausführung wird links unten in Maya angezeigt, dass Maya gerade arbeitet und auf die Eingabe von „Esc“ reagiert. Allerdings kann dies erst wirksam werden, wenn die aktuelle Methode vollständig abgearbeitet wurde.

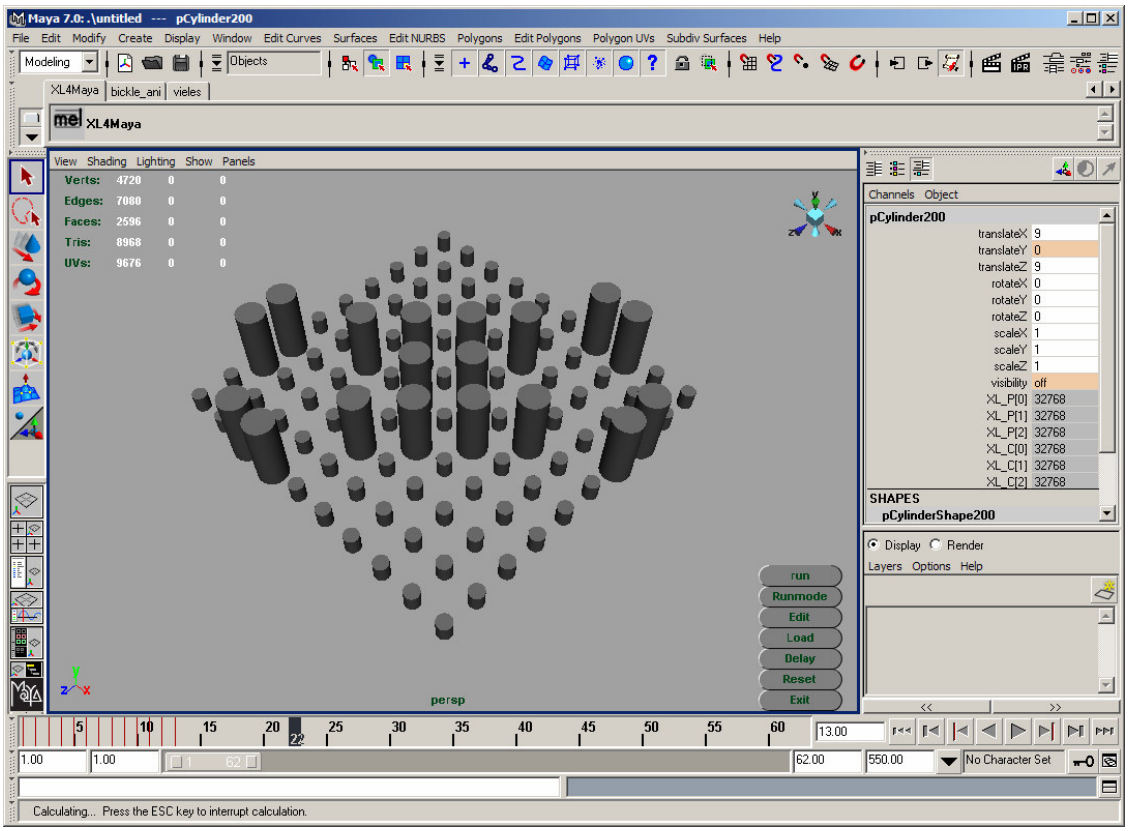


Abbildung 10: Das Game of Life während der Berechnung

Abbildung 10 zeigt das „Game of Life“ während der Berechnung. Links unten befindet sich der eben beschriebene Hinweis.

- `New File`

Wenn XL4Maya gestartet wird, kann über diesen Button eine neue Datei erstellt werden. Hierbei öffnet sich der Editor. Eine Beschreibung des Editors befindet sich weiter unten im Kapitel.

- `Load`

öffnet ein Fenster, in dem alle zuletzt geöffneten XL-Dateien angezeigt werden. Diese Daten befinden sich in der INI-Datei, welche sich im Maya-Ordner für die Anwendereinstellungen befindet.

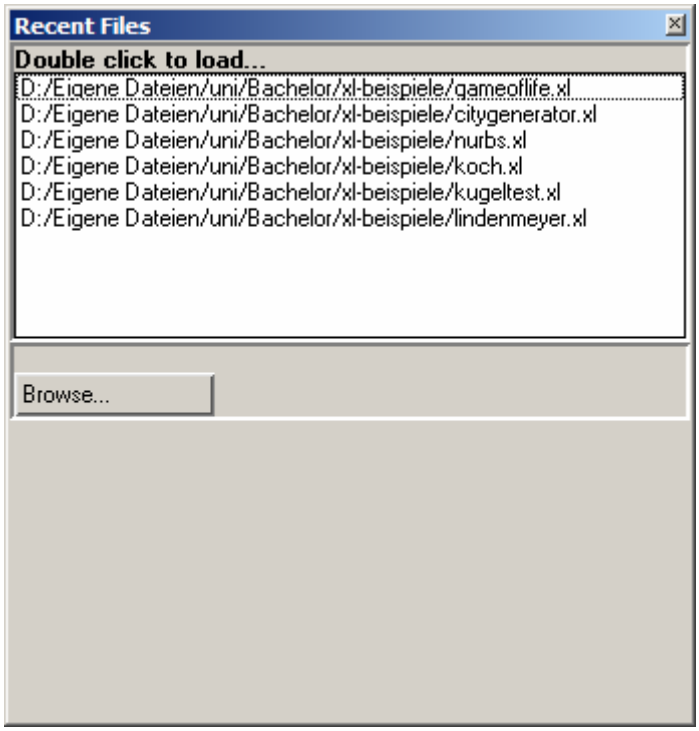


Abbildung 11: Menü „Recent Files“

Abbildung 11 zeigt das Menü. Mit einem Doppelklick auf eine Datei wird diese geladen. Sollte eine andere Datei geöffnet werden, öffnet der Button

- `Browse`

ein Fenster, mit dem andere XL-Dateien ausgewählt werden können.
Nach dem Laden einer XL-Datei verändert sich das Menü leicht.

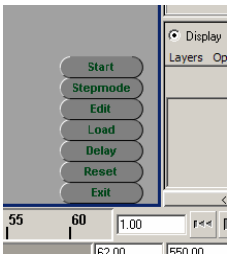


Abbildung 12: Hauptmenü nach dem Laden

Es wird der Button

- `Start`

angezeigt. Dieser startet den Compiler mit der aktuell geladenen Datei. Anschließend, im Falle des erfolgreichen Kompilierens, werden neue Buttons zur Ausführung der entsprechenden Methoden angezeigt und der Button *Start* ausgeblendet. Der Anwender kann anschließend den Compiler über einen Klick auf die gewünschte Methode anwenden. Zu beachten ist, dass eine im Code befindliche Methode mit dem Namen *init* automatisch nach dem Klick auf *Start* ausgeführt wird. Ebenso ersetzt der Button

- `Edit`

den Button *New File*. *Edit* öffnet den Editor mit der aktuell geladenen XL-Datei.

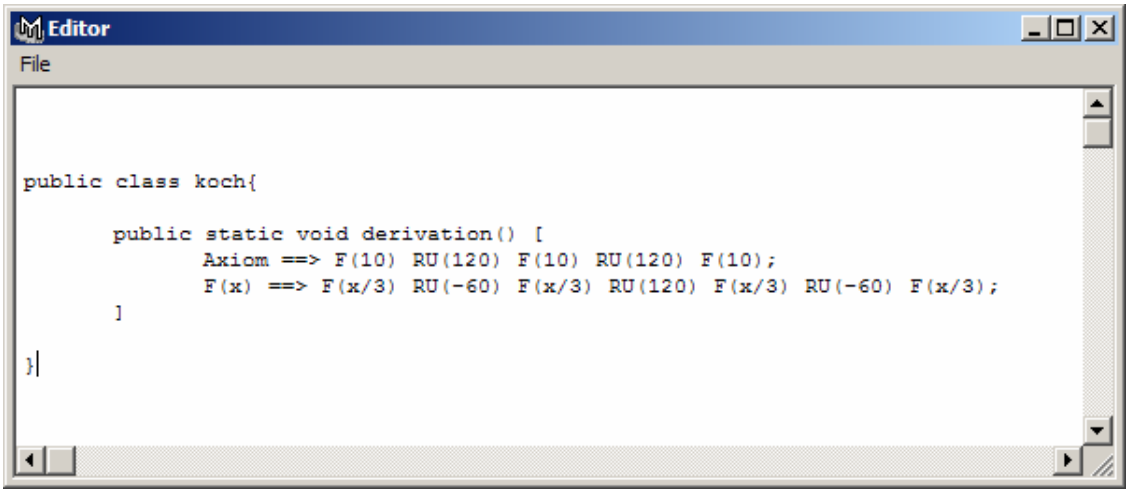


Abbildung 13: Der Editor

Die Abbildung 13 zeigt den Editor. In ihm können XL-Dateien verändert werden. Das Menü *File* hat mehrere Einträge:

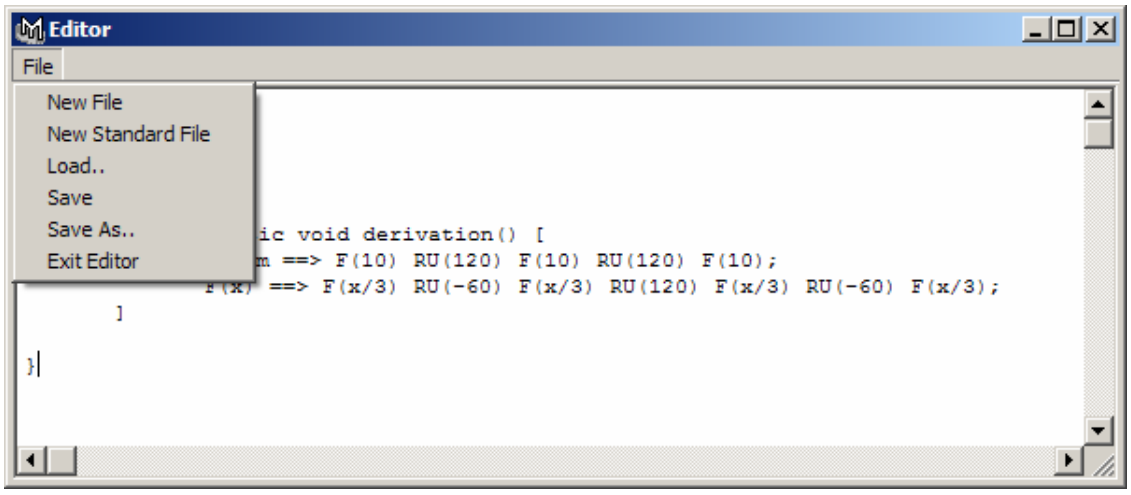


Abbildung 14: Einträge des Menüs File

- `New File`

ist äquivalent zu *New File* im Hauptmenü.

- `New Standard File`

erzeugt sofort ein XL-Gerüst mit einer Klasse und einer Methode. Dies erleichtert den Start für eine neue XL-Datei.

- `Load..`

öffnet das bereits zuvor beschriebene Fenster *Recent Files*. Die XL-Datei wird in dem Editor angezeigt. Zuvor wird darauf hingewiesen, dass die alte XL-Datei überschrieben wird. Der Anwender sollte zuvor speichern.

- `Save..`

Speichert die XL-Datei und führt ein GUI-Reset aus, das heißt eventuelle Methoden-Buttons werden gelöscht und der Start-Button wieder angezeigt.

Im Falle einer vom Anwender vorbereiteten Szene, die zu diesem Zeitpunkt eventuell schon von XL verändert wurde, sollte der Anwender noch zusätzlich ein *Reset* ausführen.
Falls die XL-Datei bis dahin noch nicht gespeichert wurde, wird ein „Save As..“ ausgeführt.

- `Save As..`

verhält sich äquivalent zu *Save*, nur dass ein Menü angezeigt wird, das den Anwender eine neuen Speicherort und Namen auswählen lässt.

- `Exit Editor`

schließt den Editor. Die XL-Datei bleibt im Speicher erhalten. Der Anwender wird gefragt, ob er die XL-Datei speichern möchte. Dies ist notwendig, da der Compiler die Datei von der Festplatte ausliest, nicht aus dem Editor.
Zurück zum Hauptmenü:

- `Delay`

öffnet ein Fenster, das den Anwender dazu auffordert, einen neuen Wert für das Delay im *runmode* festzulegen. Das *Delay* ist eine Wartezeit in ms, welche zwischen den Methodenausführungen eingehalten wird.

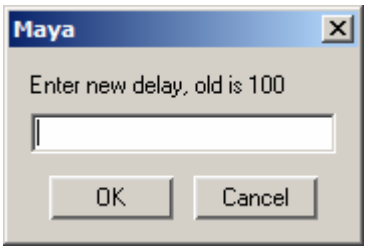


Abbildung 15: Delayänderung

In Abbildung 15 wird das Fenster abgebildet. Der Button *OK* setzt den Wert und schreibt ihn in die INI-Datei, damit er über den Neustart von Maya hinaus erhalten bleibt. *Cancel* bricht die Aktion ab. Das Fenster zeigt an, welchen Wert das *Delay* aktuell hat.

- `Reset`

setzt die GUI zurück und lädt die aktuelle Maya-Szene neu. Wenn bis dahin keine Maya-Szene gespeichert oder geöffnet wurde, wird eine neue Szene erstellt. Dies ist notwendig, damit der Zustand vor dem Anwenden des Compilers wieder hergestellt werden kann. Es ist zu empfehlen, nach jedem Speichern der XL-Datei ein *Reset* auszuführen und vor dem Anwenden des Compilers die Szene zu speichern, sofern diese vormodellierte Objekte enthält, die der Compiler verwendet oder verändert. Dies ist die beste Methode, um sicher zu stellen, dass der Compiler immer auf die gleiche Ausgangssituation zurückgreift.

- `Exit`

beendet XL4Maya. Hierbei wird die JVM nicht beendet, sondern nur die GUI entfernt. Sie kann über den Shelf-Button wieder neu gestartet werden, ohne dass Maya beendet werden muss.

6.3. Einschränkungen

XL4Maya unterliegt ein paar Einschränkungen, welche bei der Verwendung beachtet werden sollten. Zunächst sollte ein bestimmter Arbeitsfluss eingehalten werden, der erfahrungsgemäß am schnellsten gute Ergebnisse liefert und im Folgenden stichpunktartig beschrieben wird. Er gilt nur für XL-Dateien, bei denen eine bestimmte Szene in Maya vorbereitet werden muss:

1. Maya-Szene vorbereiten/anpassen
2. Diese Szene speichern.
3. XL-Datei schreiben/laden
4. Compiler auf Szene anwenden
5. Nur bei gewünschtem Ergebnis speichern, am besten unter einem anderen Namen, damit die Startszene für spätere Anpassungen erhalten bleibt. Falls das gewünschte Ergebnis nicht erreicht wurde, ohne zu speichern auf RESET drücken. Dies lädt die aktuelle Szene erneut.
6. Bei Schritt 1 fortfahren, bis gewünschtes Ergebnis erreicht ist.

Der Hintergrund dieses einfachen Arbeitsflusses ist, dass die in XL erzeugten L-System-Objekte nach einer Reinitialisierung des Compilers nicht mehr als F, RU oder andere Knoten erkannt werden. Da diese Klassifikation allein auf Java-Seite stattfindet, kann nach dem Neustart nicht mehr auf solche Objekte in Suchmustern zurückgegriffen werden.

Zwischenspeichern von Ergebnissen ist durchaus möglich und sinnvoll, besonders bei längeren Berechnungen. Dies sollte aber in einer anderen Datei geschehen. Hierbei ist zu beachten, dass nach einem RESET dann diese Szene geladen wird, nicht die ursprüngliche. Diese muss dann von Hand geladen werden.

Nun gibt es aber auch XL-Programme, welche nicht auf eine spezielle Maya-Szene zurückgreifen. Der Unterschied zum oben beschriebenen Arbeitsfluss besteht darin, dass keine Maya-Szene angepasst werden muss. Es genügt, eine neue Szene zu öffnen. XL erstellt bei der Initalisierung automatisch ein Axiom und kann dieses in den XL-Programmen benutzen. In diesem Fall kann XL solange angewendet werden, bis ein zufrieden stellendes Ergebnis erreicht wurde. Dann kann die Szene unter einem beliebigen Namen gespeichert werden. Aber auch hier ist zu beachten: Auf XL-Knoten kann nach einer Neuinitialisierung des Compilers nicht zurückgegriffen werden.

7. Beispiele

XL ist für Anwender mit wenig Programmiererfahrung recht komplex, da durch die Erweiterung von Java die Möglichkeiten stark erweitert wurden. Um diesen Anwendern den Einstieg etwas zu erleichtern, befinden sich auf den folgenden Seiten fünf Beispiele geringerer und höherer Komplexität. Alle verwenden unterschiedliche Konzepte, anhand deren die Methodiken und Techniken deutlich werden sollen. Sie demonstrieren durchaus einen Teil der Mächtigkeit von XL, doch ist diese hier längst noch nicht ausgereizt. Die Quellcodes werden ausschnittsweise behandelt. Im Anhang befinden sich die vollständigen Quellcodes. Die verwendeten

Maya-Dateien befinden sich zusammen mit der HTML-Referenz und den Beispiel-XL-Dateien auf der Installations-CD.

7.1. Kochkurve

Die Kochkurve wurde bereits im Kapitel 3.1 erwähnt. An dieser Stelle sei noch einmal der Quellcode und eine Abbildung hinzugefügt:

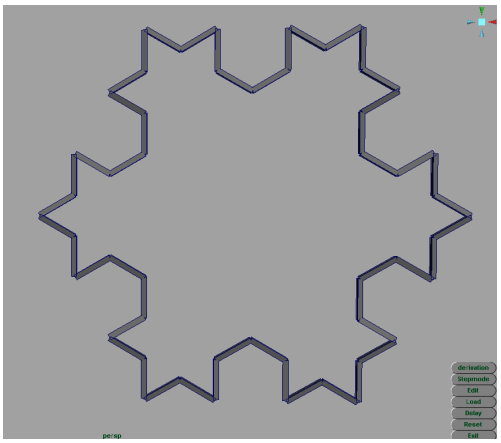


Abbildung 16: Kochkurve

Abbildung 16 zeigt die Kochkurve, welche auch „Schneeflockenkurve“ oder „Kochsche Insel“ genannt wird. Der XL-Code ist recht einfach:

```
[XL]
public static void derivation() [
      Axiom ==> F(10) RU(120) F(10) RU(120) F(10);
      F(x) ==> F(x/3) RU(-60) F(x/3) RU(120) F(x/3) RU(-60) F(x/3);
]
```

Der Code besteht ausschließlich aus Drehungen und F-Knoten. Eine Besonderheit ist F(x), hierbei ist in der Klasse F eine Methode implementiert, welche bei Übergabe eines Parameters (an dieser Stelle x) die Länge zurückgibt. Dieser Wert kann anschließend verwendet werden, Veränderungen dieses Wertes haben keinen rückwirkenden Einfluss auf die Länge des F-Knotens.

7.2. Game of Life

Das „Game of Life“ wurde von John Conway, einem Mathematiker der Cambridge-Universität, erfunden. Es ist ein zellulärer Automat und simuliert das Sterben, Leben und Vervielfältigen von Zellen. Es besteht aus einer Sammlung regelmäßig angeordneter Zellen mit zwei Zuständen: Man könnte dies als lebendig und tot, angeregt und nicht angeregt oder schlicht 1 und 0 interpretieren. Dies kann durch farbliche Unterscheidung verdeutlicht werden. Ausgehend von den Startkonditionen entstehen während des Ablaufs interessante neue Muster. Die Regeln für die Entwicklung, also der Zustandsübergänge für jede Zelle im Zustand 1, sind folgende[1]:

[1] Entnommen aus: [20].

Jede Zelle mit nur einem oder keinem Nachbar stirbt an Einsamkeit,
jede Zelle mit vier oder mehr Nachbarn stirbt an Überpopulation,
jede Zelle mit zwei oder drei Nachbarn überlebt.

Für Zellen im Zustand 0 gilt:
Jede Zelle mit genau drei Nachbarn wird angeregt.

XL4Maya liegt eine Implementierung des „Game of Life“ vor[1] und demonstriert einige der Möglichkeiten. Unter anderen werden Kantenbeziehungen und Keyframes eingesetzt. Grundlage war die Implementierung für GroImp von Ole Kniemeyer. Abbildung 17 zeigt das „Game Of Life“ nach der Initialisierung in Maya. Die großen Cylinder stellen angeregte Zellen dar, die kleinen tote.

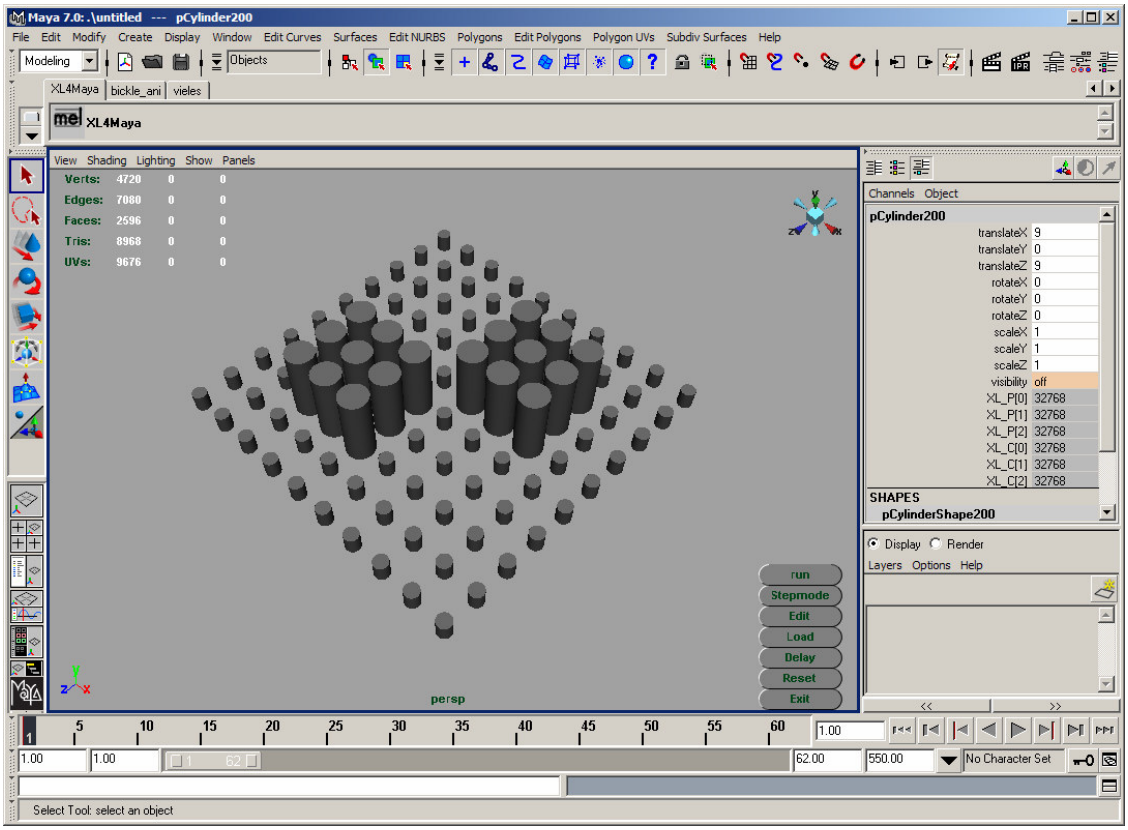


Abbildung 17: Das Game of Life

Das Beispiel besteht aus zwei öffentlichen und einer privaten Methode. Hinzu kommt die von *Cylinder* ableitende Knotenklasse *Cell*. Sie enthält alle notwendigen Methoden zum Setzen der Zustände einer Zelle. Zunächst muss die Methode *init* erklärt werden. Der zusammenhängende Quellcode befindet sich in der Beispieldatei *gameoflife.xl*.

Diese Methode ist für die Initialisierung der schachbrettartigen Anordnung der Zellen zuständig. Hierfür werden 9x9 viele Cylinder erzeugt und angeordnet.

[1] Vgl: [21].

```
[XL]
public static void init () {
	[

	Axiom ==>> for (int i : (0:9)) for (int j : (0:9))
	( [Cell(i, j, 0, (i >= 2) && (i < 8) && (j >= 2) && (j < 8)
	&& ((i < 5) ^ (j < 5)))]);
	]
		for (apply(1)) init_neighborhood ();

}
```

Dieser etwas knapp geschriebene Code erzeugt genau dieses. Hierbei wird zugleich das Grundmuster festgelegt, also bestimmt, welche Zellen zu Beginn im Zustand 1 sind. Anschließend wird die Methode *init_neighborhood* aufgerufen.

```
[XL]
private static void init_neighborhood () [
	c1:Cell, c2:Cell, ((c1 != c2) && (c1.distanceLinf (c2) < 1.1))
		==>> c1 -neighbour-> c2;
]
```

In ihr werden die Abstände zwischen jedem Zellenpaar berechnet und ggf. eine Nachbarschafts-Kante dazwischen gezogen. Der Vorteil dieser Methode ist, dass im Verlauf des „Game of Life“ nicht bei jedem Schritt diese Abstände neu berechnet werden müssen, sondern von Beginn an feststehen. Nur auf Grund dieser Vorberechnung wurde der eigentliche Ablauf um mindestens den Faktor zwei beschleunigt.

Nun ist die Szene initialisiert und dem Anwender steht es frei, den Ablauf zu starten. Doch, bevor die Run-Methode erklärt wird, noch ein Blick in die *Cell*-Klasse:

```
[XL]
public class Cell extends Cylinder{

	public int state;

	public Cell(int posx, int posy, int posz, boolean state){
		super();
		this.translateBy((double) posx, 0, (double) posy, true);
		this.setValue("radius", 0.4);

		if (state)
		{
			setKeyframeLate(this, "visibility",
					gameoflife.time+1, 1.0);
			this.state = 1;
		} else
		{
		setKeyframeLate(this, "visibility",
				gameoflife.time+1, 0.0);
			this.state = 0;
		}
	}

	public int getstate(){
		return state;
```

```
	}

	public void setlatestate(int state){
		Object[] arguments = {new Integer(state)};
		Class[] types = new Class[]{Integer.TYPE};
		invokeXLClassMethodLate(((Cell) this), "setstate",
							arguments, types);
	}
	public void setstate(Integer st){
		int state = st.intValue();
		this.state = state;
		if (state == 1){
			setKeyframe(this, "visibility",
					gameoflife.time+1, 1.0);
			}
		if (state == 0){
			setKeyframe(this, "visibility",
					gameoflife.time+1, 0.0);
			}
	}
}
```

Im Konstruktor von *Cell* wird die Position übergeben, an der die Zelle entstehen soll, und deren Anfangszustand. Je nach übergebenen Wert wird über

```
setKeyframeLate(this, "visibility", gameoflife.time+1, 1.0)
```

bereits ein entsprechender Keyframe gesetzt. Die Methode

- `int getstate()`

liefert den aktuellen Zustand der Cell zurück,

- `void setstate (int state)`

führt den eigentlichen Zustandswechsel aus, d.h. es wird ein Keyframe im nächsten Frame mit entsprechendem Sichtbarkeitswert der Variable *visibility* gesetzt.

Es ist wichtig, dass der Zustandswechsel verspätet ausgeführt wird, da sonst die darauf folgenden Berechnungen auf einen falschen Ausgangszustand zugreifen würden. Dazu ist zu wissen, dass sämtliche Regeln quasi-parallel ausgeführt werden. D.h. sie werden zwar sequenziell abgearbeitet, müssen aber alle auf den gleichen Zustand zugreifen, damit ihre Grundlagen, also die Ergebnisse der Parameterzugriffe, gleich sind. Die Veränderungen werden, wie bereits im Kapitel 5.3 erläutert, auf einen Stack gepackt und erst nach der Vollendung aller Regeln ausgeführt. Hierfür ist die Methode:

- `void setlatestate(int state)`

zuständig.

Nun können wir uns der Methode *run* zuwenden:

```
[XL]
public static void run ()
[
	{
		timeset();
	}
	x:Cell ::> {setKeyframe(x, "translateY");
	setKeyframe(x, "visibility");}
```

```
      x:Cell, (x.getstate() == 1),
      (!(sum ((* x -neighbour-> Cell *).getstate()) in (2:3))) ::>
      (x.setlatestate(0));


      x:Cell, (x.getstate() == 0),
      (sum ((* x -neighbour-> Cell *).getstate()) == 3) ::>
      (x.setlatestate(1));
]
```

Die Methode

- `void timeset()`

setzt sowohl die interne, als auch die echte Zeit in Maya, d.h. die Position des Time-Sliders. An dieser Stelle können dann die entsprechenden Keyframes erzeugt werden.

Zunächst setzt die erste der drei Regeln für jede Zelle einen Keyframe. Die zweite Regel sucht alle Zellen mit Zustand 1 und berechnet die Anzahl der Nachbarn im Zustand 0. Diese Berechnung ist zugleich eine Bedingung, denn sollte das Ergebnis (also die Summe über alle Zustände aller Nachbarn) 2 oder 3 sein, wird, wie es in den Regeln definiert ist, der Zustand der aktuellen Zelle von 1 auf 0 gewechselt. Dazu wird die bereits oben beschriebene Methode *setlatestate* der entsprechenden Zelleninstanz aufgerufen.

Ist im Gegenzug die Summe aller angeregten Nachbarn 3 und der Zustand dieser Zelle 0, wird in den Zustand 1 gewechselt.

7.3. Nurbs

Dies ist ein eher abstraktes Beispiel, welches folgende Techniken demonstrieren soll:

- der Einsatz von Nurbs
- RV und dessen Schnittpunktberechnung

Ausgangssituation ist die Szene „kopfbeispiel.mb“. Sie enthält einen polygonal modellierten Kopf, welches als Kollisionsobjekt dient und das Wachstum steuert.

An dieser Stelle macht es keinen Sinn, den gesamten Quellcode zu erklären. Deshalb zunächst eine Einführung in die Funktionsweise.

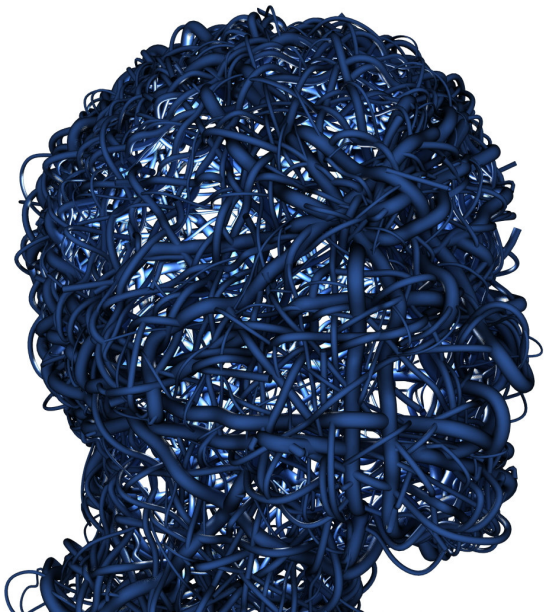


Abbildung 18: Entwicklung von Nurbs nach mehreren Stunden Laufzeit

Das in Abbildung 18 abgebildete Geflecht besteht zunächst aus einem „Hauptstrang“ und vielen „Nebensträngen“. In der Methode *useNurbs* wird nach dem Aufruf der Methode

- `Node getHead()`

welche ein Objekt mit dem Namen „ziel“ sucht und zurückgibt, über Zufall entschieden, ob an der aktuellen Position im Hauptstrang ein Nebenstrang gebildet wird. Das geschieht über

```
[XL]
Meristem ==>
if (probability(0.3))
```

An dieser Stelle stehen nachfolgend im IF-Zweig die rechte Seite mit Bildung eines Nebenstranges und im ELSE-Zweig jene ohne. Der Nebenstrang beginnt mit dem module *newZweig* und bekommt, bestimmt über den Zufall, übergeben, wieviele nachfolgende Ableitungen dieser Nebenstrang wachsen soll. In der Variable

```
[XL]
public static int count = 50;
```

deklariert am Anfang der Klasse *nurbs* legt die Obergrenze dafür fest. Mit der Entscheidung, einen Nebenstrang wachsen zu lassen, wird zunächst ein neuer Beginn für eine Nurbsextrusion festgelegt. Hierfür wird nach einer anfänglichen zufälligen Drehung ein *Circle* und eine *Curve* erzeugt.

Sowohl Haupt- als auch Nebenstrang verwenden den Knoten RV, um sich auszurichten. Hierbei wird der Knoten h, zu Beginn der Methode *useNurbs* über *getHead* bestimmt, als Zielknoten der Schnittpunktberechnung übergeben. Mit jeder Ableitung wird jedes Modul *newZweig* bzw. *Meristem* ersetzt und die Nurbscurve samt Extrusion verlängert.

Neben diesen Standardtechniken sind besonders die Techniken zur Beschleunigung der Ableitungen erwähnenswert. Zunächst wurde mit der Methode *getHead* vermieden, in jeder Regel nach dem Zielobjekt zu suchen. Dieses wird einmalig nach dem Betreten der Methode *useNurbs* durchgeführt, anschließend steht das Zielobjekt als Knoten zur Verfügung. Ebenso ist in der Methode *letitrun* eine wesentliche Beschleunigung erreicht worden. Indem dort zunächst *useNurbs* ausgeführt wird, aber nachfolgend gleich eine Multiplikation über *multiply*, steigt die Baumtiefe mit den erzeugten Knoten wesentlich langsamer und mehr Ableitungen können in der gleichen Zeit ausgeführt werden.

Diese Technik hatte an dieser Stelle besonders Sinn, da das Wachstum nur über den letzten Knoten in der Hierarchie, nämlich *newZweig* und *Meristem* stattfindet. Sämtliche Knoten davor sind für das weitere Wachstum nicht von Interesse, es ist nur wichtig, dass deren Position und Drehung beibehalten wird.

Dieses Beispiel lädt dazu ein, die Intervallgrenzen der *random*-Methoden zu verändern und deren Wirkung zu betrachten, ebenso erläutert es die Funktionsweise von Nurbsextrusionen.

7.4. Bush

Dieses Beispiel ist eine Umsetzung der Implementierung für GroImp von Ole Kniemeyer. Es mussten nur einige wenige Anpassungen vorgenommen werden.

Abbildung 19: Bush gerendert

Ausgangspunkt ist die Szene „bush.mb“. Darin befinden sich die Materialien „blattSG“ und „stengelSG“, damit diese den Blättern und Stängeln über *setShader* zugewiesen werden können. Diese Zuweisung findet in den Methoden

- `Node myF(double length)`
- `Node myLeaf(double size)`

statt, was zu einer weiteren Besonderheit führt: Diese Methoden haben einen Knoten als Rückgabetyp. Dieser Knoten wird über *new* erzeugt und zugleich wird

die Methode *setShader* angewendet. Nun kann die Methode *myLeaf* bzw. *myF* wie ein Knoten in den Ableitungsregeln behandelt werden.

Zu beachten ist, dass die Methode *init* automatisch beim Start des Compilers ausgeführt wird.

Abbildung 19 zeigt einen gerenderten Bush.

7.5. Citygenerator

Der Citygenerator ist das komplexeste aller Beispiele. Es verwendet die Techniken der Schnittpunktberechnung, Anwendung von MEL und der Verteilung von vormodellierten Objekten. Die Startszene „citygenerator.mb“ enthält eine polygonale Oberfläche, welche als Untergrund dient, drei verschiedene Stockwerkobjekte und ein Dachobjekt. Die Stockwerke unterscheiden sich nicht, sie dienen nur zur Demonstration der Technik des Duplizierens.

Abbildung 20 zeigt die bereits vorhandenen Objekte. Sie könnten durch den Anwender beliebig erweitert werden, d.h. mehr Polygone oder mehr Unterobjekte bekommen. Die Oberfläche ist eine simple *Plane*, welche mit dem *Sculpt*-Tool von Maya so bearbeitet wurde, dass sie wie eine hügelige Landschaft wirkt.

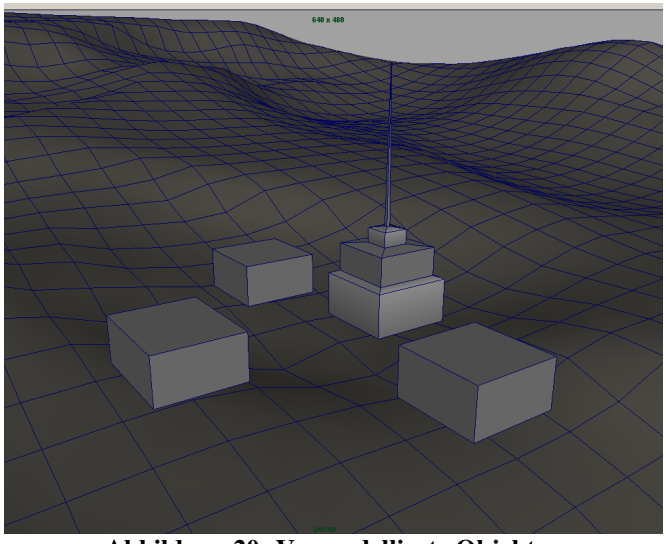


Abbildung 20: Vormodellierte Objekte

Zusätzlich zu diesen Objekten befindet sich noch ein verstecktes Objekt in der Szene, nämlich „testbox“. Dies dient als Kollisionsobjekt und schränkt das Wachstum der Stadt auf einen Bereich ein. Abbildung 21 zeigt dieses Objekt.

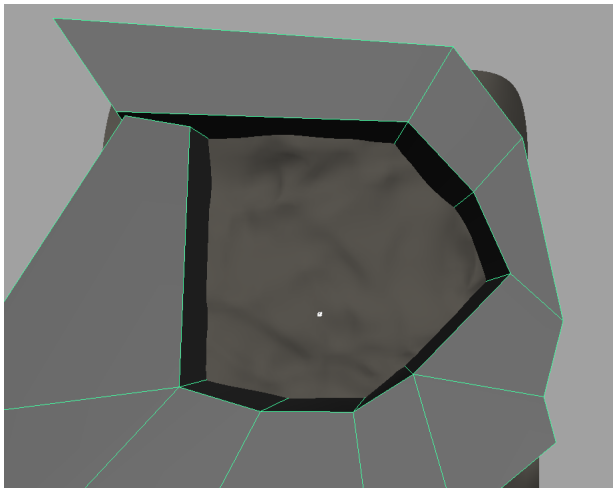


Abbildung 21: Das Kollisionsobjekt

Zunächst sei das Grundprinzip des Generators erläutert. Nach der Ausführung der Methode *init* (automatisch durch den Compiler bei einem Klick auf Start) und anschließender iterativer Ausführung der Methode *grow* beginnt das Wachstum zunächst entlang eines Hauptastes, welcher sich parallel zur Z-Achse in Maya befindet. Mit jedem Schritt werden in regelmäßigen Abständen Kreuzungen und Grundstücke erstellt, welche nichts weiter als Module sind und markieren, an welcher Stelle das Wachstum fortgesetzt wird (Kreuzung) und wo ein Gebäude entstehen wird (Grundstück). Mit jedem Grundstück der Hauptachse wird eine Nebenachse gebildet, welche sich im rechten Winkel zur Hauptachse in beide Richtungen ausbreitet. Hierbei wird ein RV-Knoten verwendet, damit die Nebenachsen einen leichten „Knick“ aufweisen und nicht schnurgerade verlaufen.

Vor der Erstellung einer Kreuzung wird getestet, ob diese sich im Kollisionsobjekt befindet, falls ja, wird sie nicht gebaut und das Wachstum hält an dieser Stelle an.

Die Grundstücke dienen zur Erzeugung der Gebäude. Zunächst wird ein Grundstück durch ein durch Zufall bestimmtes Stockwerk ersetzt und zugleich an dessen Spitze neu erzeugt. Hierfür wurden die Pivots der vormodellieren Objekte angepasst: Der *scale*-Pivot befindet sich auf der Unterseite und der *rotate*-Pivot auf der Oberseite des Objektes. Mit der Neuerzeugung des Grundstückes nach dem Duplizieren eines Stockwerkes wird dieses Grundstück in der Hierarchie unter dem Stockwerk einsortiert und somit am *scale*-Pivot platziert. Im nächsten Ersetzungschritt wird dieses Grundstück erneut gefunden, wieder ein Stockwerk platziert und dabei erneut ein neues Grundstück auf der Oberseite des Stockwerks platziert.

Nun bestimmt der Zufall noch, ob und wie lange ein Gebäude gebaut wird, ob es ein Dach bekommt oder ohne ein solches das Wachstum für das Gebäude eingestellt wird. Dies wird erreicht, indem einfach kein neues Grundstück auf der Oberseite des letzten Stockwerkes erzeugt wird. Abbildung 22 zeigt die Entwicklung der Stadt von oben betrachtet.

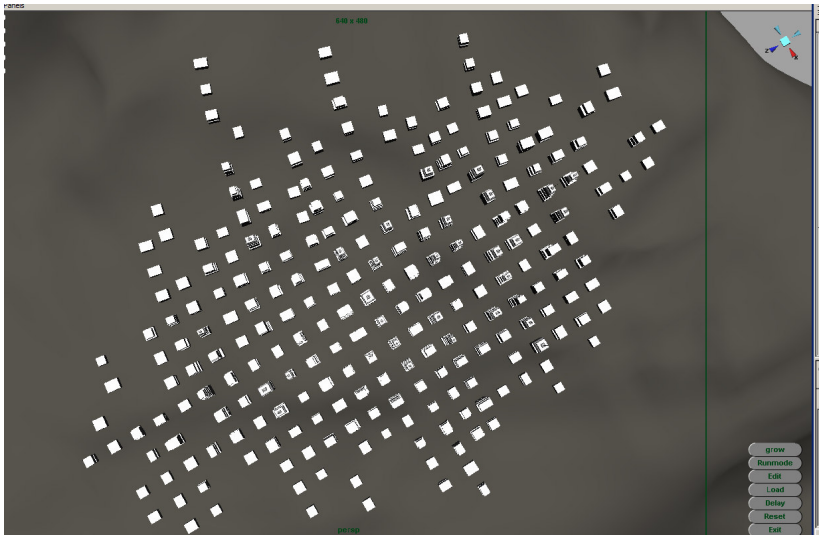


Abbildung 22: Struktur von oben

Jedem Stockwerk wird ein Material zugewiesen, welches Fensterbeleuchtung hinzufügt. Seit Beginn der Entwicklung dieses Beispieles stand fest, dass es kaum möglich ist, jedem Stockwerk ein anderes Material zuzuweisen, da dies hunderte verschiedene Materialien bedeutet hätte. Wäre im Nachhinein eine kleine Korrektur notwendig gewesen, hätten alle Materialien angepasst werden müssen. Hierfür musste über MEL ein Skript ausgeführt werden, welches speziell für diesen Zweck verfasst wurde. Es erzeugt zu dem einen Stockwerkmaterial, welches die Grundeinstellungen enthält und alle Parameter setzt, einen so genannten *switch*-Knoten. Dies ist ein DG-Knoten, welcher einen Parameter eines Materials als Input erhält und speziell für ein Objekt verändert. Somit ist es möglich, alle Objekte, die zwar das gleiche Material haben, aber nur einen Parameter variiert haben sollen, mit diesem *switch*-Knoten zu verbinden. In diesem Fall werden von Stockwerk zu Stockwerk die UV-Parameter, also die Texturkoordinaten durch einen weiteren Knoten variiert und somit der Eindruck erweckt, dass bei jedem Stockwerk andere Fenster beleuchtet sind und auch deren Größe unterschiedlich ist. Abbildung 23 zeigt die Stadt nach einigen Entwicklungsschritten, Abbildung 24 ist dessen gerenderte Variante.

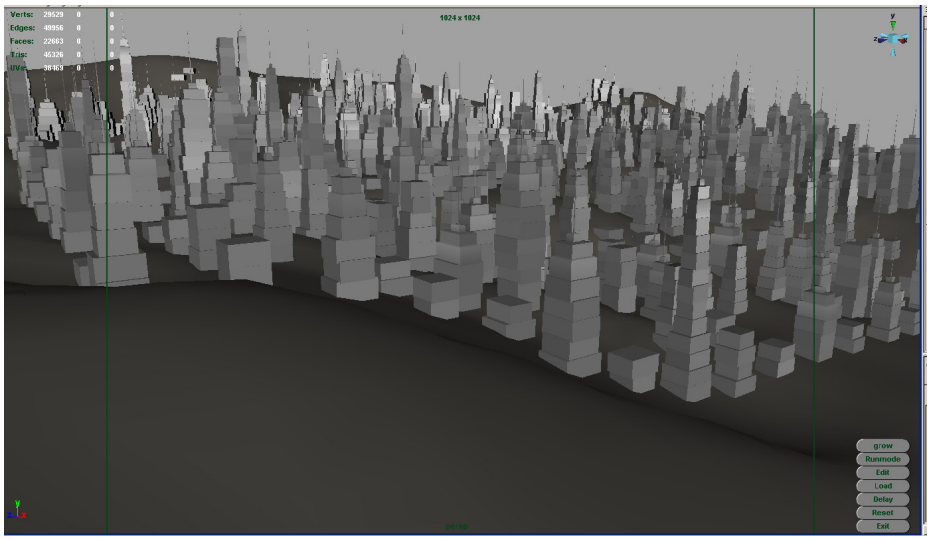


Abbildung 23: Ansicht der Stadt in Maya

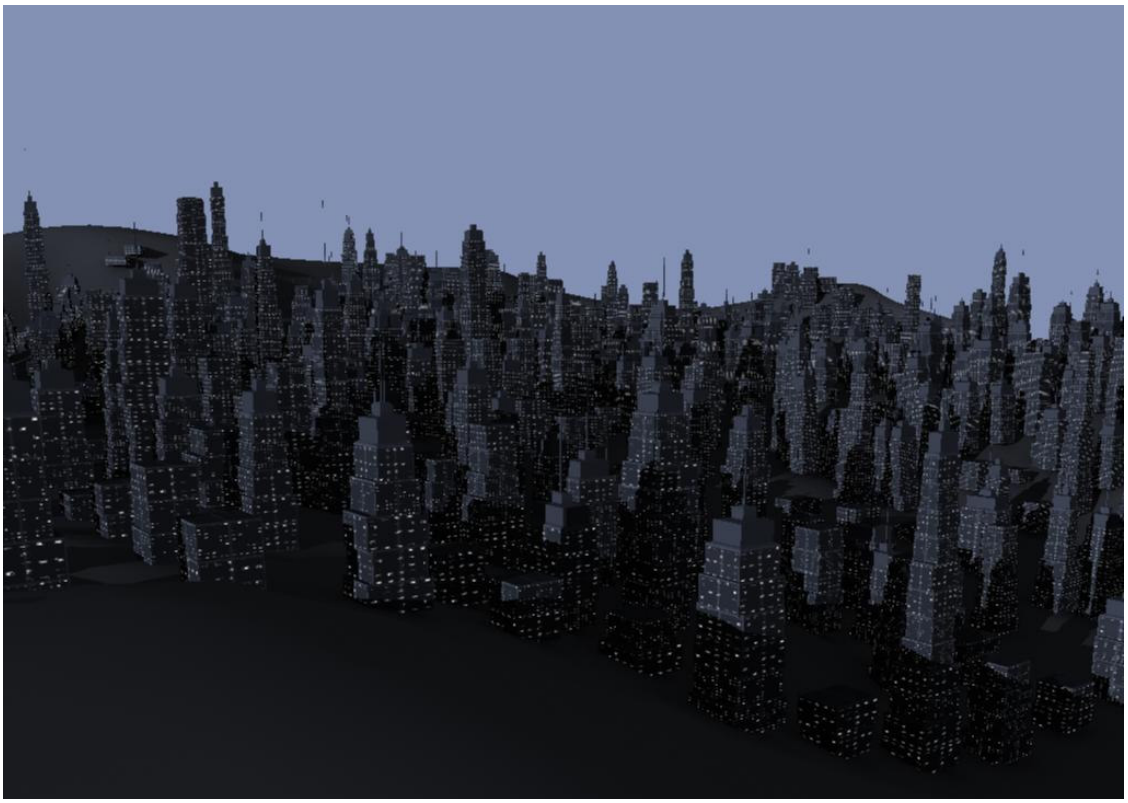


Abbildung 24: Gerenderte Stadt

Zuletzt ein paar Erläuterungen zu den eingesetzten Techniken.

- `Node getGround()`
- `Node getTestBox()`

dienen zur Beschleunigung der Suche nach diesen Objekten. So werden sie nur zu Beginn einer Regel gesucht und dann immer wieder verwendet, ohne dass neue Suchen stattfinden müssen.

- `void init()`

wird durch den Compiler automatisch nach dem Klick auf Start ausgeführt. Sie sorgt dafür, dass das Zusatzskript geladen und das Wachstum begonnen wird.
Das eigentliche Wachstum findet in der Methode

- `void grow()`

statt. Zunächst wird jede Hauptkreuzung (die Kreuzungen der Hauptachse) gesucht und jeweils um Grundstücke und Nebenkreuzung erweitert. Die Nebenkreuzungen sind für das Wachstum der Nebenachsen zuständig. Die hier eingesetzten M-Knoten suchen einen Punkt auf der Oberfläche von *pPlane1*, und dies in zwei Richtungen (nach oben und unten), da es ja sein kann, dass ein Grundstück unterhalb der Oberfläche liegt.

In der zweiten Regel werden alle Nebenkreuzungen gesucht und durch ein Wachstum zur Seite ersetzt. Auch hier kommen die M-Knoten zum Einsatz. Die Wahrscheinlichkeit, dass ein Wachstum zur Seite fortgesetzt wird, beträgt 50 Prozent. Somit sind die Nebenachsen unregelmäßig in ihrer Ausbreitung, was realistischer wirkt.

In der letzten Regel werden alle Grundstücke gesucht und über Zufall entschieden, welches Stockwerk dupliziert wird und an die Position verschoben wird. Zugleich wird über das Ende des Wachstums eines Gebäudes entschieden. Auf jedes neu erzeugte Stockwerk wird die Methode

- `void setShaderAndFreeze(Transform t)`

angewendet, mit dem Stockwerk als Parameter. Dort wird eine Instanz der Klasse *theMEL* erzeugt und an dessen Konstruktor

- `theMEL(Node t)`

der Knoten übergeben. In der Klasse *theMEL* befindet sich die Methode

- `void doCommand(Node t)`

Der Konstruktor der Klasse packt über

```
[XL]
invokeXLClassMethodLate(this, "doCommand", new Object[]{t}, new
Class[]{Node.class});
```

die Methode auf den Stack für das verspätete Ausführen. Erst in der Methode *doCommand* wird das MEL-Skript wirklich ausgeführt. Das verspätete Ausführen ist deshalb wichtig, da das Skript einen DAG-Pfad des Objektes braucht. Doch dieser ändert sich während der Ausführung einer Regel ständig, da nämlich unentwegt Kantenoperationen durchgeführt werden. Es ist im Allgemeinen empfehlenswert, MEL-Skripte verspätet auszuführen.

8. Fazit

Insgesamt ist zu bemerken, dass diese Arbeit die ersten Schritte zu einer vollständigen Anbindung von XL an Maya ist. Bei der Implementierung wurde darauf geachtet, möglichst viele Funktionen von vorn herein zur Verfügung zu stellen, um der Software eine hohe Funktionalität zu verleihen. Dabei konnte nicht auf alle Aspekte gleichwertig eingegangen werden. Der Grund dafür lag bereits in der frühen Phase der Entwicklung. Die unerwartet hohe Komplexität der Kommunikation zwischen Java und C++ (siehe Pointer-Problem, Kapitel 4.1.4) und viele kleine und große Probleme z.B. bei der Erzeugung der Java-Virtual-Machine oder der Implementierung von Properties (Vgl. Kapitel 5.4) sorgten für viele Verzögerungen. Insbesondere der Aspekt der Ausführungsgeschwindigkeit ist in dieser Arbeit ein relativ unangetastetes Thema. Zwar wurde während der Implementierung bereits auf gewisse Geschwindigkeitsvorteile geachtet, dennoch besteht an dieser Stelle noch Überarbeitungsbedarf. In vielen Fällen war keine optimale Lösung zu finden, jedenfalls nicht in dem gegebenen Zeitrahmen. Somit musste zunächst so implementiert werden, dass es überhaupt funktioniert.

Des Weiteren ist die Anzahl der L-System-Knoten recht begrenzt, ist aber zunächst ausreichend für die meisten Probleme. Viele nicht implementierte Funktionen aus GroImp lassen sich über alternative Herangehensweisen dennoch realisieren.

Relativ einfach gestaltete sich hingegen die Entwicklung der GUI. MEL bietet einen leichten Zugang dazu über einfache Befehle und Parameter. Alternativ zu der GUI steht dem Anwender auch der implementierte MEL-Befehl zur Verfügung, um XL4Maya zu steuern, somit kann dieser Befehl auch in andere, benutzerdefinierte MEL-Skripte und Maya-Funktionen integriert werden.

Zu einem weiteren großen Vorteil entwickelte sich die Möglichkeit, MEL innerhalb von XL zu verwenden. Somit stehen dem Anwender mehr Funktionen zur Verfügung, als in Java fest implementiert wurden. Ebenso die Zusatzfunktionen, wie *Duplicate* (welche es dem Anwender erlaubt, auf bereits in Maya vorhandene Objekte zuzugreifen, um diese in XL-Regeln zu verwenden), ermöglichen völlig neue Ansätze über die bekannten Grenzen von GroImp hinaus.

Alles in Allem stellt XL4Maya viele aus GroImp bekannte Funktionen in Maya zur Verfügung und bietet XL-Programmierern eine weitere Plattform. Es erweitert Maya um den Zugang zu einem gänzlich anderen Ansatz der Modellierung und verschafft Anwendern die Möglichkeit, die Vorteile herkömmlicher Skript- und Modelliertechniken aus Maya mit den Mitteln von XL zu verbinden.

Literaturverzeichnis

[1] Ole Kniemeyer, Gerhard Buck-Sorlin, Winfried Kurth: GroIMP as a platform for functional-structural modelling of plants, Functional-Structural Plant Modelling in Crop Production (eds.: J. Vos, L. F. M. Marcelis, P. H. B. deVisser, P. C. Struik, J. B. Evers). Proceedings of a workshop held in Wageningen (NL), 5.-8. 3. 2006. Kluwer, Dordrecht (accepted)

[2] Mark Adams, Erick Miller, Max Sims: INSIDE Maya 5, New Riders Publishing, Indianapolis, 2003

[3] Eric Hanson, Keneth Ibrahim, Alex Nijmeh: Maya 6 KillerTips, New Riders Publishing, Indianapolis, 2004

[4] David A. D. Gould: Complete Maya Programming, Elsevier Science, San Fransisco, 2003

[5] Mark Adams, Erick Miller, Max Sims: INSIDE Maya 5, New Riders Publishing, Indianapolis, 2003, Kap. 5

[6] Introduction to MEL Scripting, http://www.nthd.org/nthd/58, 23.8.2006

[7] Martin Aupperle: Die Kunst der Programmierung mit C++, 2. Auflage, Vieweg, Braunschweig/Wiesbaden, 2002

[8] Dirk Louis: C/C++ Professionell programmieren, Markt + Technik, München, 2000

[9] Ralph Steyer: Java 2 Kompendium, Markt+Technik, München, 2001

[10] Albert Kluge: http://www.jjam.de/Java/Applets/Fraktale/Lindenmayer.html, 2004

[11] Przemyslaw Prusinkiewicz, Aristid Lindenmayer: The algorithmic beauty of plants, Springer Verlag, 1990

[12] Lehrstuhl Grafische Systeme, BTU Cottbus: http://www-gs.informatik.tu-cottbus.de/grogra.de/grammars/xl.html, 10.9.2006

[13] Jason Osipa: Stop Staring, Sybex, Alameda, 2003

[14] Shawn Dunn, Petre Gheorghian, Matt Herzog, Scyalla Magloir, Cory Mogk, Rob Ormond: Learning Maya 7, Maya Unlimited Features, Sybex, Alameda, 2005

[15] Rob Bateman: http://www.robthebloke.org/research/maya/mfn_traversing.htm, 9.9.2005

[16] David A. D. Gould: Complete Maya Programming, Vol. II, 2005, Elsevier Science, San Francisco

[17] Antje Kunze, Jan Halatsch: Materialnetzwerke in Maya in Digital Production, Reed Business Information GmbH, München, Ausg. 2/2005

[18] Chris Maraffi: Maya Character Creation, New Riders, Indianapolis, 2004

[19] Peter Ratner: 3-D human modeling and animation, second edition, John Wiley & Sons, Inc., 2003

[20] Edwin Martin: http://www.bitstorm.org/gameoflife/, 5.9.2006

[21] Life, http://www.nthd.org/nthd/322, 23.8.2006

[22] Udo Bischof: http://xl.student-by-default.de/xl4maya/javadoclet/index.html, 10.9.2006