Diplomarbeit

„Node basierte 3D Effekt Entwicklung unter Softimage XSI ICE“

Ausgeführt zum Zweck der Erlangung des akademischen Grades

„Diplom-Ingenieur für technisch-wissenschaftliche Berufe“

am Fachhochschul Master-Studiengang Telekommunikation und Medien St. Pölten

Unter der Erstbetreuung von

Mag. Franz Schubert

Zweitbegutachtung von

Dipl.-Ing. (FH) Mario Zeller

Ausgeführt von

Sebastian Wagner

Tm071066

Wien, am

Ehrenwörtliche Erklärung

Ich versichere, dass

- ich diese Diplomarbeit selbständig verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und mich auch sonst keiner unerlaubten Hilfe bedient habe.

- ich dieses Diplomarbeitsthema bisher weder im Inland, noch im Ausland einem Begutachter/einer Begutachterin zur Beurteilung, oder in irgendeiner Form als Prüfungsarbeit vorgelegt habe.

Diese Arbeit stimmt mit der, von den Begutachtern beurteilten, Arbeit überein.

---------- ----------

Ort, Datum Unterschrift

Zusammenfassung

Immer komplexer werdende Computer generierte Szenen verlangen den Einsatz hoch spezialisierter Tools. Trotz des sehr hohen Funktionsumfangs der bestehenden Softwarepackages, ist es oft nicht möglich, gewisse Effekte mit nur einer Software zu realisieren. Nun müssen entweder weitere Softwarepackages erworben werden, oder Erweiterungen für bestehende Software entwickelt werden. Diese Diplomarbeit beschäftigt sich mit der Erstellung von 3D Effekten in Softimage|XSI unter Verwendung des Interactive Creative Environment (ICE). Im ersten Teil wird ein Überblick über Scripting und Plug-In Entwicklung gegeben. Im zweiten Teil wird das Interactive Creative Environment untersucht und der Workflow der Effekterstellung anhand von Beispielen mit der herkömmlichen Methode (die Verwendung der Tools die Softimage|XSI neben ICE zur Verfügung stellt) verglichen.

Abstract

The increasing complexity of computer generated movie scenes requires highly specialized tools. In spite of a wide functional range, sometimes it’s not possible to realize a certain effect with just one certain software. Now it’s either buying additional software or creating extensions for the existing software. This thesis deals with the implementation of 3D effects using Softimage|XSI and the Interactive Creative Environment. The first part outlines the approach of scripting and plug-in development. The second part analyzes the Interactive Creative Environment and compares the workflow of 3D effect development using ICE to the conventional method.

Danksagung

Ich möchte mich bei Mag. Franz Schubert für die Betreuung meiner Diplomarbeit bedanken. Des Weiteren möchte ich Dipl. Ing. (FH) Mario Zeller für seine Zweitbegutachtung danken.

Herzlich Bedanken möchte ich mich auch bei meiner gesamten Familie, die mich stets auf meinem Weg unterstützt haben und mir diese Ausbildung erst ermöglichten.

Ganz besonderer Dank gilt meiner Freundin Sabine für die Liebe und das Verständnis, dass sie mir während der Erstellung dieser Arbeit entgegenbrachte und für ihre stets aufmunternden Worte die mir während dieser Zeit immer den nötigen Rückenwind gegeben haben.

Inhaltsverzeichnis

1 Einleitung

Aktuelle 3D Modellierungs- und Animationssoftware ist bereits mit einem hohen Funktionsumfang ausgestattet. Ob Simulation von Menschenmassen auf einem Schlachtfeld oder ein realistisch wirkender Wasserfall, beinahe jede gängige High End Softwarelösung bietet bereits vorimplementierte Lösungen für gängige Probleme eines CG Artist. Es kann jedoch keine der heute auf dem Markt befindlichen Softwarepackages alle Bereiche der diversen Spezialgebiete von Crowd-Simulation bis Motion Capturing abdecken.

Immer komplexer werdende Filmszenen und der Anspruch an immer höheren Realismus verlangen den Einsatz von hochspezialisierten Tools. Da die Herstellerfirmen von 3D Software nur schwer jedem einzelnen Wunsch nach einem Spezialtool nachkommen können, wurden Möglichkeiten geschaffen, selbst entwickelte Lösungen in die jeweilige Software einzubinden.

Große Animationsstudios wie „Pixar“ und „Industrial Light & Magic“ entwickeln meist eigene proprietäre Softwarelösungen auf Plug-In Basis. Aber nicht nur Animationshäuser und Special Effects Firmen benötigen ihre eigenen angepassten Tools. Auch im Broadcast und Low Budget Bereich ist der Bedarf an maßgeschneiderten Tools enorm und daher gibt es einen gefestigten Markt für Plug-In und Scripting Lösungen für diverse Softwarepackages. Die meisten Anforderungen sind mit diesen Plug-Ins gedeckt. Dennoch gibt es genügend Gründe für die Entwicklung eigener Tools und Effekte: zum einen ist das Fehlen einer Lösung am Markt der Grund, zum anderen können teilweise die Kosten einer vorgefertigten Lösung zu hoch sein beziehungsweise sind die Plug-Ins überladen und werden mit einer Vielzahl an Optionen geliefert, die nicht benötigt werden.

Nicht selten werden populäre Tools und Plug-Ins in das Standard Funktionsrepertoire eines Softwarepackages aufgenommen.

Softimage|XSI bietet eine Vielzahl an Möglichkeiten für die Entwicklung eigener Effekte und Tools. Angefangen von eigenen Menüleisten über Scripting bis hin zu kompilierten DLLs (*DLL = Dynamic Link Library*) kann Softimage|XSI auf unterschiedlichen Ebenen angepasst werden. Dabei gibt es zwei verschiedene Entwicklungsumgebungen: Softimage|XSI bietet zum einen eine eigene Umgebung, die für das Scripting verwendet

werden kann. Zum anderen wird für das Erstellen von Plug-Ins ein externer Compiler benötigt, der die Programmiersprache C++ in ein Binary als DLL kompiliert, welche in Softimage|XSI eingebunden werden kann.

Ein anderer Ansatz der Effekt- und Toolentwicklung wurde mit ICE, dem „Integrated Creative Environment“, geschaffen. Hierbei muss der/die EntwicklerInn keinen eigenen Code schreiben, sondern kann Funktionssegmente, in diesem Zusammenhang auch Nodes genannt, einfach per Drag and Drop in einer grafischen Oberfläche zusammenhängen.

1.1 Problembenennung

Speziell in kleineren Animationsstudios bzw. in den Grafikabteilungen der Fernsehstudios werden oft schnelle und möglichst einfache Lösungen benötigt, die billig zu erzeugen sind und auch von anderen Benutzern ohne größeren lerntechnischen Aufwand einzusetzen sind.

Die Entwicklung eines Effektes oder Tools kann sich jedoch, je nach Anforderungen und System, sehr aufwändig und komplex gestalten. An Scriptsprachen wie JScript oder VB Script kommt man dabei nur selten vorbei. Bei besonders hohen Anforderungen, speziell an die Geschwindigkeit in der Entwicklungsphase der Szene, kann auf einen kompilierten Effekt nicht verzichtet werden.

Digital Artists sind meistens keine Softwareentwickler, die auch die komplexesten Algorithmen in ein lauffähiges Programm oder Script verpacken können. Und nicht alle Softwareentwickler sind Digital Artists, die alle Anforderungen an einen schnellen und unkomplizierten Workflow kennen und diesen umsetzen können. Das Einsetzen von Digital Artist in Verbindung mit einem Softwareentwickler könnte hier eine gute Lösung bieten, doch diese kann auch sehr schnell das Budget eines kleineren Unternehmens sprengen.

1.2 Ziel und Aufbau der Arbeit

Diese Diplomarbeit soll sich mit dem Entwickeln von 3D Effekten auf grafisch unterstützten, Node basierten Entwicklungsumgebungen beschäftigen. Im speziellen wird das „Interactive Creative Environment“, kurz ICE, von Softimage|XSI behandelt.

Im zweiten Kapitel dieser Arbeit werden grundlegende Begriffe erläutert, die dem Leser einen Einblick in die Materie geben soll und die für das weitere Verständnis der Arbeit relevant sind.

Im folgenden Teil werden die auf dem Markt vertretenen Softwarepackages und deren Einsatz von grafischen, Node basierten Entwicklungsumgebungen sowie Scripting-Möglichkeiten und Erweiterbarkeit beleuchtet.

Das vierte Kapitel der Arbeit beschäftigt sich mit dem SDK ($SDK = Software\ Development\ Kit$) von Softimage|XSI. Dabei wird auf die verschiedenen Erweiterungsmöglichkeiten von Softimage|XSI eingegangen und genauer erörtert, um dem Leser einen Einblick in die Entwicklung von Scripts, Plug-Ins und anderen Tools zu geben. Dies ermöglicht eine bessere Hervorhebung der Unterschiede des Workflows zwischen der herkömmlichen Methode des Scriptings und dem Einsatz von ICE.

Kapitel Fünf beschäftigt sich detailiert mit dem Interactive Creative Environment und beleuchtet deren Funktionsweise und Aufbau.

Im sechsten Kapitel wird die Funktionsweise von ICE im Produktionseinsatz anhand einer eigens dafür erstellten Animation erörtert. Zusätzlich wird dieselbe Szene mit herkömmlichen Tools, wie zum Beispiel dem XSI Partikel System oder Rigid Bodies, und Scripting Einsatz erstellt, um direkte Vergleiche ziehen zu können.

Der siebte Teil der Arbeit beschäftigt sich mit einer Analyse der im sechsten Kapitel gewonnenen Erkenntnisse.

1.3 Forschungsleitende Fragestellung

Kann ICE das Scripting ersetzen, und wenn nicht, wo sind die Grenzen von ICE?

Kann ICE die vorhandenen Simulationsmethoden wie z.B. Soft Bodies ablösen?

Welche Vorteile bietet ICE gegenüber dem herkömmlichen Workflow der Effekterstellung?

1.4 Hypothese

Aus der in Punkt 1.3 formulierten Fragestellung ergibt sich folgende Hypothese:

Sowohl Scripting als auch die herkömmliche Herangehensweise an Rigid- und Soft-Bodies, Cloth und Fluid Simulationen sind selbst für effektreiche Szenen obsolet.

ICE kann den bisherigen Workflow vom Erstellen komplexer Effekte komplett ersetzen. Des Weiteren erlaubt es dem User eine genauere und zentralisiertere Kontrolle über die Effekte, da deren Komponenten bis in das kleinste Detail manipuliert werden können.

2 Begriffsdefinitionen

2.1 Scripting

Die Bezeichnung „Scripting“ ist gleichzusetzen mit „ein Script schreiben/ein Script erstellen“. Ein Script ist ein Programmcode *(Für den Menschen leichter lesbare, logisch aufgebaute Maschinenanweisungen),* der durch Interpreter umgesetzt wird. Der Interpreter analysiert den erstellten Programmcode auf syntaktische Korrektheit und arbeitet jede einzelne Deklaration und jeden Befehl nacheinander ab. Aufgrund dieser Tatsache ist der Ablauf des Programms, respektive Scripts, entsprechend langsamer als kompilierter Programmcode *(siehe Punkt 2.4 Compiler*). Was dies für das Scripting in Softimage|XSI bedeutet, wird in Kapitel 4 genauer erläutert. Als vorteilhaft erweist sich jedoch das leichte Abändern des Programmcodes und die Möglichkeit des direkten Eingreifens in den Code während der Laufzeit. So können die Auswirkungen der Änderungen direkt beobachtet werden. [Lev03]

Es gibt mehrere unterschiedliche Einsatzmöglichkeiten für Scripts. Eines der häufigsten Gebiete ist der Bereich rund um Web Applikationen. Dabei werden komplexere Funktionen auf dem Server, auf dem die Website liegt, ausgeführt beziehungsweise direkt beim Endbenutzer im Browser, der dabei als Interpreter fungiert. Als Beispiel sei hier die Bestellung in einem Online Shop angeführt.

Ein anderes Einsatzgebiet ist das Automatisieren von Arbeitsabläufen in Betriebssystemen. Als Beispiel sollen hier der Windows Scripting Host unter den Windows Betriebssystemen und das Shell Scripting unter Unix basierten Systemen genannt werden.

Der Einsatz von Scripting auf Betriebssystemebene deckt sich mit dem von 3D Softwarepackages. Auch hier werden Arbeitsschritte automatisiert beziehungsweise werden dort Scripts eingesetzt, wo die manuelle Abarbeitung um ein Vielfaches aufwändiger wäre als das Erarbeiten eines Algorithmus *(siehe Punkt 2.5 Algorithmus)* und deren Implementierung in einem Programmcode.

Ein weiterer, vor allem im Zusammenhang mit Scripting, häufig vorkommender Begriff im Bereich der 3D Animation und Modellierung ist die Expression. Eine Expression ist in der Mathematik ein Ausdruck für einen Folge wohl geformter *(einer Regel folgend)* Symbole. In diversen DCC *(= Digital Content Creation)* Programmen werden

Expressions für das Erstellen von mathematisch beschreibbaren Bewegungsabläufen eingesetzt, wie zum Beispiel das Hüpfen eines Balles *(siehe Abb. 1).* [Max03]

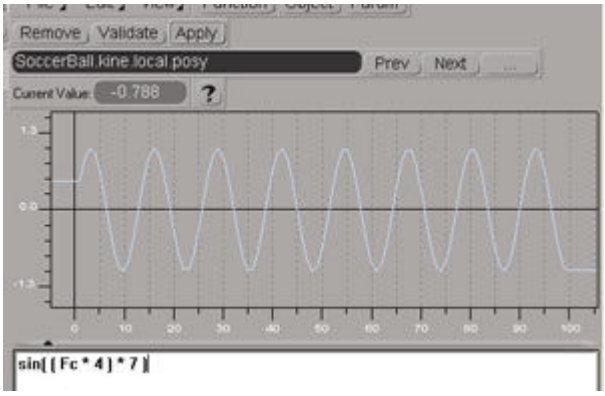

Abbildung 1: Expression in XSI

2.2 Plug-In

Unter einem Plug-In versteht man eine Erweiterung eines Programms, die auch von Drittanbietern oder dem versierten Endbenutzer selbst erstellt werden können. Dabei wird die Funktionalität des eigentlichen Programms so erweitert, dass die hinzugefügten Funktionen direkt aus dem Programm aufgerufen werden können. Ein eigenständiges Ausführen des Plug-Ins ohne das Hauptprogramm ist nicht möglich.

Plug-Ins sind nicht automatisch Erweiterungen Dritter. So kann es bei komplexeren Programmen vorkommen, dass Funktionalität in Plug-Ins ausgelagert wird. Das hat den Vorteil, dass nicht benötigte Teile der Applikation nicht geladen werden müssen, was wiederum zu einem schlankeren, schnelleren und stabileren Programm führt. Vor allem in Audio- und Grafikprogrammen kommt dieses Konzept häufig zum Tragen, da hier ein besonders hoher Funktionsumfang besteht. Ein Auslagern der Funktionalität in Plug-Ins hat nicht nur den Vorteil der besseren Stabilität und höheren Geschwindigkeit: Das Hauptprogramm lässt sich in einer abgespeckten Version günstig verkaufen und der Benutzer kann sich die benötigte Funktionalität dazukaufen. Dies ist zum Beispiel beim Audioprogramm Pro Tools der Fall. Hier kann zum Beispiel das Plug-In „Strike“, ein digitales Schlagzeug, dazugekauft werden, was für den Einsatz in der Musikproduktion gedacht ist. Wird Pro Tools jedoch für das Vertonen eines Filmes verwendet, muss der Benutzer nicht für die Mehrkosten dieser Funktionalität aufkommen, sofern sie nicht benötigt wird. [Plu08]

Ein Plug-In verwendet die vom Hauptprogramm bereitgestellte Schnittstelle, auch API genannt *(siehe Punkt 2.3 API)*, um auf dessen Funktionen zugreifen zu können und diese in Folge erweitern zu können. Zusätzlich muss im Hauptprogramm ein Plug-In Manager implementiert sein, um die Plug-Ins in der Software registrieren und verwalten zu können. [Plu08]

Grundlagen über die Entwicklung und Verwaltung von Plug-Ins in Softimage|XSI werden in Kapitel 4 behandelt.

2.3 API

API ist die Abkürzung für „Application Programming Interface“. Es beschreibt die von Softwareentwicklern zur Verfügung gestellte Schnittstelle des Programms, die zur Entwicklung von Erweiterungen für das jeweilige Programm dient beziehungsweise lassen sich je nach API auch eigenständige Applikationen erstellen. So basiert beinahe jedes Programm auf der vom Betriebssystem bereit gestellten API, um zum Beispiel Funktionen wie der Speicherabruf oder das Zugreifen auf Netzwerkverbindungen zu ermöglichen. [Ore00]

APIs lassen sich in vier Klassen einteilen. Zum einen gibt es die Funktionsorientierte API, zum Beispiel DLLs, die Funktionen als Kommunikation mit der neu entwickelten Software verwendet. Dateiorientierte APIs, wie zum Beispiel Treiberdateien unter Unix Systemen, werden über Dateisystemaufrufe angesprochen. Objektorientierte APIs verwenden Schnittstellenzeiger zur Kommunikation. Unter einem Schnittstellenzeiger versteht man einen Zeiger auf eine Objektinstanz, die auf die implementierte Methode in der Schnittstelle zeigt. [Cod08] Unter einer protokollorientierten API versteht man hardware- und betriebssystemunabhängige Schnittstellen. Zusätzlich kann bei der protokollorientierten API noch zwischen einem allgemeinen Protokoll, wie zum Beispiel SOAP *(Simple Object Access Protocoll)*, und einem anwendungsspezifischen Protokoll, wie zum Beispiel SMTP (*Simple Mail Transfer Protocoll),* unterschieden werden. [Ore00]

In diesem Zusammenhang soll hier auch der SDK, der Software Development Kit, erwähnt werden. Ein SDK ist eine Sammlung von Dokumentationen und Tools, wie zum Beispiel Editoren, für eine Software, die eine API zur Verfügung stellt. SDKs können genutzt werden um Plug-Ins zu erstellen *(siehe Punkt 2.2 Plug-In).*

Die von Softimage|XSI bereit gestellte API und der XSI SDK werden in Kapitel 4.5 genauer erläutert.

2.4 Compiler

Computer interpretieren lediglich Folgen von Instruktionen und sind nicht in der Lage den für Menschen leichter lesbaren Programmcode direkt auszuführen. Vor der Ausführung der beschriebenen Befehle muss der Code also in eine für den Computer verständliche Maschinensprache übersetzt werden. Das Programm das diesen Vorgang übernimmt wird Compiler genannt. [Nik08]

Auf Windows Betriebssystemen sind die Ergebnisse nach einem Kompiliervorgang in der Regel entweder so genannte EXE *(EXE = execute bzw. executable)* Dateien, die direkt ausgeführt werden können oder DLLs, die zwar nicht als eigenständiges Programm ausgeführt werden können, dafür aber für andere Programme als Funktionsbibliothek fungieren. Auf Unix Systemen werden die Ausführbaren Dateien gängig mit BIN, für Binary, *(Binary = Binär, also Maschinencode)* und die Funktionsbibliotheken mit LIB, für Library, gekennzeichnet.

In Kapitel 4.5.3 werden die Vor- und Nachteile von kompiliertem Programmcode gegenüber dem von Interpretern ausgeführten Programmcode erläutert.

2.5 Algorithmus

Vom etymologischen Standpunkt her leitet sich das Wort Algorithmus vom Namen eines persischen Gelehrten, *Abu Ja'far Mohammed ibn Musa al Khwarizimi*, ab und bedeutete in einer alten Bezeichnung soviel wie „Das Rechnen mit arabischen Zahlen". [Bli02]

Zeitgeschichtlich später verwendete Charles Babbage den Begriff Algorithmus für seine auf Lochkarten gespeicherten Funktionsaufrufe für die von ihm gebaute Rechenmaschine. Diese Grundidee sollte sich bis in die Neuzeit durchsetzen. [Har03]

In der heutigen Informatik beschreibt der Begriff Algorithmus ein Verfahren zur Lösung eines Problems, dass durch ein Computerprogramm realisiert werden kann. [Rob99]

Grundsätzlich besteht ein Algorithmus aus einer Eingabe, aus der Verarbeitung und aus einer Ausgabe. Dabei werden nur jene Funktionen als Algorithmus bezeichnet die auch berechnet werden können.

Ein Algorithmus, also eine Funktion f, muss nach Levi und Rembold folgende Forderungen erfüllen:

- ***„Exaktheit**: Die Funktion f kann präzise auf formale Weise beschrieben werden.*
- ***Finitheit**: Die Beschreibung von f ist endlich lang.*
- ***Vollständigkeit**: Die Beschreibung von f ist vollständig, d.h. sie umfasst alle Einzelfälle.*
- ***Effektivität**: Die Einzelschritte von f sind elementar und real ausführbar.*
- ***Terminierung**: Die Funktion f hält nach endlichen vielen Schritten an und liefert ein Resultat; damit wird die Haltebedingung erfüllt.*
- ***Determinismus**: Die Funktion f liefert für die gleichen Eingabewerte stets das gleiche Ergebnis, wobei die Folge der Einzelschritte für jeden Eingabewert genau festgelegt ist. „* [Lev03]

Ein Algorithmus benötigt also Rechen- und Kontrollanweisungen, Fallunterscheidungen, Iterationen und Rekursionen. Der Funktionsumfang einer Programmiersprache kann diese Anforderungen erfüllen. [Lev03]

Algorithmen lassen sich neben arithmetischen Funktionen noch in Sortier- und Suchverfahren aufteilen:

> *„Indeed, I believe that virtually every important aspect of programming arises somwehre in the context of sorting or searching!“ (Knuth, 1981 in Blieberger, Burgstaller, & Schildt, 2002 S. 151)*

Ernst spricht von 3 Schritten die bei der Ausführung von Algorithmen notwendig sind:

- Finden eines Algorithmus, also Lösungsweges, mit definierten Ein- und Ausgabedaten sowie deren funktionale Abhängigkeit.
- Formulieren des Algorithmus in einer Programmiersprache, die eine verständliche und vollständige Beschreibung zulässt.
- Ausführen des Programms auf einem Computer. [Har03]

2.6 Multithreading

In den Anfängen des Computerzeitalters konnten Computer Befehle nur sequentiell abarbeiten und immer nur genau einen Befehl zum gegebenen Zeitpunkt ausführen. Dies hatte den Nachteil, dass während der Ein- und Ausgabeoperation der Prozessor im Leerlauf lief, was eine immense Ressourcenverschwendung darstellte. Aufgrund dessen wurde die Idee geboren, dem Prozessor während dieser Zeit eine andere Aufgabe zu zuteilen.

Heutige Betriebssysteme regeln welches Programm wann und wie die zur Verfügung stehende Prozessorzeit nützen darf. Genau genommen ist ein Programm ein statisches Abarbeiten von Befehlen und erst das Betriebssystem macht daraus durch die Administration von Ressourcen einen dynamischen Prozess, der es dem Programm erlaubt ungenutzte Ressourcen zu verwenden. Diese Tatsache schafft die Illusion, dass auf dem Computer mehrere Prozesse gleichzeitig ablaufen. [Bli02]

Besteht eine Applikation aus mehreren Prozessen, kann es vorkommen, dass diese verschiedenen Prozesse Daten untereinander austauschen müssen. Dies geschieht über Kanäle, die bei großen Datenmengen schnell überlastet sind, was zu Geschwindigkeitseinbußen führt. Um dies zu umgehen wurde das Konzept des Threads entwickelt. Diese Threads kommunizieren nun nicht über Kanäle, sondern über gemeinsam genutzte Speicherbereiche. [Har03]

Multithreading bedeutet also, dass ein Prozess mehrere Threads erzeugt. Steht diesem Prozess jedoch nur ein Prozessor zur Verfügung, so werden die einzelnen Threads sequentiell abgearbeitet. Bei einem Mehrkernsystem oder mehreren Prozessoren in einem Rechner können diese Threads auf die vorhandenen Prozessoren aufgeteilt werden und tatsächlich gleichzeitig ausgeführt werden. Diese Funktionalität muss von Seiten der Software unterstützt werden, damit die Anwendung davon Profitieren kann. [Lev03]

Softimage|XSI unterstützt Multithreading in vielen Bereichen, wie zum Beispiel beim Rendern, Animieren oder bei ICE Nodes.

2.7 3D Effekt

Allgemein versteht man unter dem Begriff 3D Effekt eine Vielzahl an grafischen Effekten, wie zum Beispiel Lensflare, Schatten oder Tiefenschärfe, die beim Betrachter die Illusion eines perspektivischen, räumlichen Sehens erzeugen.

Des Weiteren versteht man unter diesem Begriff auch stereoskopische Effekte. Bei der Stereoskopie wird die Funktionsweise vom Auge und Hirn nachempfunden. Dabei werden zwei Bilder von einem Objekt erstellt, wobei das zweite in der Horizontalachse leicht versetzt aufgenommen wird, so wie es beim menschlichen Auge der Fall ist.

Im Zusammenhang mit dieser Arbeit beschreibt der Begriff 3D Effekt Techniken und Konzepte für aufwändige, in der 3D Software realisierte Effekte, wie zum Beispiel Rauch und Feuer oder Simulationen von Flüssigkeit und Stoff. Dabei werden mehrere Arbeitsschritte von der Konzeptionierung und Erstellung über die Justierung bis hin zur brauchbaren Ausgabe für das Compositing benötigt.

2.8 Prozedurale Entwicklung

In der Softwareentwicklung beschreibt der Begriff prozedurale Entwicklung beziehungsweise prozedurale Programmierung das Unterteilen des zu lösenden Problems in kleinere, leichter lösbare Einheiten, die nacheinander abgearbeitet werden.

Im Bereich des 3D Modeling gibt es unterschiedliche Auslegungen des Begriffs prozedurale Entwicklung beziehungsweise Modeling. Es gibt die Methode der Erzeugung der Geometrie über eine Liste von geometrischen Operationen. Anstatt das Ergebnis einer Operation zu speichern, wird hier die Funktion mit den entsprechenden Parametern abgespeichert.

Ein weiterer Ansatz ist eine der prozeduralen Programmierung nahe gelegten Methode, bei der die Erzeugung der Geometrie auf einem sehr niedrigen Level ansetzt. Durch Aufrufen von Funktionen, Funktionsschleifen und bedingten Anweisungen beziehungsweise Verzweigungen wird eine Geometrie erzeugt. Dies erfolgt in einer dafür geeigneten Programmiersprache, wie zum Bespiel der Generative Modeling Language. [Mar08]

Diesem Ansatz folgt auch die Implementierung eines grafischen, Node basierten Systems von Ganster und Klein. Dabei werden Operatoren und Funktionen miteinander auf einer

grafischen Oberfläche verbunden. Dies erlaubt ein non-lineares Entwickeln komplexer Geometrie mit vergleichsweise wenig handwerklichem Aufwand. [Gan07]

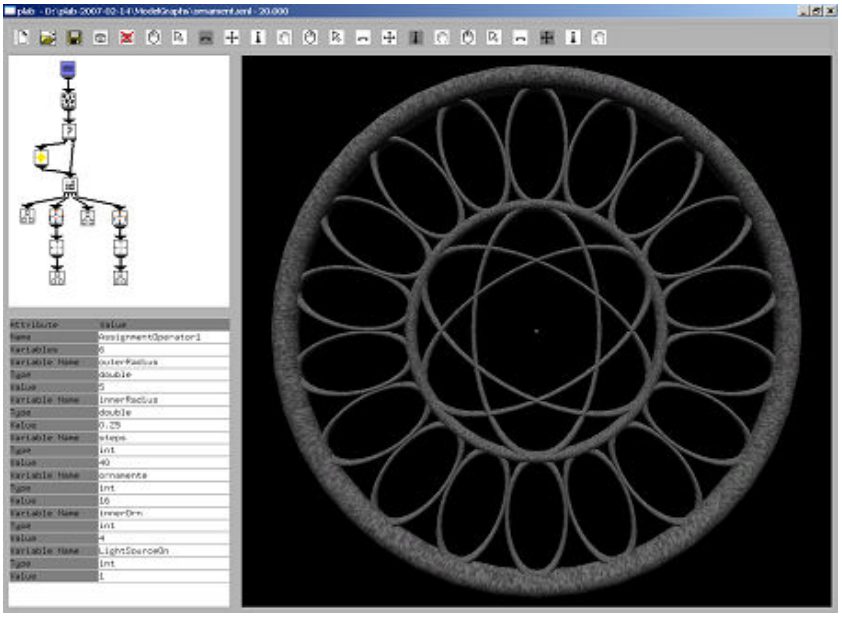

Abbildung 2 Implementierung eines grafischen Systems für prozedurales Modeling

3 Überblick 3D Software

In diesem Kapitel werden die gängigsten, kommerziell eingesetzten, nicht spezialisierten 3D Modeling- und Animationstools beleuchtet. Dabei sollen unterschiedliche Schwerpunkte bei der Betrachtung gesetzt werden. Das Hauptaugenmerk liegt bei den Scriptingeigenschaften der Softwarepakete und die Verwendung von Nodes vom Scripting Bereich bis zur Texturgestaltung. Die Auswahl erhebt nicht den Anspruch der Vollständigkeit. Die Zusammenstellung erfolgte aufgrund subjektiver Eindrücke des Autors, wie Vorkommen in Fachliteratur, Referenzen in diversen Produktionen, sowie Größe der Unternehmen und Produktumfang.

Neben folgenden Softwarepackages gibt es eine Vielzahl an sehr günstiger beziehungsweise lizenzfreier Open Source Software. Zusätzlich gibt es auf unterschiedliche Bereiche spezialisierte Tools, wie zum Beispiel ZBrush, dass sich auf Sculpting *(verändern der Geometrie basierend auf Tiefenwert des digital aufgemalten Pixels)* spezialisiert, oder Massive, das vor allem bei aufwändigen Crowd-Simulations *(Simulieren einer großen Zahl anscheinend eigeständig denkender Individuen, zum Beispiel Soldaten auf einem Schlachtfeld)* eingesetzt wird, oder Vue, das zur prozeduralen Generierung von Landschaften verwendet wird.

3.1 Softimage|XSI

Softimage|XSI wurde im Jahre 2000 von der Firma Softimage Co. in Kanada entwickelt. Die Geschichte von Softimage|XSI begann jedoch schon 1988 beziehungsweise 1986 mit der Gründung von Softimage durch Daniel Langlois, als das Softwarepackage noch unter dem Namen Softimage Creative Environment vertrieben wurde. Microsoft kaufte 1994 Softimage Co. und begann mit der Portierung der Software auf Windows Plattformen. Ab Version 3.0, im Jahre 1995, wurde der Produktname offiziell in Softimage|3D abgeändert und lief schlussendlich mit der Version 4.0 im Jahre 2001 aus, mittlerweile unter Avid Technology Inc., um in einer überarbeiteten Version von Softimage|XSI 1.0 abgelöst zu werden. Bis zur Übernahme durch Microsoft wurde das Produkt nur in Kombination mit einer Silicon Graphics Workstation verkauft und war mit einem Preis ab 50.000$ rein für den professionellen Einsatz ausgelegt. Bekannt war Softimage|3D vor allem für seine leistungsstarken Animationstools. [Sof09]

Softimage|XSI wurde ab Version 1.0 in den Versionen Foundation, Essentials und Advanced verkauft, um den unterschiedlichen Ansprüchen und Budgets der Benutzer gerecht zu werden. Mit der Einführung der XSI Reihe wurde der Focus der Software auf die Spielentwicklung ausgedehnt und mit den Versionen XSI Mod Tool und XSI EXP zwei für diese speziellen Anforderungen angepasste Versionen auf den Markt gebracht.

Softimage|XSI, Softimage|3D sowie die Mod Tool Versionen wurden für erfolgreiche Spiele wie „Half Life 2“, „Crysis“ und „Metal Gear Solid“, verwendet. Im Bereich Film und TV wurde die Software für Filme wie „Sin City“, „Jurassic Park“, „Star Wars“, „Matrix“ und diverse CocaCola Werbespots verwendet.

Mit der Version 1.0 von Softimage|XSI wurde der RenderTree eingeführt, was das Erstellen von Materialien und Texturen über ein Node basiertes, prozedurales System erlaubt. Dabei werden zwischen Shader, Textur und Tool Nodes unterschieden, wobei die Tool Nodes für die Manipulation der erstellten Materialien dienen, wie zum Beispiel die Farbkorrektur einer Textur. [Sof08]

Das mit der Version 7.0 eingeführte Interactive Creative Environment, kurz ICE, folgt demselben Aufbau, wie dem des Render Trees. Dabei stellt ICE eine komplette visuelle Programmierumgebung für das prozedurale Erstellen von Partikel-Effekten, Deformationen und Simulationen dar. Abbildung 3 zeigt Softimage|XSI 7.0 mit geöffnetem ICE Editor. Die genauere Erläuterung der verschiedenen Nodes und der ICE Funktionsumfang würde den Rahmen dieser Einführung sprengen und soll deshalb in Kapitel 5 und 6 genauer erläutert werden. [Sof08]

Neben ICE gibt es in Softimage|XSI eine Vielzahl an Scriptingmöglichkeiten. Dabei verwendet Softimage|XSI keine eigens entwickelte Scriptsprache, sondern erlaubt einem bereits erfahrenen Programmierer eine Auswahl an weit verbreiteten Scriptsprachen. Unterstützt werden:

- VB Script, die per Default voreingestellt ist
- JScript
- Perl
- Python

Dabei können neben dem herkömmlichen Scriptingeinsatz wie Automatismen, Funktionen und Pipeline Scripting auch eigene Menüs mit eigenen Operatoren erstellt werden. Eine

detailliertere Ausführung über den Scripting Einsatz in Softimage|XSI sowie der Plug-In Entwicklung findet sich in Kapitel 4.

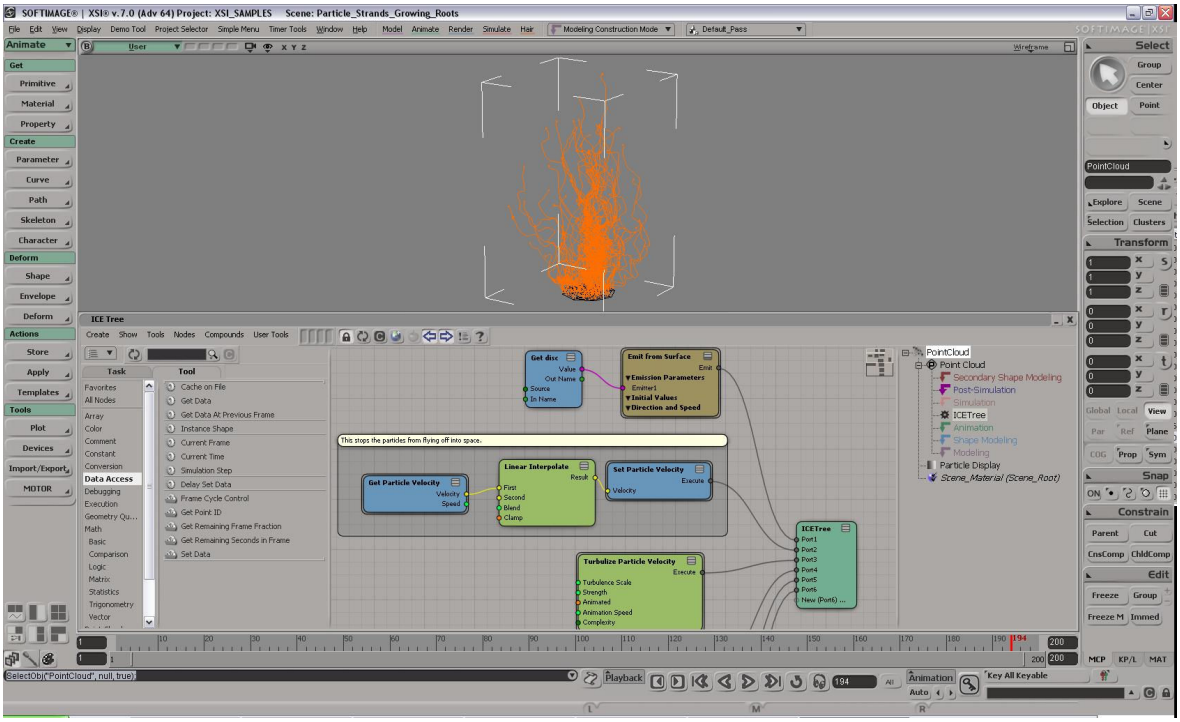

Abbildung 3 XSI mit geöffnetem ICE Editor

3.2 Houdini

Houdini wurde von Side Effects Software Inc. Anno 1991 entwickelt und wird momentan in der Version 9.5 vertrieben. Houdini wurde in über 5000 Szenen in mehr als 200 Filmen, wie zum Beispiel „Spiderman“, „Sin City“, „Blade“ oder „Herr der Ringe“, eingesetzt. Die zugrunde liegende Technologie von Houdini wurde bereits zweimal mit dem „Academy of Motion Pictures, Arts and Sciences Award“ ausgezeichnet. [Sid08]

Die ursprüngliche Stärke von Houdini waren vor allem das Partikel System. Mittlerweile wurden die Schwächen im Bereich Animation und Modeling ausgemerzt, was Houdini zu einem mächtigen Allroundtool machte.

Houdini hebt sich durch seine in allen Bereichen zugrunde liegende prozedurale Herangehensweise ab. Operatoren, die dem Benutzer in Form von Nodes bereit gestellt werden, werden per Drag and Drop in einem Netz zusammengehängt (siehe Abb. 3). Um sich ein Bild von der Arbeitsweise machen zu können, sollen im Folgenden kurz die einzelnen Operatoren erläutert werden. Dabei gibt es folgende Kategorien von Operatoren:

- **Channel Nodes**: bietet verschiedene Funktionen, wie zum Beispiel Filter, Schnittstellen zu MIDI Hardware und diverse Operationen, wie löschen, kopieren, erzeugen von Konstanten, etc.
- **Compositing Nodes**: bietet Operatoren zur Verarbeitung von Bild-Daten, wie zum Beispiel Keying, Motion Blur oder Bump Maps.
- **Object Nodes**: diese Operatoren repräsentieren die Eigenschaften von Objekten in einer Szene, wie zum Beispiel Würfel, Kameras oder Lichter.
- **Particle Nodes**: erzeugt und manipuliert Partikel sowie Partikelsysteme.
- **Render output Nodes**: bietet Operatoren für Rendern und Bildausgabe. Dabei können neben der Houdini eigenen Mantra Render Engine auch externe Engines, wie MentalRay oder RenderMan, eingebunden werden.
- **Dynamic Nodes**: bieten Operatoren für das Erstellen von dynamischen Simulationen.
- **Shader Nodes**: implementieren Materialien und Shader.
- **Surface Nodes**: erzeugt und manipuliert Geometrie.

- **VEX Nodes**: VEX steht für Vector Expression, eine Houdini internen Scriptsprache, und erlaubt es dem Benutzer eigene Operatoren beziehungsweise Nodes zu erstellen.

Houdini bietet neben einer eigenen Scriptsprache HScript und Vector Expression auch Unterstützung für eine Vielzahl an gängigen Scriptsprachen, wie zum Beispiel Python oder JavaScript.

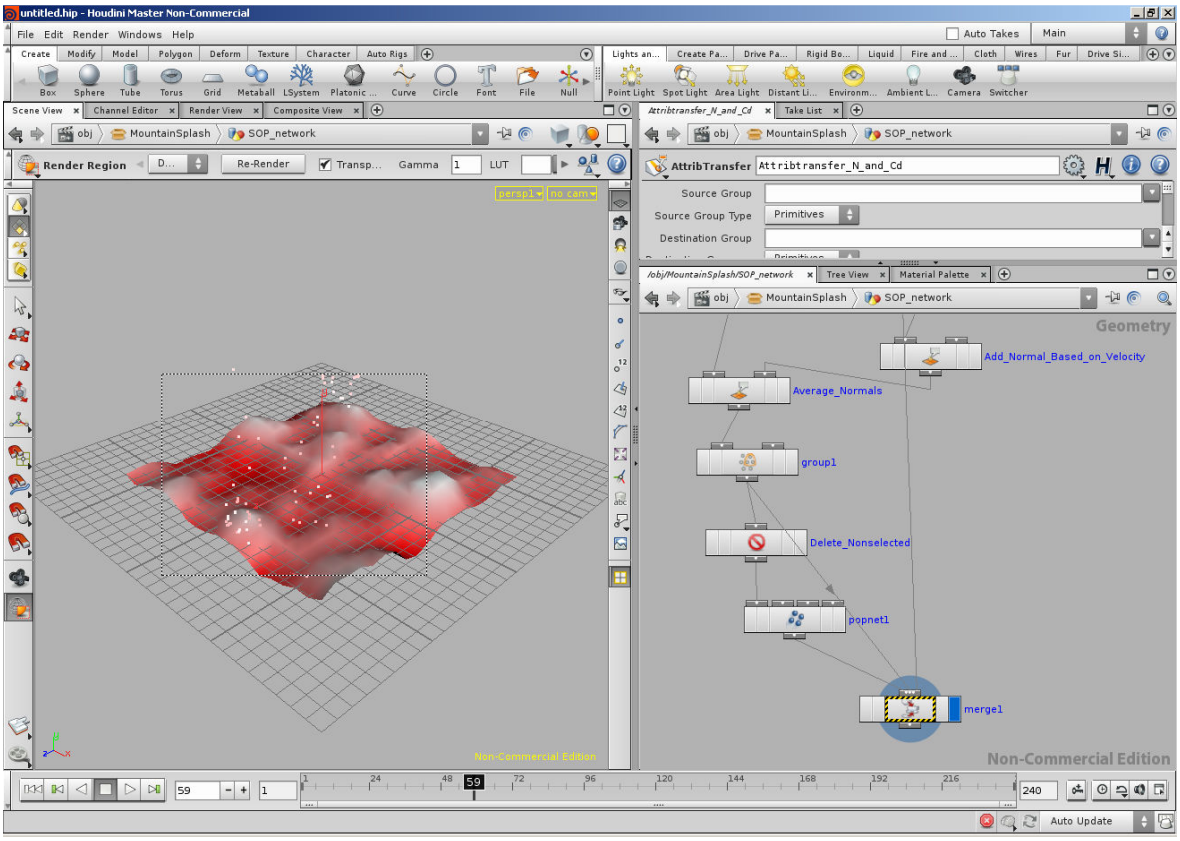

Abbildung 4 Houdini Oberfläche mit geöffnetem Node Fenster

3.3 Maya

Maya wurde 1998 in der Version 1.0 von Alias Wavefront für IRIX Systeme veröffentlicht. Dabei stellt Maya ein Konglomerat von drei verschiedenen Softwarepackages dar, die sich alle im Unternehmen Alias Wavefront zusammenschlossen (Wavefront The Advanced Visualizer, Thomson Digital Image Explorer und Alias Power Animator). Maya wurde bereits seit den ersten Versionen bei George Lucas‘ Industrial Light and Magic neben Sofitmage eingesetzt und für Filme wie „Jurassic Park“ und „Terminator 2“ verwendet. [Tle08]

Maya verwendet, wie Houdini, eine eigene Scriptsprache. Diese Scriptsprache, genannt MEL (Maya Embeded Language) wird nicht zur Erstellung von Automatismen und Funktionen verwendet, sondern bildet auch die Basis der gesamten Benutzeroberfläche von Maya. Dies ermöglicht eine hohe Anpassungsfähigkeit von Maya und das Einbinden und Abfangen von Funktionsaufrufen in eigene Erweiterungen.

Hypergraph bezeichnet in Maya die Objektstruktur einer Szene. Dabei werden nicht nur die reinen Hierarchien dargestellt, sondern auch einzelne Operationen. Wird zum Beispiel ein gebogenes Rohr im Hypergraph dargestellt, so bildet der erste Node das Profil, also ein Kreis, der folgende Node die Richtung und Länge in Form eines weiteren Splines. Darauf folgt ein Extrusion Node, der die Geometrie entsprechend der vorhergehenden Nodes erzeugt usw. Zusätzlich lassen sich verschiedene Nodes miteinander verknüpfen und sich gegenseitig beeinflussen. So ist es zum Beispiel möglich, ein Material Node direkt in einen Extrusion Node zu leiten, um diesen dann beispielsweise durch den Farbwert des Materials zu steuern. [Mah04]

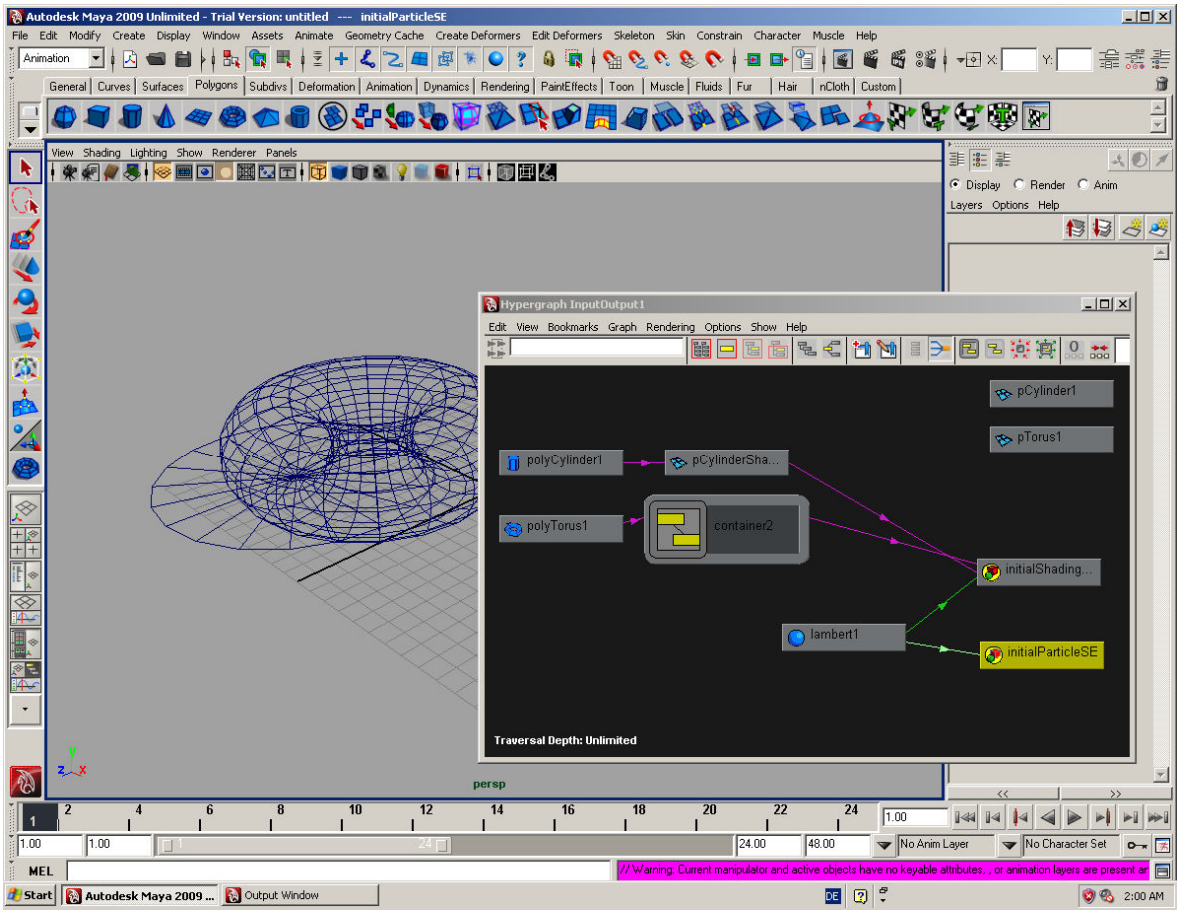

Abbildung 5 Maya mit geöffnetem Hypergraph Fenster

3.4 Cinema 4D

Maxon Computer GmbH wurde 1986 in Deutschland von Harald Egel, Harald Schneider und Uwe Bärtels gegründet. Im Jahre 1988 wurden bereits Zweigstellen in Californien, USA und England eröffnet. Die ursprünglichen Tätigkeitsbereiche lagen bei der Erstellung von Fachzeitschriften in der Computerbranche sowie der Entwicklung von Hard- und Software. Später verlagerte sich der Focus rein auf die Softwareentwicklung mit der Spezialisierung auf Grafiksoftware. Dabei sind die zwei größten Mitglieder der Produktfamilie Cinema 4D als Animations- und Modelingtool und BodyPaint 3D, das die dreidimensionale Erstellung von Texturen direkt auf das Drahtgittermodel ermöglicht. Neben Medienproduktionen wird Cinema 4D auch stark in der Architekturvisualisierung verwendet, was durch eine eigene, speziell auf die Bedürfnisse des Architekten angepasste Version von Cinema 4D unterstützt wird.

Bekanntere Filmproduktionen die Cinema 4D und BodyPaint 3D im Einsatz hatten sind unter anderem Spiderman, The Incredible Hulk sowie Die Chroniken von Narnia. [Max08]

Cinema 4D verwendet die eigens entwickelte Scriptsprache C.O.F.F.E.E., welche an Java angelehnt ist. Als grafische Erweiterung und um auch in Programmieren weniger erfahreneren Benutzern den Zugang zum reichhaltigen Funktionsumfang von C.O.F.F.E.E. zu ermöglichen, wurde XPresso eingeführt. Dabei handelt es sich um einen grafischen Editor, der es erlaubt einfache Routinen und Operationen über Nodes zu komplexen Funktionen, genannt XGroups, zusammen zu hängen. Dabei gibt es vier verschiedene Node Gruppen:

- **XPresso**: Beinhaltet sämtliche grundlegende Operationen und Funktionen, wie zum Beispiel Multiplikation, Boolsche Operationen oder auch das Generieren von Noise.
- **Hair**: Sämtliche Funktionalität im Zusammenhang mit Haaren, wie zum Beispiel Haar Kollisionen, befindet sich in den Hair Nodes.
- **Motion Graphics**: Bietet Nodes zum Erfassen von Bewegungsdaten.
- **Thinking Particles**: Beinhaltet Nodes zum Erstellen und Manipulieren von Partikeln.

Da XPresso nicht die gesamte C.O.F.F.E.E. Funktionalität abdecken kann, gibt es einen extra Node, der es erlaubt C.O.F.F.E.E. Scripten direkt in XGroups einzubinden. [Arn03]

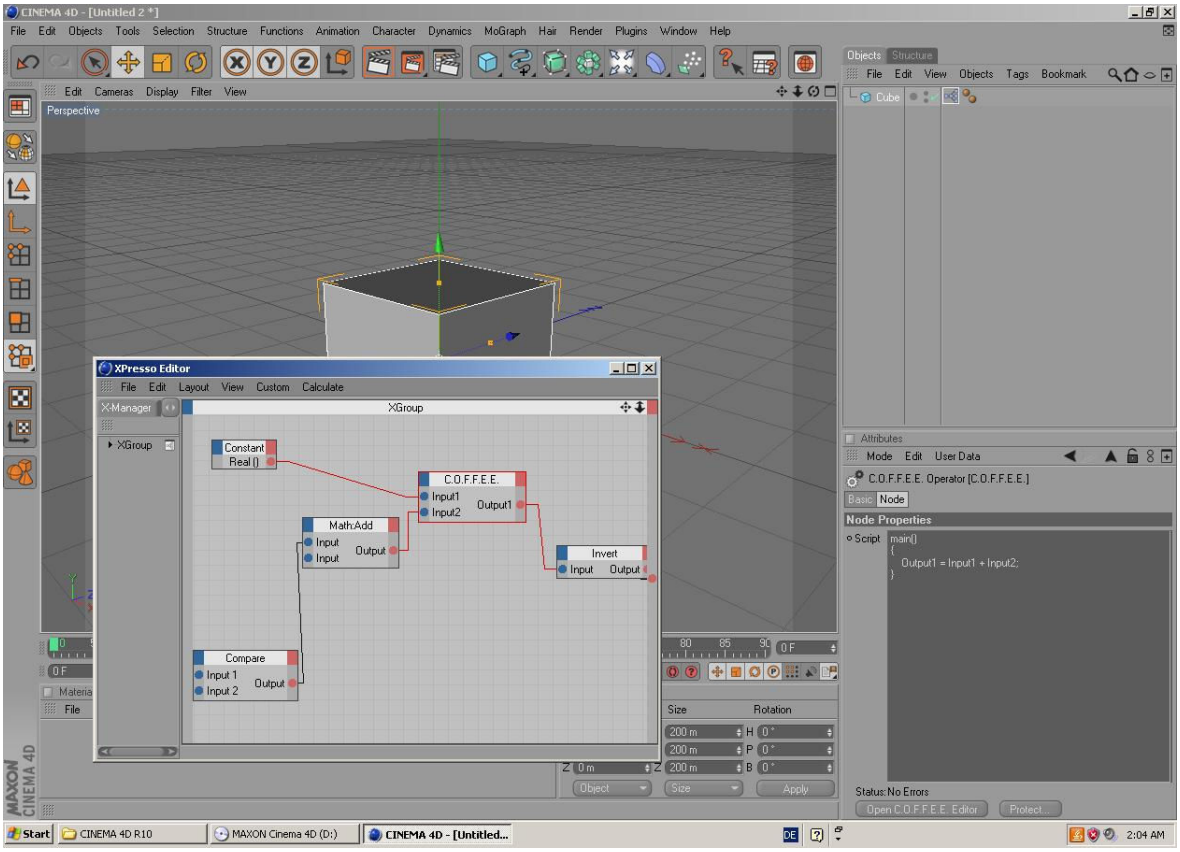

Abbildung 6 Cinema 4D mit geöffnetem Xpresso Editor und C.O.F.F.E.E. Editor

3.5 Lightwave 3D

Die Firma NewTek wurde 1985 von Tim Jenison gegründet und setzte mit dem Video Toaster für Amiga neue Maßstäbe für das Videoediting im Desktopbereich. Für diese Entwicklung erhielt NewTek 1993 den „Emmy Award for Technical Achievement“. Im Jahre 1995 wurde das 3D Animationssystem Lightwave für Windows veröffentlicht und erhielt bereits 1997 den Acadamy Award für die besten Visual Effects im Film Titanic. Zudem wurde Lightwave in diversen Star Trek Serien und aktuelleren Kinofilmen wie The Dark Knight oder 300 eingesetzt. [New09] [Sco08]

Lightwave 3D verwendet ebenfalls wie die bereits erwähnten Softwarepackages eine eigene Scriptsprache, die von einer bereits bestehenden Sprache abgeleitet ist. Im Fall von LScript ist die Scriptsprache angelehnt an Java und C. LScript kann, wie die meisten Scriptsprachen, direkt in Lightwave 3D erstellt und ausgeführt werden. Dabei gibt es drei unterschiedliche Arten von Scripten. Zum einen wird der Programmcode erstellt und ausgeführt. Zum anderen können die Scripts auch direkt in Lightwave 3D, plattformunabhängi, kompiliert werden, was es ermöglicht, den Code für Dritte unleserlich

zu machen, um die Algorithmen zu schützen. Zusätzlich bietet Lightwave 3D die Möglichkeit, Scripten ohne jegliche Programmierkenntnisse zu erstellen. Dies geschieht über den LScript Commander. Dabei werden lediglich Befehle bereit gestellt die auch über das User Interface von Lightwave ausführbar sind. Somit ist der LScript Commander kein vollständiger Ersatz für programmierte LScripts. Um die Handhabung komplexerer Scripten zu erleichtern, liefert Lightwave 3D einen einfachen GUI *(Graphical User Interface)* Editor für die LScripts. [War07]

Lightwave 3D verwendet ein Node basiertes System zur Entwicklung komplexer Shader und Materialien. Dabei lassen sich die verschiedenen Parameter nicht nur durch Material und Shader Nodes beeinflussen, sondern auch durch Eigenschaften verschiedener Objekte. So lässt sich zum Beispiel die Transparenz eines Materials entsprechend der Brennweite der Kamera verändern oder die Position einer Textur entsprechend der Rotation eines Objektes. [War07]

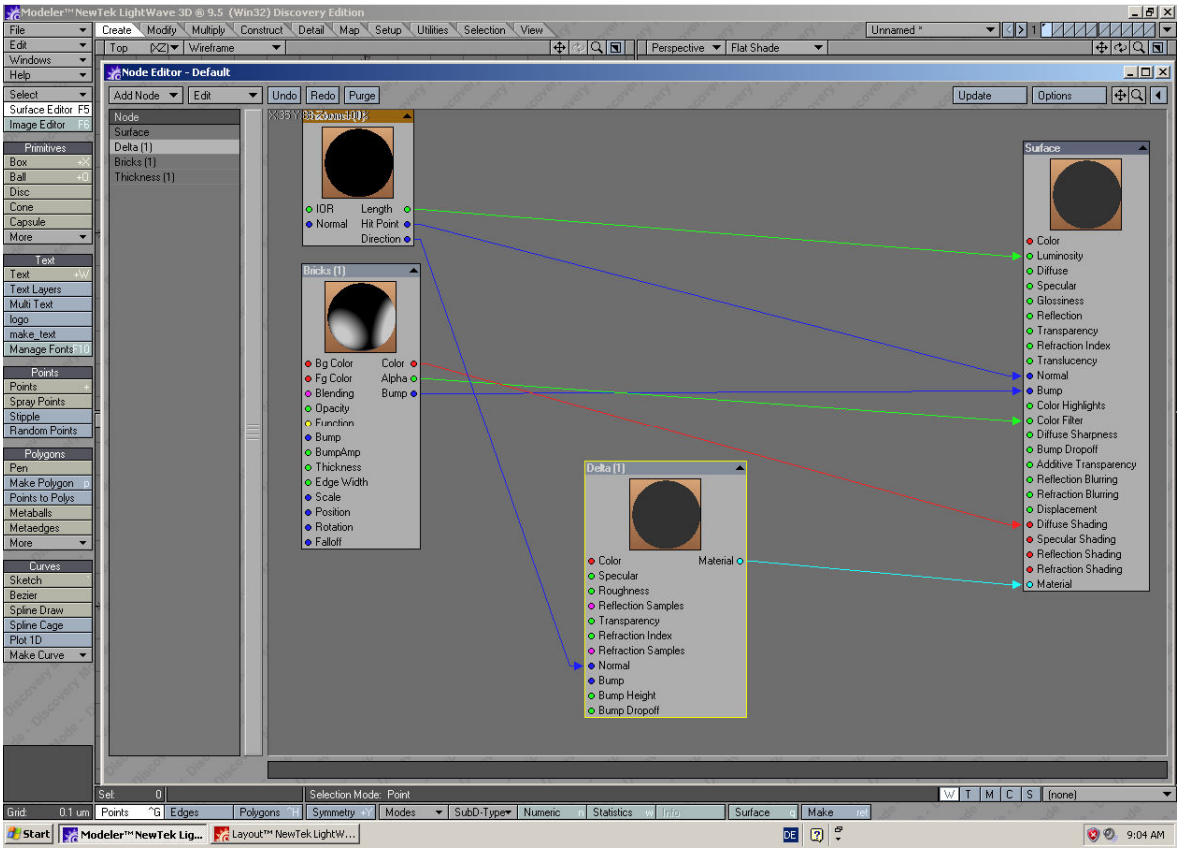

Abbildung 7 Lightwave 3D mit geöffnetem Node Editor

4 XSI Plug-Ins und Scripting Development

4.1 Grundlegendes

Softimage|XSI bietet eine Vielzahl an verschiedensten Erweiterungsmöglichkeiten. Beginnend bei eigenen Shadern über Scripten in JScript oder VB Script, bis zu kompilierten und hochspezialisierten Tools. Das folgende Kapitel soll einen kurzen Überblick über dieses umfangreiche Themengebiet geben, um dem Leser die Unterschiede zu dem in Kapitel 5 erläuterten Interactive Creative Environment besser verdeutlichen zu können.

Werden eigene, spezialisierte Tools benötigt, bedarf es zuerst einer genauen Bedarfsanalyse. Vieles kann mit einem einfachen Proxy Parameter oder einer Expression realisiert werden, ohne dabei auf aufwändige Scripten mit grafischen Benutzeroberflächen oder kompilierte Plug-Ins zurückgreifen zu müssen. Es gibt drei Aspekte, die es zu beachten gibt:

- Je mehr Parameter und je komplexer das Script wird, umso mehr kann sich der Umstieg auf ein kompiliertes Plug-In rentieren, was vor allem hinsichtlich der Geschwindigkeit Vorteile mit sich bringt. Dies ist zum Beispiel bei Animationen mit Simulationen ein wichtiger Aspekt.
- Wird oben genannte Tatsache mit einem häufigen Verwenden des Tools kombiniert, wie zum Beispiel ein Set von Modeling Tools, spricht dies ebenfalls für das Erstellen eines Plug-Ins, da die Ausführung eines Scripts immer mit mehreren Arbeitsschritten verbunden ist und so den Workflow behindert.
- Werden die Tools Dritten zur Verfügung gestellt, lassen sich bei Scripten die Algorithmen und damit mögliche Firmengeheimnisse nur schwer oder gar nicht verbergen, da in Scripten die Algorithmen und der Code immer in einer für den Menschen lesbaren Form vorhanden sind. [Guy01] und [And05]

Verwaltet werden die Scripten und Plug-Ins über den Plug-In Manager in Softimage|XSI. Dieser bietet einen Überblick über sämtliche installierte Plug-Ins und Erweiterungen und bietet des Weiteren diverse Wizards *(Dialogfenster der Schrittweise das Auswählen und Zusammenstellen von Parametern und Komponenten erlaubt)* für die einfachere Erstellung von Plug-Ins. Detailierte Informationen über die Erstellung von Plug-Ins und Scripten finden sich in den Kapiteln 4.4, 4.5 und 4.6. Man unterscheidet außerdem

zwischen sich selbst installierenden Plug-Ins und Plug-Ins, die vom Benutzer eigenhändig über den Plug-In Manager installiert werden müssen. Selbst installierende Plug-Ins müssen sich dabei in den jeweiligen Verzeichnissen der Applikation befinden und die Scripten beziehungsweise kompilierten Plug-Ins eine Funktion implementieren die das Script oder Plug-In in der Applikation registrieren. Beim Start von Softimage|XSI werden dann die jeweiligen Ordner überprüft und etwaige darin vorhandene Erweiterungen automatisch über diese Funktion installiert. Besteht ein Plug-In aus mehreren unterschiedlichen Dateien, wie zum Beispiel DLLs, JScript Dateien und anderen Ressource Dateien, lassen sich diese in eine einzelne Datei zusammenfassen, in Softimage|XSI Addon genannt, und können somit einfacher verwaltet und an Dritte weitergegeben werden. Die Installation erfolgt dabei über ein eigenes Menü außerhalb des Plug-In Managers. [XSI08]

In den folgenden Kapiteln wird genauer auf die Programmierung und Scriptentwicklung eingegangen. Dabei ist anzumerken dass die Erläuterungen auf Informationen und Dokumentationen basieren, die für Windows Plattformen geschrieben worden sind.

4.2 XSISDK

Unter dem Softimage|XSI Software Devekopment Kit versteht man nicht nur die Schnittstellen, Scriptingmöglichkeiten und Tools von Softimage|XSI, sondern auch die Dokumentation eben dieser, ohne die ein erfolgreiches und produktives Entwickeln nicht möglich wäre.

Im Detail besteht der XSI SDK aus einer API für C++, der Scripting Command API und dem XSI Objekt Modell. Dabei enthält der SDK Bibliotheken für Windows und Linux, je nach Distribution. Des Weiteren wird bei kompilierten Plug-Ins zwischen 32 und 64 Bit Systemen unterschieden. Abbildung 7 zeigt die Architektur des XSI SDK. In den Kapiteln 4.3 und 4.4 wird genauer auf die API und das XSI Objekt Modell eingegangen und deren unterschiede erläutert. Kapitel 4.5 geht anschließend genauer auf die Scripting Command API ein. [Sof08]

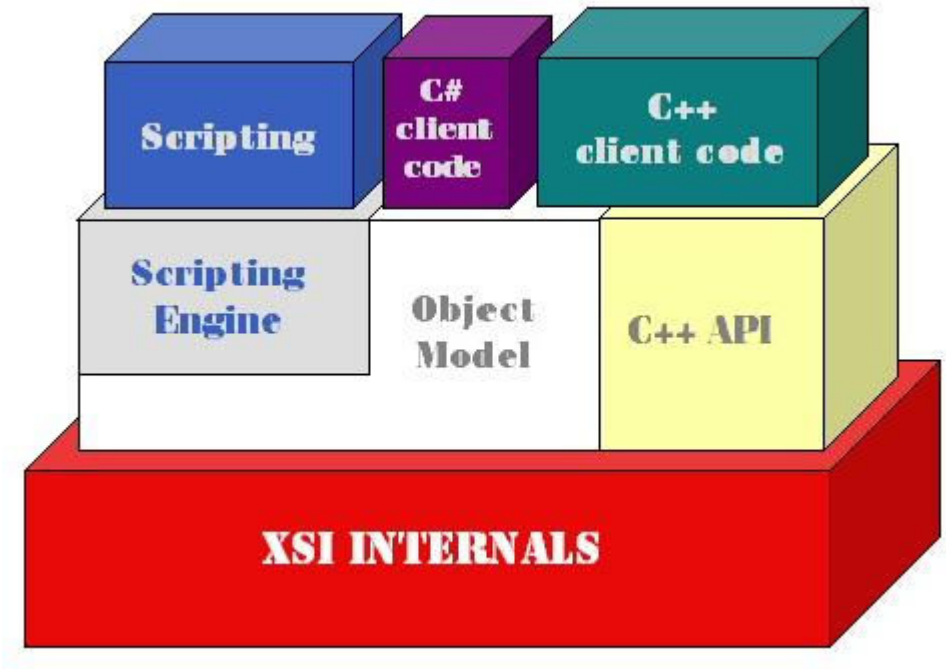

Abbildung 8 Die XSI SDK Architektur [Sof08]

4.3 XSI Objektmodell

Das XSI Objekt Modell besteht aus Objekten, wie zum Beispiel Kameras, Partikel, Shader oder Property Page Items, aus Eigenschaften der Objekte, wie zum Beispiel Farbe oder Position, aus Methoden, die diese Objekte und deren Eigenschaften manipulieren können, wie zum Beispiel Punkte hinzufügen, Objekte deformieren oder die Farbe ändern, und aus Konstanten, wie zum Beispiel Filter und Grundobjekte. Die daraus entstehenden Hierarchien lassen sich grafisch darstellen. [Sof08]

Abbildung 8 zeigt die Objekt-Struktur auf Applikationsebene. Die Abgebildete Struktur zeigt nicht die vollständige Hierarchie des Objekt Modells. Die Hierarchie unter dem Model wird in Abbildung 10 dargestellt. Des Weiteren sind die angeführten Bezeichnungen nur die Namen der Objekte und können nicht eins zu eins in einem Script übernommen werden.

Um ein Objekt oder eine Eigenschaft eines Objektes zu erhalten, welches sich in der Szene befindet, muss generell immer der gesamte Pfad zu eben diesem angegeben werden. [SBl08] Neben soeben genannter Methode, Eigenschaften und Objekte anzusprechen, gibt es die Möglichkeit, Objekte direkt über die Selektion anzusprechen. Der Benutzer braucht sich im Script nicht mehr um den Pfad des Objektes kümmern, sondern muss lediglich das Selection-Objekt ansprechen, welches die vom Benutzer ausgewählten Objekte enthält. Die

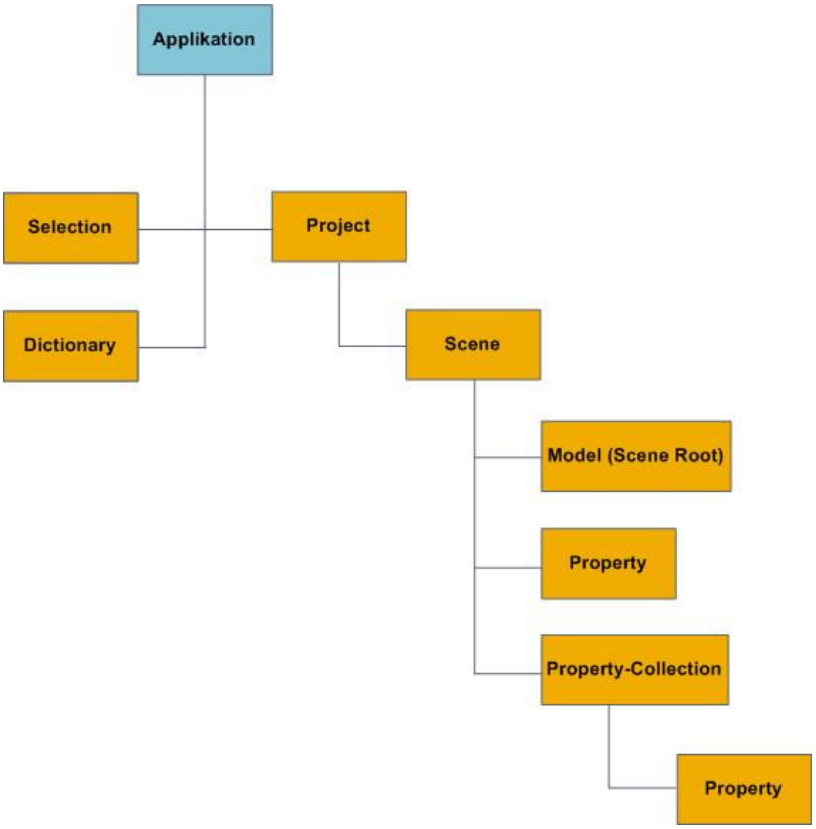

Abbildung 9 XSIOM auf Applikationsebene [SBl08]

dritte Methode bedient sich des Dictionary-Objekts. Das Dictionary-Objekt beinhaltet hauptsächlich Definitionen der XSI Objekte und ermöglicht das Überprüfen der Zugehörigkeit eines bestimmten Objekts zu einer Klasse, wie zum Beispiel Lichter oder Shader. Zudem lassen sich über die Methoden des Dictionary-Objekts Objekte direkt über ihren Namen ansprechen. [Sof08]

Das folgende kurze Code Beispiel in JScript greift auf den Namen der Scene Root zu, weist dieser einen neuen Namen zu (Zeile 1, siehe unten), übergibt diesen der Variable X (Zeile 2) und gibt anschließend den Namen in der Konsole des Script-Editors aus (Zeile 3). Der Name ist dabei eine Eigenschaft der Scene Root. Die hier eingesetzte Methode entspricht der ersten erläuterten Herangehensweise.

```
1: Application.ActiveProject.ActiveScene.Root.Name = "Neuer Name";
2: var x = Application.ActiveProject.ActiveScene.Root.Name;
3: LogMessage(x);
```

In Softimage|XSI besteht jede Szene aus zumindest einem Modell, der Scene Root, in deren Hierarchie der Großteil der XSI Objekte zu finden sind. Dabei ist zwischen abstrakten und realen Objekten zu unterscheiden. Abstrakte Objekte sind als solche nicht im Objektexplorer von Softimage|XSI zu finden. Sie dienen lediglich als Container für Objekte und erben deren Eigenschaften und Methoden beziehungsweise stellen diese den

untergeordneten Objekten zur Verfügung. (In Abbildung 10 grün hinterlegt). Dies ermöglicht dem Entwickler das Zugreifen auf Objekte und Eigenschaften über diese abstrakten Objekte. Reale Objekte finden sich im Objektexplorer und lassen sich direkt über die Pfadangabe ansprechen. Des Weiteren gibt es Objekte, wie zum Beispiel die Group *(3D Objekte die zu einer Gruppe zusammengefasst wurden)*, die zwar real im Objektexplorer zu finden sind, aber nur über die hierarchisch darüber stehende Sammlung oder Collection, in diesem Beispiel die GroupCollection, ansprechbar sind. Diese Collection wiederum ist für den Benutzer nicht sichtbar. [Sof08] und [SBl08] Das folgende Beispiel in VB Script soll diese Eigenschaft verdeutlichen sowie die Funktionsweise eines abstrakten Objekts demonstrieren:

```
1: Set oRoot = ActiveSceneRoot
2: Set oGroup = oRoot.AddGroup
3: Set oCube = oRoot.AddGeometry( "Cube", "MeshSurface")
4: oGroup.AddMember oCube

5: For Each oMember in oGroup.Members
6:  LogMessage oMember.Name & " ist Mitglied von " & oGroup.Name
7: Next

8: For Each oOwner in oCube.Owners
9:    LogMessage oOwner.Name & ", "
10: next
```

In Zeile 1 (siehe oben) wird der Variable oRoot die momentane aktive Scene Root zugewiesen. Dieser wird in Zeile 2 eine Gruppe hinzugefügt und der Pfad in der Variable oGroup gespeichert. In Zeile 3 wir der Szene ein Würfel Grundobjekt hinzugefügt und der Pfad in der Variable oCube abgelegt. In der folgenden Zeile 4 wir der zuvor erstellten Gruppe der Würfel hinzugefügt. Von Zeile 5 bis Zeile 7 wird für jedes Mitglied der Gruppe der Name ausgegeben und deren Gruppenzugehörigkeit angeführt. In einer weiteren Schleife von Zeile 8 bis 10 werden die Besitzer, also zu wem das Objekt gehört (siehe Ergebnis des Scripts), des Würfels ausgegeben. Dabei stellt die Collection „Owners“ eine Eigenschaft des Abstrakten Objektes ProjectItem dar (Siehe Abbildung 10), die dem Würfel-Objekt übergeben wird. Wird das Script in einer leeren Szene ausgeführt, lautet das Ergebnis wie folgt:

```
' INFO : cube ist Mitglied von Group
' INFO : Scene_Root,
' INFO : Layer_Default,
' INFO : Background_Objects_Partition,
' INFO : Group,
```

Mit der Methode LogMessage wird im Script Editor immer eine Zeichenkette ausgegeben, die mit „INFO“ gekennzeichnet ist. Die Gruppe „Group“ enthält erwartungsgemäß nur ein Objekt mit dem Namen „cube“. Der Würfel wird unter mehreren Besitzern aufgeteilt: die Scene Root, der Default Layer des Scene Layers, die Hintergrundpartition des Default Renderpasses und die Gruppe zu der der Würfel hinzugefügt wurde. Dies sind, bis auf die Gruppe, allesamt Standardwerte.

Das XSI Objekt Modell findet auch Verwendung beim Erstellen von ICE Scripten. Sämtliche Eigenschaften und Objekte in XSI lassen sich über Daten-Nodes auslesen und zuweisen. Im Kapitel 5 wird im Zuge der Erörterung von ICE genauer auf diese und weitere Funktionen eingegangen. [XSI08]

Da die vollständige Erläuterung der einzelnen Objekte der Model-Hierarchie den Rahmen dieser Arbeit sprengen würde, sei an dieser Stelle auf die XSI SDK Dokumentation verwiesen, wo sich sämtliche Eigenschaften und Methoden der Objekte finden.

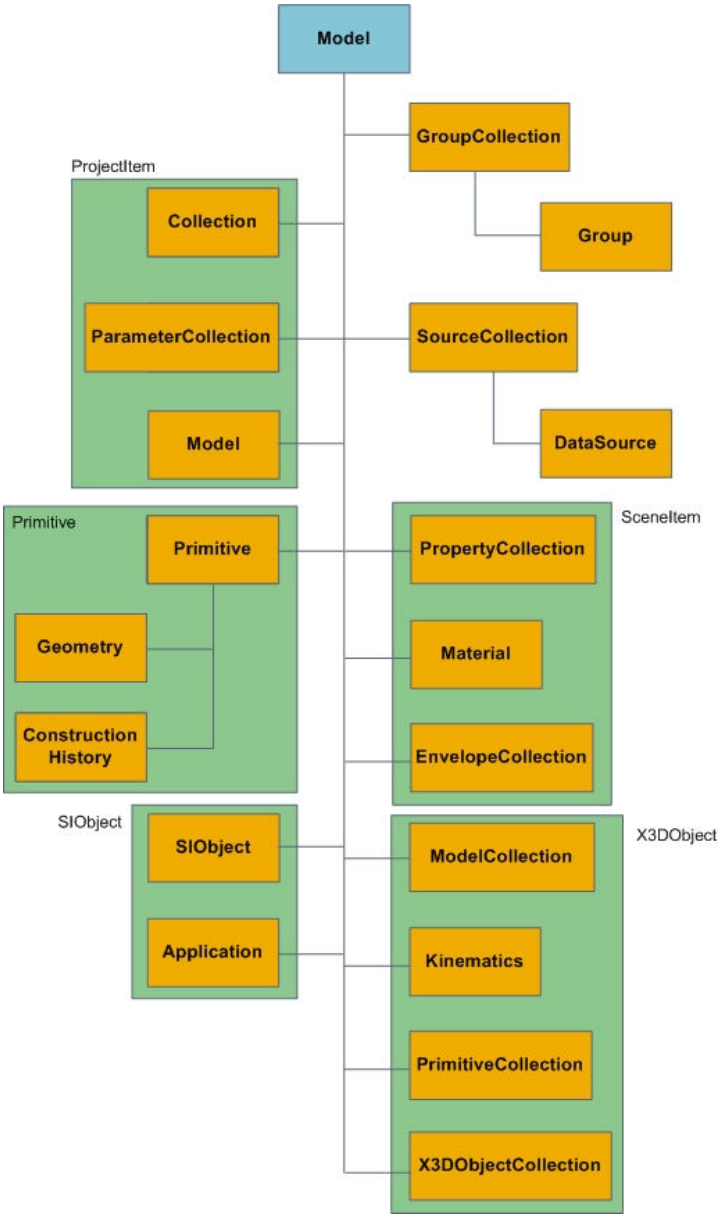

Abbildung 10 XSIOM auf Model-Ebene [SBl08]

4.4 Plug-In Verwaltung

Plug-Ins werden über den Plug-In Manager verwaltet (siehe Abbildung 11). Eine Übersicht liefert dabei der „Tree“, der zum einen den „User Root“ anzeigt, in welchem die eigenen und die Plug-Ins Dritter abgelegt werden, und zum anderen den „Factory Root“, der einen Überblick über die mitgelieferten Plug-Ins und Addons von Softimage|XSI liefert. Neben diesen zwei Kategorien gibt es noch die Möglichkeit so genannte Workgroups einzubinden. Bei Workgroups handelt es sich, ähnlich wie bei Addons (Siehe 4.1), um ein Set von Plug-Ins, Scripten, Layouts und anderen Erweiterungen die von Benutzern die am gleichen Projekt arbeiten über ein Netzwerk eingebunden werden können. Verwaltet werden die Workgroups unter dem gleichnamigen Reiter im Plug-In Manager.

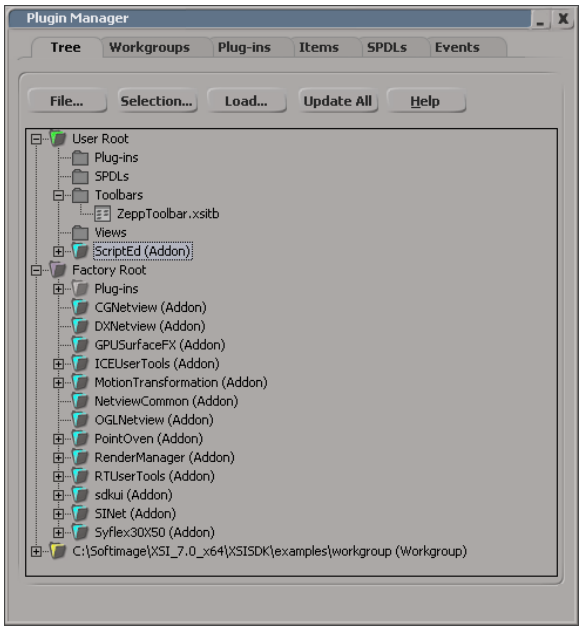

Abbildung 11 Plug-In Manager

Plug-Ins werden im Reiter Plug-Ins installiert, verwaltet und erstellt.

Unter dem Reiter Items werden Komponenten die ein Plug-In beinhalten kann, wie zum Beispiel eigene Menüs, Propertypages und DLLs, angezeigt und können je nach Typ gefiltert werden.

SPDL Dateien, die beschreiben wie ein Objekt in XSI erstellt wird und die Grundlage für die Shaderentwicklung und in früheren Versionen von Softimage|XSI auch für Custom Operators bilden, finden sich unter dem gleichnamigen Reiter. Die SPDL Dateien können hier mit einem Wizard erstellt werden.

Eigene und vorinstallierte Events, also Aktionen die beim Auslösen eines bestimmten Ereignisses aufgerufen werden, wie zum Bespiel das automatische Erstellen eines Würfels beim erstellen einer neuen Szene, finden sich im Reiter Events. [XSI08]

Im folgenden Kapitel wird genauer auf die Erstellung von Plug-Ins und Scripten eingegangen. Dabei wird zuerst grob in kompilierte Plug-Ins und in gescriptete Plug-Ins unterschieden und folgend die einzelnen Erweiterungen im Detail erläutert.

4.5 Kompilierte Plug-Ins

Kompilierte Plug-Ins für Softimage|XSI werden in der Programmiersprache C++ entwickelt. Die Plug-Ins können nicht direkt in XSI erstellt werden, sondern müssen in externen Editoren geschrieben und mit entsprechenden Compilern in DLLs übersetzt werden. Um das Erstellen eines Plug-Ins zu erleichtern und zu beschleunigen, stellt der SDK in Softimage|XSI einen Wizard zur Verfügung, der, je nach Art des Plug-Ins, ein Code-Grundgerüst mit den wichtigsten Funktionen erstellt und gleich die für den jeweiligen Typ des Plug-Ins notwendigen Bibliotheken einbindet. Zusätzlich wird bei Softimage|XSI unter dem Windowsbetriebssystem eine Projektdatei für Microsoft Visual C++ angelegt, in der sämtliche für die Kompilation des Codes notwendigen Informationen in Form einer XML Datei abgelegt sind. Dazu gehört die Art der Ausgabe, im Fall eines Plug-Ins immer eine DLL, der Ordner in dem die DLL abgelegt werden soll, um welche Plattform und ob es sich um ein 32 oder 64 Bit System handelt sowie wo sich die Bibliotheken befinden die benötigt werden. [Sof08]

Unter Linux wird eine GNU Make Datei erstellt. GNU steht dabei für „GNU is not Unix“ und ist ein Projekt der Free Software Foundation. Bei Make handelt es sich um ein Programm, welches die Erstellung von ausführbaren Dateien beziehungsweise SO *(Shared Object)* Dateien, dem Linux äquivalent der DLL Dateien, über das so genannte Makefile kontrolliert. Inhaltlich ist das Makefile ähnlich der Projektdatei, jedoch mit dem Unterschied, dass sich Programme auch direkt über Make installieren und updaten lassen. [GNU06]

In Softimage|XSI können alle Erweiterungen auch durch ein Script implementiert werden, mit Ausnahme von eigenen ICE Nodes, die aus Kompatibilitätsgründen nur in kompilierter Form als DLL eingebunden werden können. Zudem würde ein Script innerhalb eines ICE Scripts starke Geschwindigkeitseinbußen mit sich bringen (Siehe Punkt 4.5.3 Interpreter

versus kompiliertem Code). Eine Ausnahme bilden eigene ICE Compounds, die jedoch selber nur aus ICE Nodes bestehen und im eigentlichen Sinn keine Plug-Ins sind. [Sof08]

4.5.1 API (Application Programming Interface)

Seit Version 3.0 von Softimage|XSI gibt es eine C++ API, die die zuvor verwendete COM API von Microsoft abgelöst hat. Das Component Object Model, kurz COM, bezeichnet grob die API von Microsoft, die eine Kommunikation zwischen Prozessen beziehungsweise DLLs ermöglicht. Plug-Ins müssen also über diese COM Schnittstellen mit Softimage|XSI kommunizieren. [ITW09]

Der Vorteil der C++ API gegenüber der COM API liegt in der Unabhängigkeit der Microsoft Bibliotheken, die mit der Verwendung des Component Object Models einher geht. Somit lässt sich der Code einfacher auf Linux portieren.

Nicht alle Objekte des Object Models wurden in die C++ API portiert. Somit kann es durchaus erforderlich sein, für gewisse Ansprüche auf die COM API um zu steigen, sollte die benötigte Funktionalität nicht in der C++ API vorhanden sein. Um ein möglichst plattformunabhängiges Entwickeln zu ermöglichen und um nicht auf unterschiedliche Bibliotheken von Drittanbietern ausweichen zu müssen, wurde eine Klasse für die C++ API entwickelt, die es ermöglicht, COM Methoden auch unter Linux zu verwenden. Diese Klasse gibt Zugriff auf alle Objekte des Object Models und ist somit aufgrund der einfacheren Verwendung auch unter Windows als Ersatz zur COM API verwendbar. [SBl081]

4.5.2 Plattformabhängigkeit

Aufgrund der oben genannten Gegebenheiten war die Plattformabhängigkeit bei der Plug-In Entwicklung bis Version 3.0 von Softimage|XSI noch stärker ausgeprägt, als es bis dato der Fall ist. Durch die Einführung der C++ API wurde diese Problematik bezogen auf die XSI Umgebung weitgehend entschärft. Werden jedoch andere Bibliotheken verwendet, wie zum Beispiel DirectX unter Windows oder SDL *(Simple Direct Media Layer)* unter Linux, muss sämtliche Funktionalität des Plug-Ins auf die jeweilige Plattform angepasst werden.

Wird ein Plug-In unter Verwendung der C++ API entwickelt, besteht der Hauptunterschied in der Ausgabe des Compilers, der unter Windows eine DLL Datei und unter Linux eine SO Datei erstellt die von Softimage|XSI verwaltet werden können. Da es eine Vielzahl an unterschiedlichsten Linux Distributionen und somit unterschiedliche Versionen von Systembibliotheken gibt, werden die für das Plug-In benötigten Bibliotheken gleich zur SO-Datei gepackt. Dies wird „static link“ genannt und verhindert, dass der Benutzer des Plug-Ins aufwändig nach der für sein System Passenden Bibliothek suchen, installieren oder gar kompilieren muss. [SBl082]

4.5.3 Interpreter versus kompiliertem Code

Im Folgenden werden die Vor- und Nachteile von kompiliertem Code gegenüber einem Script aufgelistet.

Pro	Contra
Kompilierter C++ Code schneller bei geringerem Speicherverbrauch	Schwieriger zu lernen
Kompatibilität mit Bibliotheken von Drittanbietern	Strenge Kontrolle bei Datentypen
Code kann geheim gehalten werden	Teile des Objektmodells sind nicht direkt verfügbar in der C++ API
Code ist „mächtiger“: Zugriff auf Hardware und Betriebssystem möglich;	Kompilation verhindert direktes editieren; DLL muss immer neu kompiliert und geladen werden.
	Aufwändigere Konfiguration der Arbeitsumgebung notwendig

Tabelle 1 Pro/Contra kompilierte Plug-Ins [SBl081]

Des Weiteren sind unter Windows die vom Softimage|XSI Plug-In Wizard erstellen Projektdateien für Microsoft Visual C++ und somit für eine proprietäre Entwicklungsumgebung.

4.6 Scripting

Softimage|XSI verwendet durchgängig bereits vorhandene und weit verbreitete Scriptsprachen und verzichtet auf den Gebrauch einer eigenen Scriptsprache. Die Auswahl der Scriptsprache hängt dabei lediglich von den Präferenzen des Entwicklers ab. Zur Auswahl stehen dabei:

- JScript: JScript ist eine an JavaScript, 1995 von Netscape entwickelt, und Java, entwickelt von Sun Micorsystems im Jahre 1991, angelehnte Scriptsprache von Microsoft. Bis zu Version 5.6 war JScript vollständig kompatibel zu Softimage|XSI, seit der Version JScript .NET werden viele Erweiterungen und Erneuerungen jedoch nicht mehr unterstützt. Vorteile von JScript sind die Unterstützung einer COM basierten API, Zugriff auf das WScript.Shell das den Zugriff auf Dateien und Systemvariablen gibt oder die Ähnlichkeit zu Hochsprachen wie C++ mit der einhergehenden Mächtigkeit. Nachteile sind die Speicherverwaltung, das Fehlen von freien Bibliotheken sowie die mangelnde Weiterentwicklung der Sprache Seitens von Microsoft. [Bun09] und [SBl07]
- VB Script: VB Script wurde von Microsoft im Jahre 1995 entwickelt und ist eine Ableitung der Programmiersprache Visual Basic für Windows. VB Script wird automatisch mit Windows installiert und ist auch die Standardsprache für den Windows Scripting Host. Vorteile von VB Script sind die weite Verbreitung in der XSI SDK Dokumentation, die einfache Erlernbarkeit der Sprache und die Stabilität. Als Nachteilig gelten für erfahrene Programmierer der verwirrende Syntax (So ist zum Beispiel der „=“ Operator gleichzeitig zuweisend als auch vergleichend) sowie das schlechte Error-Handling. [Akt091] und [SBl071]
- Perl: Entwickelt wurde Perl im Jahre 1987 von Larry Wall und war eine Mischung aus C, AWK *(Eine spezielle, für Textverarbeitung entwickelte Sprache)* und Unix Befehlen. Zu den Stärken von Perl zählt vor allem der Umgang mit regulären Ausdrücken *(Beschreibung von Zeichenketten mittels syntaktischen angeordneten Symbolen)*. Als Nachteilig gilt die nachlässige Disziplin bei der Formatierung die einen Programmcode für Dritte oft unleserlich gestalten. [Per09]
- Python: Guido van Rossum entwickelte 1990 die Programmiersprache Python. Python besitzt, im Gegensatz zu JScript, VB Script und Perl eine Datentypisierung

(Variablen, die zum Beispiel Ganzzahlen, Gleitkommazahlen oder Zeichen beinhalten, müssen als solche definiert werden). Die Vorteile von Python sind die geringe Anzahl an Schlüsselwörtern *(Wörter die von der Programmiersprache reserviert sind und nicht als Variable verwendet werden können)*, die leichte Les- und Erlernbarkeit sowie die Vielzahl an freien Bibliotheken. Als nachteilig gilt die mangelnde Unterstützung für Windows in den 64 Bit Versionen. Damit Python verwendet werden kann, muss unter Windows eine eigene Version von Softimage|XSI geladen werden und die Python Unterstützung installiert werden. [Pyt09] , [SBl083] und [Tho08]

Um eine Scriptsprache in Softimage|XSI verwenden zu können, müssen diese alle ActiveX Scripting unterstützen. ActiveX wurde 1996 von Microsoft entwickelt und ist eine Technologie, die nach dem Component Object Model Arbeitet und hauptsächlich für Webanwendungen entwickelt wurde. ActiveX Controls sind kompilierte Komponenten die alleine jedoch nicht lauffähig sind. Ausgeführt werden diese in systemnahen ActiveX-Containern, welche auf Zugriff auf die Dateistruktur und Systemressourcen haben.

Unter Linux werden die ActiveX Bibliotheken automatisch mit der Installation von Softimage|XSI mit installiert. [Akt09]

4.6.1 Command Modell versus Object Modell

In Softimage|XSI gibt es zwei unterschiedliche Herangehensweisen an das Scripting, die je nach der Erfahrenheit des Programmierers beziehungsweise je nach den Ansprüchen an das Script ausgewählt werden können:

- Das Object Model: Das Object Model verwendet Eigenschaften und Methoden der Objekte die sich in der Szene befinden. Eine ausführliche Erläuterung dazu findet sich in Kapitel 4.3.
- Das Command Model: Beim Command Model werden die Befehle, die in Softimage|XSI ausgeführt werden können, eingesetzt. Die Vorgehensweise erinnert an das Erstellen eines Macros. Beim Abarbeiten einer Routine durch den Benutzer über das XSI Interface, zum Beispiel das Erstellen eines Würfels und das Extrudieren entlang der X-Achse, wird jeder Befehl in der Script-History gespeichert. So kann das Script kopiert und wieder verwertet werden. Der Vorteil des Command Models ist, dass es nicht zwingend notwendig ist, die Syntax und die

Befehle zu lernen, da diese ausgeführt und anschließend aus der History kopiert werden können. Als großer Nachteil erweist sich die Ausführungsgeschwindigkeit, da jeder Befehl einzeln ausgeführt und in der History abgelegt wird. Des Weiteren ist eine Wiederverwendbarkeit nicht garantiert, da die Objekte in der Szene immer über ihre Namen angesprochen werden. Tabelle 2 zeigt einen Vergleich der Eigenschaften des Command Models und des Object Models. [Mic01] und [And05]

Command Model	Object Model
+ Einfacher zu erlernen	+ Mächtiger
+ Gut für kurze, Macro ähnliche Scripten	+ Wiederverwendbar
- Langsam	+ Schneller, da direkte Kommunikation mit XSI Kern über COM/ActiveX
	- Komplexer

Tabelle 2 Command Model vs. Object Model

Das folgende kurze Beispiel zeigt zum einen die Verwendung des Command Models und zum anderen das Object Model. Im Beispiel wird ein Würfel erstellt und die Kantenlänge auf zwei eingestellt.

Command Model Version:

```
CreatePrim("Cube", "MeshSurface", null, null);
SetValue("cube.cube.length", 2, null);
```

Object Model Version (in JScript):

```
var ob = ActiveSceneRoot.AddGeometry("Cube", "MeshSurface");
ob.length = 2;
```

4.7 Die Erweiterungen und Scriptingmöglichkeiten im Detail

Scripting kann in Softimage|XSI für eine Vielzahl an Einsatzmöglichkeiten verwendet werden beziehungsweise gibt es für gewisse Bereiche, wie zum Beispiel Pipelinescripting, kaum andere Lösungswege.

Das Entwickeln von Scripten erfolgt dabei direkt in Sofitmage|XSI, im so genannten Script Editor (Abb. 13). Der Script Editor besteht aus einem Editor-Fenster (weisser Hintergrund) und einem History-Fenster (grauer Hintergrund), in dem die Konstruktions-History in Form des Command Models und die Script-Ausgabe, sofern vorhanden, geloggt wird. Es gibt mehrere Möglichkeiten die Scripten auszuführen. Die einfachste Methode erfolgt über den Ausführ-Button direkt im Script Editor. Die Scripten können auch über eigene Menüs in die Oberfläche von Softimage|XSI eingebunden werden. Scripten können auch automatisch ausgeführt werden, zum Beispiel bei der Selektion eines bestimmten Objektes. Eine ähnliche Methode ist das Binden eines Scriptes an ein Aktion, zum Beispiel das automatische Speichern der Szene beim Freezen eines Models. Zusätzlich können Scripten auch im so genannten Synoptic View *(ein auf HTML basierendes, frei erstellbares Layout das Controller für die Animation bietet)* und im Net View *(Ein vollwertiger Internetbrowser innerhalb des Programms, der auch Daten in Softimage|XSI schreiben und lesen kann)* eingebunden werden. [Sof08] und [XSI08]

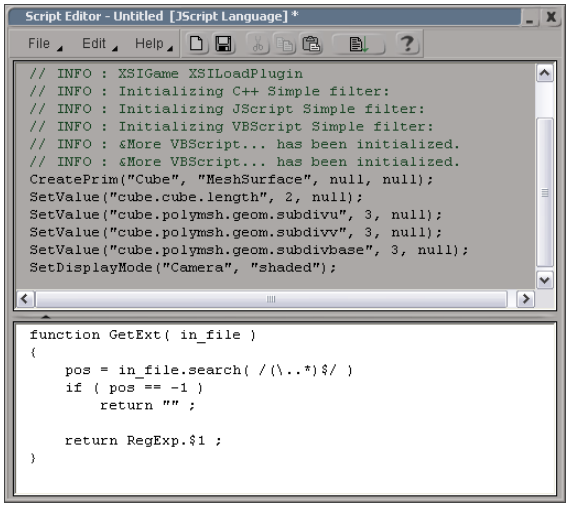

Abbildung 12 Script Editor

Da es sich beim Script Editor nicht um eine integrierte Entwicklungsumgebung handelt, ist der integrierte Debugger *(Programm zur Kontrolle und Inspektion von Software; Werte von Variablen können zum Beispiel während der Laufzeit ausgelesen*

und verändert werden) von Softimage|XSI recht schwach und ermöglicht lediglich das Setzen von Haltepunkte im Script. Der Script Editor ermöglicht jedoch das Einbinden von externen Werkzeugen zum Debuggen. [Sof08]

In diesem Kapitel 4.7 werden die Scriptingmöglichkeiten in Softimage|XSI genauer erörtert und mit einem Beispiel verdeutlicht.

4.7.1 Pipeline Scripting

Bei heterogenen Produktionen, also solche bei denen unterschiedlichste Software zum Einsatz kommt, ist es oft notwendig, eine Szene oder einen Effekt in unterschiedlichen Programmen zu bearbeiten, die jeweils auf unterschiedliche Schwerpunkte ausgelegt sind. Ein Beispiel: Ein möglichst detailreicher menschlicher Kopf soll erstellt werden. In Softimage|XSI wird grob die Form eines menschlichen Kopfes modelliert. Anschließend wird das Modell exportiert und zum Modellieren der Details in ZBrush *(verändert die Geometrie basierend auf dem Tiefenwert des digital aufgemalten Pixels)* geöffnet. Das hier erstellte Displacement Map *(eine Reliefdarstellung)* wird wieder in Softimage|XSI geöffnet und zusammen mit einer Textur auf das grobe Modell gelegt. Dieser Vorgang wird als Production-Pipeline bezeichnet. [Dig05]

Nicht immer werden alle Formate in den jeweiligen Softwarepackages unterstützt. Deshalb müssen manche Formate (wie zum Beispiel Autocad DXF oder Initial Graphics Interchange Specification IGES) per Plug-In in ein passendes Format konvertiert werden.

Nicht nur Formatumwandlung kommt in einer Production-Pipeline vor, auch Automatismen zur schnelleren Weiterverarbeitung sind ein häufiges Einsatzgebiet für Pipeline Scripting. [Dig05] und [Tic08]

4.7.2 Batch Scripting

Bei Softimage|XSI ist es möglich, das Programm aus einer Kommandozeilenumgebung laufen zu lassen, ohne dabei die grafische Benutzeroberfläche laden zu müssen. Dies ermöglicht ein schnelles abarbeiten von Batch Prozessen, wie zum Beispiel das Rendern von Szenen. Aus diesen Batch Prozessen können auch Scripte in XSI gestartet werden beziehungsweise können Parameter an diese übergeben werden.

Bei dem über den Batch Prozess gestarteten Script muss nicht komplett auf eine grafische Benutzeroberfläche verzichtet werden. Es lassen sich eigene, auf dem XSI Kern basierende, grafische Oberflächen gestalten und anzeigen, ohne die gesamte, Ressourcen verbrauchende Oberfläche von Softimage|XSI zu laden. [SBl06]

4.7.3 Erweiterungen für Szenenelemente und XSI Features

Die hier vorgestellten Erweiterungen und Plug-Ins stellen nicht alle in Softimage|XSI möglichen Erweiterungen dar. Die hier erwähnten Erweiterungen können alle über den Wizard im Plug-In Manager erstellt werde.

Neben diesen Erweiterungen gibt es noch eine Vielzahl anderer Möglichkeiten Softimage|XSI anzupassen:

- Shader: Jede Geometrie in Softimage bekommt automatisch einen Shader zugewiesen. Ein Shader definiert dabei die unterschiedlichsten Materialeigenschaften eines Objektes, wie zum Beispiel Glanzlichter, Farbe oder Oberflächenstruktur. Neben einfachen Material Shadern gibt es eine Vielzahl unterschiedlichster Shader, wie Volume Shader, die Nebel-Artige Strukturen simulieren, oder Geometrie Shader, die eine an das Modell angepasste, aber nicht ausmodellierte Geometrie simuliert. Entwickelt werden die Shader in C++. [Raa05]
- Custom Display Host: Beim Display Host handelt es sich um eine Engine zur Darstellung der 3D Objekte in Softimage|XSI. Hier können auch eigene Engines eingebunden werden, zum Beispiel eine Game Engine für die Spielentwicklung. Entwickelt werden Display Hosts in C++.
- Custom FX Operator: Softimage|XSI ermöglicht Compositing direkt im Programm selbst. Im so genannten FX Tree können verschiedene Quellen, wie zum Beispiel der Alpha Kanal und der Tiefen Kanal, zusammen composed werden. Die im FX Tree verwendeten Operatoren kann der Benutzer auch selbst erstellen und werden in C++ entwickelt.
- Renderer: Neben MentalRay bietet Softimage|XSi die Möglichkeit eigene Render Engines zu entwickeln und einzubinden. Die Entwicklung erfolgt ebenfalls in C++.
- Treiber: Wird eine externe Hardware angeschlossen, die von Softimage|XSI nicht nativ unterstützt wird, können eigene Schnittstellen basierend auf der XSI API und

der API der angeschlossenen Hardware entwickelt werden. Die Entwicklung erfolgt in C++. [Sof08]

Die folgenden Erweiterungen werden über den Plug-In Manager erstellt und verwaltet.

4.7.3.1 Property

Custom Properties bestehen aus einem Set von benutzerdefinierten Parameteren mit dem dazugehörenden grafischen Interface, der Property Page. Die Proxy Properties sind im Gegensatz dazu eigene Property Pages, die ein Set von ausgewählten, jedoch einem Objekt zugehörenden, Parametern steuern. [XSI08]

Es gibt zwei unterschiedliche Möglichkeiten eine Custom Property zu erstellen. Zum einen direkt über das XSI Menü *ANIMATE>CREATE:PARAMETER>NEW CUSTOM PARAMETER SET* das eine leere Property Page erstellt, die aus dem gleichen Menü mit Parametern versehen werden kann und zum anderen über den Property Wizard im Plug-In Manager. Die Verknüpfung der Parameter erfolgt auf mehrere Arten: [XSI08]

- Parameter verlinken über den Parameter Connection Editor: Ein grafischer Editor erlaubt das verknüpfen von Parametern.
- Expressions: Über den Expression Editor können Expressions zu den Parametern verlinkt werden.
- Custom Operators: Beim Erstellen von Custom Operators können Parameter direkt auf die Custom Properties gelinkt werden.
- Scripting: Custom Properties können direkt aus einem Script erstellt und verlinkt werden um die Parameter des Scriptes grafisch steuern zu können. [XSI08]

Das folgende Beispiel zeigt, wie eine Custom Property erstellt und verwendet werden kann.

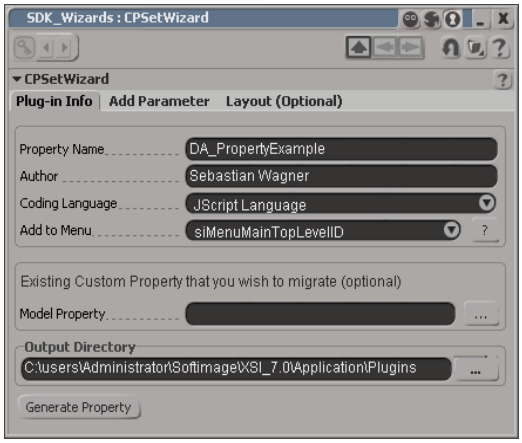

Abbildung 13 Custom Property Wizard

Im Plug-In Manager wird der Custom Property Wizard (Abb. 14) aufgerufen. Hier wird der Name, Autor und die Scriptsprache festgelegt. Zusätzlich kann die Custom Property zu einem Menü verlinkt werden. In diesem Beispiel wird JScript als Scriptsprache verwendet und die Property in das Top Level Menü eingebunden.

Optional kann in diesem Wizard eine bereits bestehende Custom Property in eine neue eingebunden werden. Gespeichert werden Custom Properties, falls nicht anders angegeben, unter dem Plug-In Ordner des angemeldeten Benutzers.

Unter dem Reiter „Add Paramter“ (Abb. 15) können die benötigten Parameter hinzugefügt werden. Es werden dabei die Datentypen unterschieden. In diesem Beispiel wird nur ein ganzzahliger Parameter hinzugefügt.

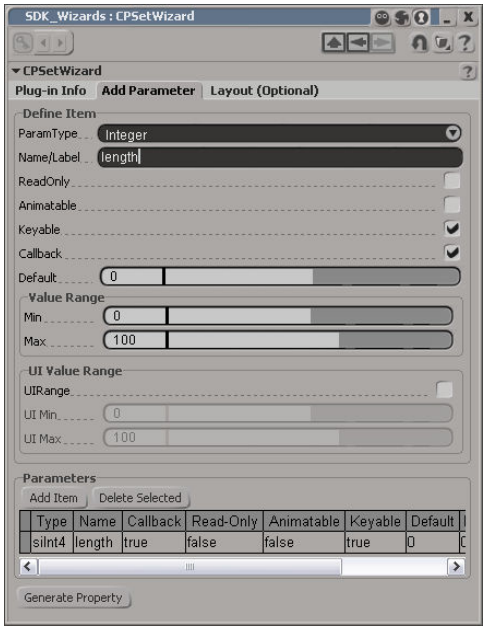

Abbildung 14 Custom Property Wizard - Parameter hinzufügen

Mit „Generate Property“ wird das Grundgerüst des Scriptes erstellt und kann nun angepasst werden. Das Custom Property soll dazu dienen, die Kantenlänge eines selektierten Würfels über den eigens definierten Parameter zu steuern. Das Code-Beispiel zeigt den Funktionsblock der ausgeführt wird, wenn der Parameter der Custom Property verändert wird. Für jeden Parameter wird dabei ein eigener Funktionsblock erstellt.

```
1: function DA_PropertyExample_MyLength_OnChanged( )
2: {
3:  Application.LogMessage("DA_PropertyExample_MyLength_OnChanged
called",siVerbose);
4:  var oParam = PPG.MyLength;

5:  if (Selection.Count == 0){
6:      LogMessage("Please select the Cube Primitive");

7:  }
8:  else {
9:      Selection(0).length = oParam;
10: }
11: }
```

Abbildung 15 Die Custom Property

In Zeile 4 wird der Variable oParam der Wert des Parameters mit dem Namen MyLength zugewiesen. Von Zeile 5 bis Zeile 10 wird überprüft, ob der Benutzer ein Objekt selektiert hat und im Falle einer positiven Überprüfung, der Wert von oParam an die Eigenschaft Kantenlänge übergeben. Der blau gefärbte Code ist dabei der vom Benutzer erstellte Code und der schwarze der vom Wizard erstellte Code.

4.7.3.2 Commands

Ein Custom Command ist ein Script oder Plug-In mit einer benutzerdefinierten Funktion. Dieser kann an Menüs und Tastaturkürzel gebunden werden und auch von anderen Scripts aufgerufen werden. [Sof08]

Erstellt wird ein Custom Command auf die gleiche Art und Weise, wie eine Custom Property. Dabei wird der Name des Plug-Ins, Autor und Scriptsprache festgelegt sowie das Custom Command in einem Menü abgelegt (Siehe Property 4.7.3.1).

Im folgenden Beispiel wird ein Null-Objekt erstellt und ein ICE Tree mit einem Particle Emitter an das Null-Objekt gebunden. Das Beispiel ist in JScript und verwendet eine Mischung aus Object Model und Command Model. Das Code-Beispiel zeigt den Funktionsblock der Ausgeführt wird, wenn der Benutzer das Custom Command ausführt.

```
1: function DA_CommandExample_Execute(  )
2: {

3:  Application.LogMessage("DA_CommandExample_Execute called",siVerbose);
4:  GetPrim("Null", null, null, null);
5:  EmitPointsFromObject("null");
6:  return true;

7: }
```

Zeile 4 erstellt ein Grundobjekt und Zeile 5 setzt das Null-Objekt als Particle Emitter, was automatisch mit einem Erstellen eines ICE-Trees verbunden ist. Der blau gefärbte Code ist dabei der vom Benutzer erstellte Code und der schwarze der vom Wizard erstellte Code.

4.7.3.3 Filter

Bei einem Custom Filter handelt es sich um einen benutzerdefinierten Filter. Ein Filter ermöglicht das gefilterte Selektieren von Objekten in der Ansicht beziehungsweise eine gefilterte Darstellung im Explorer. Die Filter werden automatisch beim „Select"-Menü in der Toolbar des Layouts und in der Suchleiste des Explorers eingebunden.

Bei der Erstellung über den Plug-In Wizard wird neben Name, Programmiersprache und Autor auch die Art des Filters festgelegt. Es wird hier zwischen unspezifischen Objekten, 3D Objekten, Properties, Kanten, Punkte, Polygone, und Knoten *(bei Bezier-Splines gesetzte Kontrollpunkte)* unterschieden. [Sof08]

Das folgende Beispiel in JScript filtert die Null-Objekte aus einer Selektion. Das Code-Beispiel zeigt den Funktionsblock der ausgeführt wird, wenn der Benutzer einen Filter auswählt.

```
1: function DA_FilterExample_Match( in_ctxt ){

2:    Application.LogMessage("DA_FilterExample_Match called",siVerbose);

3:    var obj = in_ctxt.GetAttribute("Input");
4:    return obj.IsClassOf(siNullID);
5: }
```

Die Funktion ist kontextabhängig, das heißt, es macht einen Unterschied ob der Filter im Explorer oder im View ausgeführt wird. Dies geschieht automatisch und es wird lediglich

ein Input verlangt, wie zum Beispiel eine Selektion oder der Tree im Explorer. Dies geschieht in Zeile 3, wo die Objekte der Variable „obj“ zur weiteren Verarbeitung übergeben werden. Bei der Variable „obj“ handelt es sich nicht um eine Collection, sondern ein einfaches Objekt, da die Funktion für jedes Objekt einzeln ausgeführt wird. In Zeile 4 wird überprüft ob das Objekt ein Null-Objekt ist und dementsprechend ein „true“ oder „false“ zurückgeliefert. Bei einem „true“ erscheint das Objekt im View als selektiert beziehungsweise wird das Objekt im Explorer aufgelistet.

Der blau gefärbte Code ist dabei der vom Benutzer erstellte Code und der schwarze der vom Wizard erstellte Code.

4.7.3.4 Events

Bei jeder Aktion des Benutzers, wie zum Beispiel das Erstellen einer neuen Szene, wird ein Event getriggert. Dieser Event lässt sich anpassen beziehungsweise können eigene Events erstellt werden.

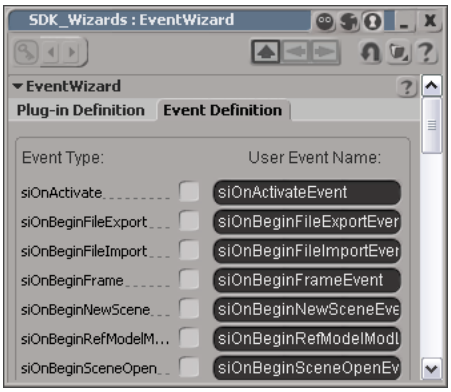

Abbildung 16 Event Wizard - Event Definition

Neben den Angaben über Name, Scriptsprache und Autor des Plug-Ins muss unter dem Reiter „Event Definition“ mindestens ein Event ausgewählt werden, das getriggert wird bevor das Grundgerüst des Events erstellt werden kann. Dabei gibt es eine Vielzahl an unterschiedlichsten Events, vom Tastendruck über das Speichern der Szene bis hin zu periodischen Events, die hier ausgewählt werden können (Abb. 17). [Sof08]

Das folgende Beispiel in JScript „freezed“ *(Die Operatoren, die während des Modeling verwendet wurden, werden in fixe Werte konvertiert und können anschließend nicht mehr bearbeitet werden.)* die Modelle in der Szene bevor die Szene geschlossen wird.

```
1: function siOnCloseSceneEvent_OnEvent( in_ctxt )
2: {
3:      Application.LogMessage("siOnCloseSceneEvent_OnEvent
4:                              called",siVerbose);
5:    FreezeModeling(null, null, null);
6:    LogMessage("All models freezed");
7:    return true;
8:}
```

Da die Funktion keine sofort ersichtlichen Ergebnisse hat, wird zusätzlich eine Meldung mitgeloggt, dass die ausgeführte Aktion bestätigt (Zeile 6). In Zeile 5 wird die Freeze-Aktion ausgeführt.

Der blau gefärbte Code ist dabei der vom Benutzer erstellte Code und der schwarze der vom Wizard erstellte Code.

4.7.3.5 Custom/Scripted Operators

Ein Operator verändert die ihm angegebenen Daten und gibt das Ergebnis aus, wie zum Beispiel ein Knitter-Operator: Als Eingangsdaten dienen die Polygone eines Geometrieobjekts, die dann im Operator durch einen Algorithmus geteilt und verdreht werden.

Es werden vier verschiedene Typen von Operatoren unterschieden:

- Generatoren: Der Operator erzeugt eine Geometrie.
- Deformer: Die Geometrie der Objekte wird verändert.
- Constraints: Der Operator verändert die globale Position eines Objektes oder beeinflusst deren Parameter.
- Topologie: Der Geometrie werden Punkte oder Kanten hinzugefügt. [Sof08]

Es gibt zwei unterschiedliche Arten von benutzerdefinierten Operatoren. Zum einen den Custom Operator, der über den Plug-In Wizard erstellt wird und zum anderen den Scripted Operator, der direkt in der Szene erstellt wird. Der Vorteil des Scriped Operator ist, dass sich der Benutzer nicht um das Verwalten des Operators im Plug-In Manager kümmern muss. Das Script kann bequem in einem extra dafür vorgesehenen Editor erstellt werden, der dem Benutzer nützliche Werkzeuge und ein übersichtliches Interfaces bietet. Der Nachteil ist, dass der Operator so nicht an Dritte weiter gegeben werden kann und da Operatoren animierbare Parameter besitzen, beeinflussen gescriptete Operatoren auch die Ausführungsgeschwindigkeit. Soll ein Operator wiederverwendet werden, empfiehlt die XSI Dokumentation daher die Portierung eines Scripted Operator in ein kompiliertes Plug-

In. Nichts desto trotz lassen sich Custom Operators auch in einer Scriptsprache als Plug-In erstellen. Wie auch schon bei den anderen Plug-Ins geschieht das über den Wizard im Plug-In Manager. Es müssen hier wieder der Name des Plug-Ins, Autor und Scriptsprache ausgewählt werden. Zusätzlich müssen Ein- und Ausgänge definiert werden (zum Beispiel die globale Position eines Objektes als Eingang und der Winkel der Kanten eines anderen Objektes zum Objektmittelpunkt als Ausgang). Des Weiteren können Parameter angegeben werden, das Layout der Property Page angepasst werden und bestimmt werden, ob Funktionsblöcke für die Initialisierung und Terminierung des Operators erstellt werden sollen. [Sof08]

Das folgende Beispiel zeigt die Erstellung eines Scripted Operators in JScript, da der Funktionsblock des Operators derselbe ist, wie beim Custom Operator. Im Beispiel soll ein Zylinder, der als Rad dient, einen Pfad entlang rollen. Der Operator sorgt dafür, dass sich das Rad entsprechend dem zurückgelegten Weg dreht.

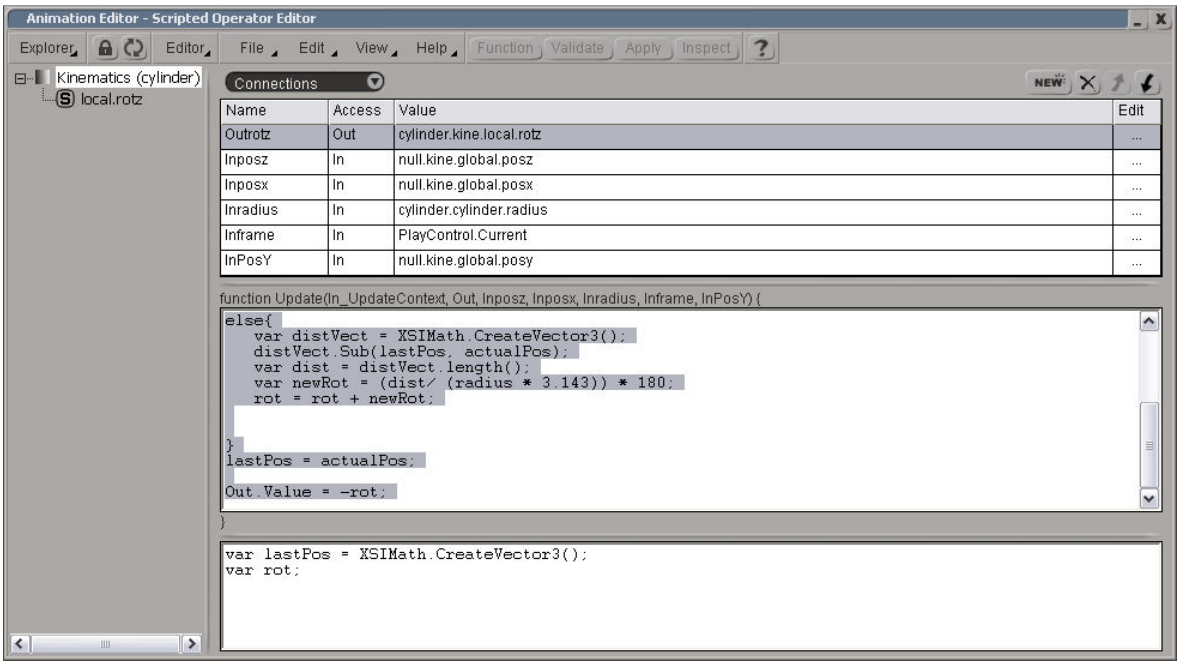

Abbildung 17 Scripted Operator Editor

Die Szene enthält eine Spline als Pfad, einen Zylinder als Rad und ein Null-Objekt als Steuerung.

Abbildung 18 zeigt den Editor für den Scripted Operator. Der Explorer auf der linken Seite zeigt die Parameter die das Ergebnis des Operators erhalten. Die Tabelle listet die Ein- und Ausgänge des Operators auf. Benötigt wird die globale Position des Null-Objekts das dem Pfad folgt und später den Zylinder steuert. Zudem werden der Radius des Zylinders und

der aktuelle Frame eingelesen. Der Rotations-Wert des Zylinders in der Z-Achse erhält die Ausgabe des Operators.

Das Obere der zwei Editor-Fenster enthält den Code, der ausgeführt wird, wenn der Operator ausgeführt beziehungsweise aktualisiert wird. Das zweite Fenster enthält Code der die Hauptfunktion unterstützt. Dies ist jedoch nicht immer notwendig.

Hauptfunktion:

```
1: var frame = Inframe.value;
2: var actualPos = XSIMath.CreateVector3();
3: var radius = Inradius.Value;

4: actualPos.Set(Inposx.Value,InPosY.Value,Inposz.Value);

5: if(frame < 2){

6:  rot = 0;
7: }
8: else{
9:      var distVect = XSIMath.CreateVector3();
10:     distVect.Sub(lastPos, actualPos);
11:     var dist = distVect.length();
12:     var newRot = (dist/ (radius * 3.143)) * 180;
13:     rot = rot + newRot;


14: }
15: lastPos = actualPos;

16: Out.Value = -rot;
```

Behelfs Fenster:

```
1: var lastPos = XSIMath.CreateVector3();
2: var rot;
```

Von Zeile 1 bis 3 werden die Werte der Eingänge an die Variablen übergeben beziehungsweise diese initialisiert. Zuerst wird der aktuelle Frame der Zeitleiste übergeben. Die momentane Position wird als 3D Vektor initialisiert und anschließend der Radius des Zylinders an die Variable übergeben. In Zeile 4 wird der 3D Vektor mit den aktuellen Werten gefüllt. Von Zeile 5 bis 7 wird überprüft, ob der momentane Frame über 1 steht, und somit eine Rotation vorhanden ist. In Zeile 9 wird erneut ein 3D Vektor erstellt, der zur Berechnung der Distanz zwischen der aktuellen und der letzten Position benötigt wird. In Zeile 10 findet die eigentliche Berechnung statt und Zeile 11 übergibt den skalaren Wert der Berechnung an die Variable „dist“. In Zeile 12 findet die Berechnung des Rotationswertes statt und wird in Zeile 13 zum letzten Rotationswert hinzugefügt. In Zeile 15 übernimmt die Variable für die letzte Position die Werte der aktuellen Position,

die im nächsten Frame als Ausgangsposition dient. In Zeile 16 wird schlussendlich der Rotationswert an den Ausgang des Operators und somit an den Rotationswert des Zylinders übergeben.

Der Code im Behelfs Fenster initialisiert die Variable für die letzte Position des Zylinders als 3D Vektor und die Rotationsvariable.

4.7.3.6 Compilierte ICE Nodes

ICE Nodes sind Funktionseinheiten und Operatoren in ICE Scripts (Siehe Kapitel 5). Benutzerdefinierte Nodes können über den Plug-In Manager erstellt werden. Der Unterschied zu den anderen Plug-In Wizards im Manager ist, dass ICE Nodes nur in C++ entwickelt werden können und somit einen Compiler benötigen.

Beim Erstellen des ICE Nodes müssen wieder Name des Nodes und der Autor angegeben werden. Als Programmiersprache steht nur C++ zur Auswahl. Zudem muss eine Kategorie angegeben werden, unter der der ICE Node abgelegt wird. Per Default ist hier „Custom ICE Nodes“ angegeben, was dem Menüeintrag in der Node Auswahl im ICE Editor entspricht. Zusätzlich kann angegeben werden, ob der ICE Node Singlethreading unterstützen soll, da dies das Ansprechen der gesamten Eingangsdaten ermöglicht (Siehe Kapitel 5). Im Reiter Input Ports und Output Ports werden die Ein- und Ausgänge des Nodes definiert. Hier wird der ICE Node Datentyp angegeben, die Struktur der Daten (zum Beispiel Arrays) und der Kontext (zum Beispiel die komplette Geometrie oder jeder einzelne Punkt der Geometrie) angegeben. Soll der Ein- oder Ausgang mit einem anderen verknüpft sein, kann das hier ausgewählt werden. Ein- und Ausgänge können hier über die Group ID in Gruppen zusammengefasst werden, was das spätere Arbeiten mit den Nodes durch eine bessere Übersicht erleichtert. [Sof08]

Auf ein Beispiel mit Programmcode soll hier verzichtet werden, da die Auseinandersetzung mit dem Code ein umfangreiches Vorwissen verlangt und den Rahmen eines Überblicks sprengen würde. Für den interessierten Leser sei an dieser Stelle auf die SDK Dokumentation von Softimage|XSI verwiesen.

5 XSI ICE

5.1 Überblick

Mit Version 7.0 von Softimage|XSI wurde das Interactive Creative Environment, kurz ICE, eingeführt. Es handelt sich bei ICE um ein visuelles, prozedurales Programmiersystem zum Erstellen und Kontrollieren von Simulationen und Deformationen. ICE liefert zudem ein eigenes Partikel System, zusätzlich zum bereits bestehenden System von XSI. Grundsätzlich bietet ICE Funktionalität, die in XSI vorhanden ist und stellt sie als Sammlung von Nodes und Node-Gruppen, so genannten Compounds, welche wiederum als einzelne Nodes dargestellt werden, zur Verfügung.

Eine der Stärken von ICE liegt in der Unterstützung von Mehrkernprozessoren beziehungsweise der Einsatz von Multithreading. Softimage|XSI ermittelt dabei Datenpakete die in den Nodes bearbeitet werden und verteilt diese auf die vorhandenen Prozessoren, was ein äußerst effizientes Arbeiten ermöglicht und oft schneller ist als kompilierter Code.

Beim Erstellen eines ICE Scriptes kann sich der Benutzer komplett auf die Logik des Algorithmus konzentrieren ohne dabei auf Syntaxfehler zu achten, da ICE ein in sich geschlossenes System bildet. [Aut09] und [Sof08]

5.2 Komponenten eines ICE Scripts

In diesem Kapitel sollen die Komponenten eines ICE Script erläutert werden und deren Eigenschaften aufgezeigt werden.

5.2.1 Node

Ein Node, oder auch Knotenpunkt, ist die kleinste Einheit eines ICE Scriptes und repräsentiert eine abgeschlossene Funktionseinheit mit mindestens einem Eingang und/oder einem Ausgang zum Austauschen von Daten.

Ein ICE Script besteht aus Nodes mit unterschiedlichen Funktionen, wie zum Beispiel einer Multiplikation oder das Lesen von Daten. Ein Node hat, je nach Funktion, eine Property Page zum Steuern der Werte. Verknüpft werden Nodes über deren Ports, also Ein-

und Ausgänge. Die Ports unterscheiden Datentypen wie Arrays oder Ganzzahlen. Dem Benutzer wird das Erkennen der Kompatibilität zwischen den Nodes erleichtert durch eine Farb-Codierung der Ein- und Ausgänge. Ein erzwungenes Verbinden inkompatibler Nodes durch den Benutzer ist nicht möglich. [XSI08]

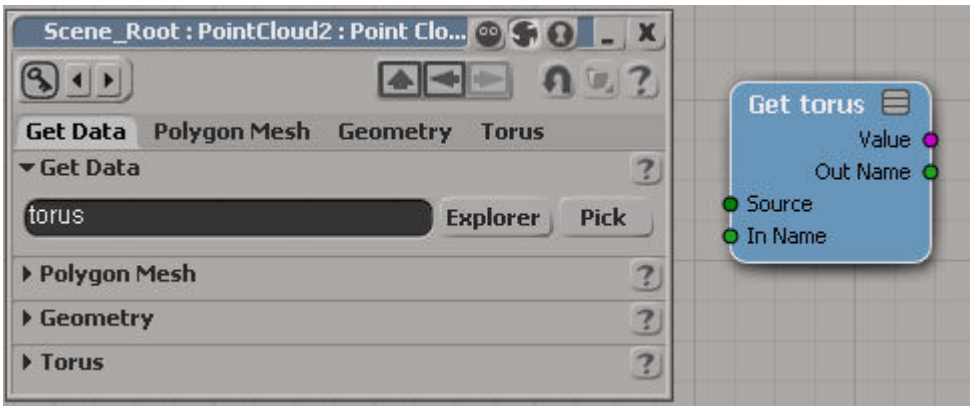

Abbildung 18 ICE Node "Get Data" mit geöffneter Propertypage

5.2.2 Compound

Ein ICE Compound besteht aus mehreren ICE Nodes beziehungsweise anderen Compounds, die zusammen eine Funktion bilden. Compounds können verwendet werden, um komplexe ICE Scripten übersichtlicher zu gestalten oder erstellte Funktionen zu exportieren, um sie in anderen Scripten erneut zu verwenden oder an Dritte weiter zu geben. Gespeichert werden Compounds in einer XML Datei, die alle notwendigen Daten enthält, wie Verbindungen, Ein- und Ausgänge sowie die Werte der Nodes. Sollen Compounds kommerziell an Dritte weitergegeben werden, können diese auch gesperrt exportiert werden, um zu verhindern dass Compounds geöffnet und analysiert werden können. [XSI08]

Die Ein- und Ausgänge eines Compounds können vom Benutzer definiert werden beziehungsweise können nur jene Ports erstellt werden, die auch an den Nodes innerhalb des Compounds noch nicht belegt sind. Diese Ports können mit Filtern belegt werden um maximale und minimale Werte auszufiltern. Des Weiteren kann angegeben werden, ob der eingehende Wert mehrfach verwendet wird, also ob der Port an mehrere Nodes weitergeleitet wird. Abbildung 20 zeigt den von Softimage|XSI mitgelieferten Compound „Footprints“, dass das Simulieren von Fußspuren realisiert. Der rote Node zeigt den

Compound in der ICE Tree Ansicht. Der graue Rahmen zeigt an, dass sich der Benutzer innerhalb des Compounds befindet. An der linken und rechten Seite liegen die Ein- und Ausgänge des Compounds. [XSI08]

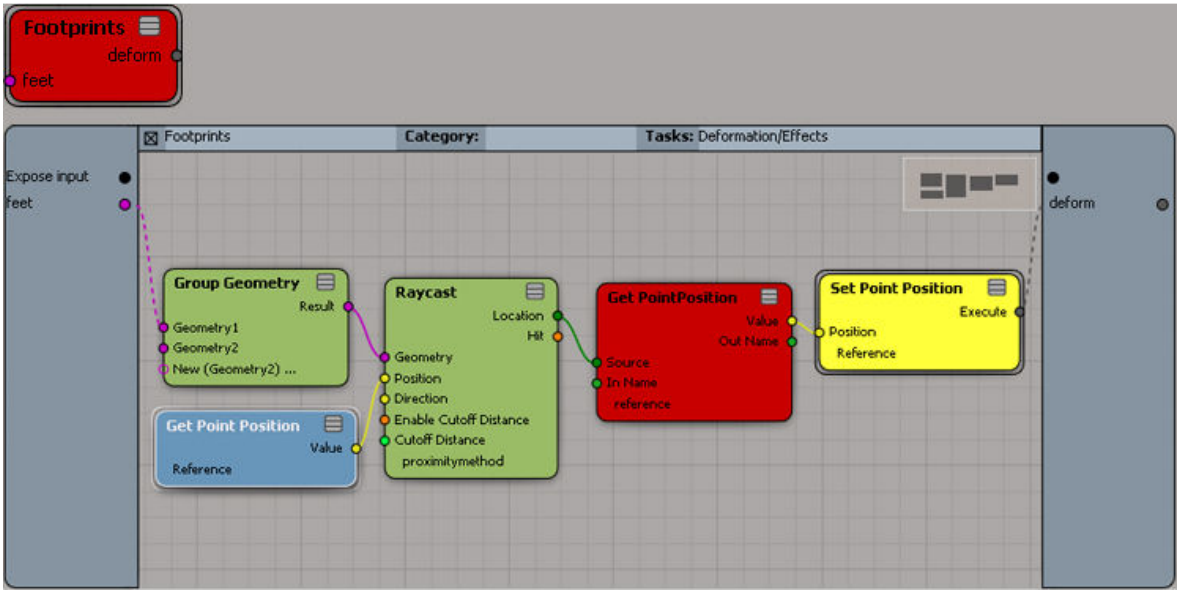

Abbildung 19 ICE Compound "Footprints"

5.2.3 ICE Tree

Der ICE Tree ist das Herzstück eines ICE Scriptes und befindet sich immer am Ende des Datenflusses eines ICE Scripts und hat nur Eingänge, an die die ausführbaren Operationen des ICE Scripts angehängt werden können. Ein funktionierendes ICE Script besteht also aus mindestens zwei Nodes: einem Funktionsnode und dem ICE Tree Node. Beim Erstellen eines ICE Scriptes muss immer zuerst der ICE Tree angelegt werden. Dabei wird zwischen simulierenden und nicht simulierenden ICE Trees unterschieden. Die Erläuterung der genauen Funktionsweise findet sich im folgenden Punkt 5.3. [XSI08]

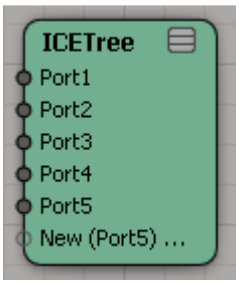

Abbildung 20 Der ICE Tree

5.3 Grundlegende Funktionsweise

Ein ICE Script fällt unter die Kategorie eines Operators und ist im Operator-Stack *(Die Liste der am Objekt ausgeführten Operatoren. Es werden verschiedene Modi unterschieden: Modeling, Shape Modeling, Animation, Simulation, Post-Simulation und Secondary Shape Modeling. Beim Bearbeiten des Objektes kann in diese Modi gewechselt werden und dementsprechend werden die Operatoren auch unter dem entsprechenden Modus im Operator Stack abgelegt. So kann ein Operator für z.B.: die Animation erhalten bleiben während die Operatoren beim Modeling gefreezed werden.)* eines Objektes sichtbar. Folglich kann ein ICE Script nur erstellt werden, wenn auch ein Objekt selektiert wurde. Da es sich um einen Operator handelt, darf das Model nicht gefreezed werden, wenn sich das mit einem ICE Tree Operator versehene Objekt noch in der Modellierungsphase befindet, da mit einem Freezen des Models auch der ICE Tree gefreezed wird und somit das ICE Script nicht mehr editier- beziehungsweise sichtbar ist. Ein Objekt kann aufgrund dieser Tatsache mehrere ICE Trees besitzen. Dabei spielt die Reihenfolge der ICE Trees im Operator Stack eine Rolle, da diese der Reihe nach, von unten nach oben, ausgeführt werden, so wie dies auch bei herkömmlichen Operatoren, wie Knitter und Biege Operatoren, der Fall ist. [XSI08]

Der ICE Tree wirkt auf jeden Punkt des Geometrie Objekts, sofern im ICE Script keine Filter eingebaut wurden.

5.3.1 Simulierende und nicht-simulierende ICE Trees

ICE Trees lassen sich in simulierende und nicht simulierende Trees unterteilen. Der Unterschied liegt dabei lediglich in der Position des Trees im Operator Stack. Liegt ein ICE Tree in der Simulationsregion, so ist der Eingangswert des Operators nicht das 3D Objekt oder das Ergebnis eines vorhergehenden Operators, sondern das Ergebnis des gleichen Operators zu einem früheren Zeitpunkt, in diesem Fall des vorhergehenden Frames. Nicht simulierende ICE Trees werden standardmäßig in der Modellierungsregion abgelegt. [XSI08]

Des Weiteren können ICE Scripten grob in zwei Arten von Anwendungsbereichen unterschieden werden: Deformer und Partikel.

ICE Trees, die rein als Deformationsoperatoren verwendet werden, werden automatisch in der Modellierungsregion abgelegt. Der ICE Tree befindet sich dabei immer im Operator Stack des zu deformierenden Objektes. [XSI08]

Bei ICE Partikeln handelt es sich nicht um eine Particle Cloud, so wie dies beim herkömmlichen Partikel System von Softimage|XSI der Fall ist, sondern um eine Point Cloud mit Punkten, wie sie jede Geometrie enthält. Der Unterschied ist, dass die Punkt-Positionen simuliert werden. Der ICE Tree wirkt beim Simulieren von Partikeln auf die Point Cloud und nicht auf das emittierende Objekt. Es können aber auch nicht simulierende ICE Trees auf „Point Clouds“ angewendet werden. [XSI08]

5.3.2 Einlesen und Manipulieren von Daten

Eine Vielzahl an zur Verfügung stehenden Funktionen für den ICE Tree sind Compounds, die von Softimage bereitgestellt werden. Diese lassen sich alle in ihre kleinsten Funktionseinheiten, Nodes, auflösen. In diesem Kapitel sollen die wichtigsten Nodes erörtert werden, die zum Einlesen und Manipulieren von Daten aus einer Szene benötigt werden. Grundlegende Compounds werden in Kapitel 6 zu den praktischen Beispielen erläutert.

5.3.2.1 Get Data, Set Data

Der „Get Data“ Node kann grundsätzlich jede in der Szene Root befindliche Eigenschaft (siehe Object Model) auslesen. Da die Selektion des Eingangswertes über einen Objekt Explorer erfolgt, werden Eigenschaften, die für die Verwendung in ICE gedacht sind, in einer Liste hervorgehoben. Die in der Liste angeführten Eigenschaften werden ICE Attributes genannt. Der Benutzer kann die benötigte Eigenschaft auch von Hand eingeben und somit Eigenschaften auslesen, die nicht in dieser Liste aufscheinen. Eine Kompatibilität der eingelesenen Werte mit den ICE Operatoren ist hier jedoch nicht mehr gewährleistet, da jede Form von Datentyp eingelesen werden kann. So kann zum Beispiel die Objekt-Kategorie eines Würfels nicht in einem ICE Tree verwendet werden:

```
cube.Categories
```

die Eigenschaft „Categories“ des Würfels liefert ein String Array zurück mit dem Inhalt „General, Primitve, Modeling“. Diese Werte können von ICE nicht verarbeitet werden, da keine Nodes vorhanden sind, mit der diese Werte bearbeitet werden könnten.

Der „Set Data“ Node ist eigentlich ein Compound. Der ihm zugrunde liegende Node ist der „Set one data“ Node, der genau einen Parameter setzt. Beim „Set one data“ Node lassen sich alle Eigenschaften, die ICE verarbeiten kann, setzen. Ausnahmen bilden Eigenschaften die nur lesbar sind, wie IDs oder Anzahl der Polygone eines. Der Grund für das Platzieren des „Set one data“ Nodes in einem Compound ist, dass mehrere Parameter zusammengefasst werden können und über eine Property Page manipuliert werden können. [Aut08]

5.3.2.2 Add Points

Der „Add Point“ Node fügt einer bestehenden Point Cloud Punkte hinzu. Über die eingehenden Ports „PointPosition“ und „OnCreation“ werden die Punkte mit den benötigten Parameter erstellt. Die „PointPosition“ setzt die Punkte anhand von 3D Vektoren oder Objektmittelpunkten. Über den „OnCreation“ Port werden den Partikeln Eigenschaften angehängt, die je nach Art des Einsatzgebietes definiert sind. Für Simulationen werden in der Regel mindestens die Position, Geschwindigkeit und Richtung benötigt, wobei für das Modeling Position, Form und Skalierung genügen. [XSI08]

Der „Add Points“ Node ist der zentrale Node des Emitter Compounds, der, im Zusammenhang mit dem „Simulate Particles“- oder „Simulate Rigid Bodies“ Node (Siehe 5.3.2.3), Partikel von angegebenen Quellen emittiert.

5.3.2.3 Simulation

ICE unterscheidet zwischen Rigid Body Simulation und Particle Simulation. Die Particle Simulation errechnet die Position der Partikel zum gegeben Frame anhand der Position, Geschwindigkeit, Masse und Richtung der Partikel im vorhergegangenen Frame. Der Rigid Body Simulations-Node geht auf die gleiche Weise, wie der Particle Simulations-Node vor, jedoch mit dem Unterschied, dass auf Kollisionen mit Geometrien und die damit einhergehenden Parameter wie Reibung und Elastizität geprüft wird und die Position der Partikel anhand dessen berechnet wird. Die Rigid Body Simulationen in ICE erlauben nur Kollisionsberechnungen der Bounding-Geometrie. So kann zum Beispiel eine einfache Schale die aus einem Würfel oder ähnlichem modelliert wurde, nicht einfach mit Rigid Body simulierten Partikeln gefüllt werden, da die Bounding Box die Öffnung der Schale abschließt und dort eine Kollisionsberechnung erfolgt. [XSI08]

5.3.2.4 Mathematische Operatoren und Funktionen

Die mathematischen Nodes bieten eine Vielzahl an mathematische Operationen und Funktionen. Unterschieden werden dabei:

- Grundlegende Operationen wie Addition, Multiplikation oder Logarithmen
- Vergleichende Operationen wie Minima, Größer- und Kleiner Gleich
- Logische Operationen wie And, Or und Not
- Matrizen Rechnung wie Multiplikation und Transponieren
- Statistik Funktionen wie Durchschnitt und Summen
- Trigonometrie Funktionen wie Sinus, Cosinus und Tangens
- Vektor Rechnung wie Kreuzprodukte, Normalisieren und Skalare Produkte [XSI08]

5.3.2.5 Kontrollfluss Nodes

Die „Execution" Nodes werden verwendet, um bedingte Anweisungen und Verzweigungen in einem ICE Script zu realisieren. Dazu gehört die „If" und „Case" Abfrage. Des Weiteren werden Nodes zum Erstellen von Schleifen bereit gestellt. Dabei wird werden zwischen „While" Schleifen und „Repeat" Schleifen unterschieden. Erstere führt eine Wiederholung nur so lange aus, bis ein bestimmtes Ereignis eintrifft. Die „Repeat" Schleife führt eine Iteration zu über einer angegebene Anzahl durch.

Bei bedingten Anweisungen, also einer „If" oder „Case" Abfrage, ist ein Port immer für eine „true" oder „false" Abfrage verantwortlich. Diese boole'schen Werte können entweder durch andere Nodes übergeben werden oder über eine Property Page gesetzt werden. Die auszuführenden Anweisungen beim Eintreffen eines Falles sind dabei typenneutral und ermöglichen das Ausführen jeglicher Nodes. [XSI08]

5.4 Rendering und Shader

In Softimage|XSI werden ICE Partikel als Geometrie behandelt. Das heißt, es können unterschiedliche Arten von Shadern auf Partikel angewendet werden. Oft werden Partikel jedoch dazu verwendet, um volumetrische Effekte wie Rauch, Feuer oder Wolken zu simulieren. Die dazu verwendeten Shader nennt man Volume Shader. Bei Volume Shader wird simuliert, wie das Licht auf kleinste Partikel, wie sie in Wolken oder Rauch vorkommen, reagiert und interagiert. Die Volume Shader können nun entweder auf eine Geometrie angewendet werden, wobei die Flächen des Objektes gleichzeitig die Begrenzung der Ausdehnung des volumetrischen Effekts darstellen, oder aber direkt als

Environment Shader auf die Szene, was dazu führt, dass sich der volumetrische Effekt durch die ganze Szene zieht. [Raa05] und [XSI08]

Mit dem ICE System werden eigene Volume Shader für die ICE Partikel mitgeliefert. Der Unterschied ist, dass die ICE Shader wie normale Scene Materials verwendet werden können, wozu im Gegensatz dazu herkömmliche Partikel einen eigenen Particle Renderer verwenden. [XSI08]

Der zentrale Node im Render Tree bei Volume Shadern ist die so genannte Particle Volume Cloud. Dieser verwendet die Geometrie der „Point Cloud“ und rendert diese als Volumen. [XSI08]

5.4.1 Attribute Shader

Neben herkömmlichen Shadern, wie Volume oder Gradient Shader, gibt es Attribute Shader. Attribute Shader greifen auf Partikeleigenschaften zu, die von einem ICE Script gesteuert werden und erzeugen dadurch eine Verbindung zwischen dem Render Tree und dem ICE Tree. Diese ermöglichen das Steuern von Shadern über Werte aus den ICE Tree Berechnungen. Um Shader über Werte aus dem ICE Tree steuern zu können, müssen die entsprechenden Nodes im ICE Tree vorhanden sein. So kann zum Beispiel die Distanz von Partikeln zu einem Objekt nicht den Shader steuern, wenn im ICE Tree kein Node verwendet wird, der die Distanz zum Objekt überprüft. [XSI08]

6 Praktisches Beispiel

6.1 Ablauf

Um die Unterschiede zwischen dem ICE System und der herkömmlichen Methode 3D Effekte zu erstellen, klar und deutlich zeigen zu können, werden die ausgewählten Beispiele jeweils auf diese zwei Arten realisiert. Dabei soll versucht werden, beim Erstellen einer Szene möglichst keine Rücksicht auf die Machbarkeit in ICE zu nehmen. Dies soll erst mit der Umsetzung in ICE bestätigt oder wiederlegt werden.

Die Vorgehensweise bei beiden Methoden basiert teilweise auf „Trial and Error“, da gerade bei ICE Scripten ein häufiges Testen der Simulation notwendig ist und die ICE Scripten nach und nach an Funktionalität und Komplexität gewinnen. Zum Zweck der einfacheren Nachvollziehbarkeit wird die Vorgehensweise linear beschrieben.

6.1.1 Einsatz von ICE

ICE soll in dem Umfang eingesetzt werden, um die in Kapitel 1 aufgestellten Hypothesen und Forschungsfragen validieren beziehungsweise beantworten zu können. Folgende Anwendungsbereiche sollen dabei bearbeitet werden:

- Rigid und Soft Body Simulationen
- Particle Simulationen mit Volumen Effekten
- Modeling, inklusive Scripting

6.2 Produktionsdokumentation

6.2.1 Rigid Bodies, Soft Bodies

In dieser Beispielszene sind 16 Quaderförmige Objekte in Diamantform in einer Schale aufgestellt. Diese sollen durch einen weichen, deformierbaren Ball umgestoßen werden. Die Animation des Balles und der umfallenden Steine soll dabei Simuliert werden. Die Animation beziehungsweise die Simulation soll über 200 Frames gehen. Abbildung 22 zeigt ein gerendertes Standbild der Szene.

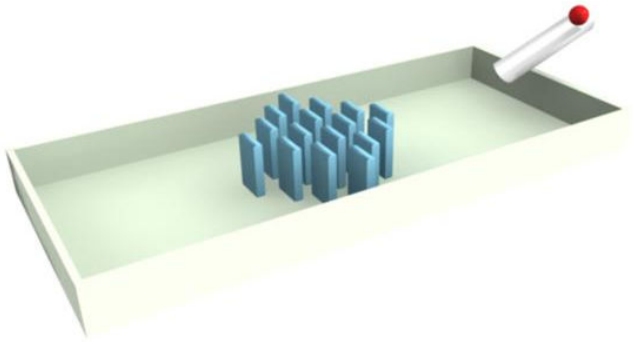

Abbildung 21 Szenenaufbau

6.2.1.1 Herkömmliche Methode

Um dem Ball ein weiches, gummiartiges Verhalten zu geben, benötigt dieser eine Soft Body Simulation. Eine Soft Body Simulation ist ein Verformungsoperator im Operator Stack des Objekts, der auf die Punkte der Geometrie wirkt. Die Simulation basiert auf einem Netz von Federn welches um das Objekt aufgespannt ist. Je dichter das Netz ist, desto genauer wird die Berechnung. Die Federn bestimmen die Elastizität des Objekts.

Eine Interaktion des Soft Body Objekts mit den umgebenden Geometrien zu ermöglichen, müssen diese dem Soft Body Objekt als Hindernisse definiert werden. Eine Interkation kann dabei in unterschiedlichen Detailstufen stattfinden. Über eine Bounding-Geometrie, welche als Quader, Kugel oder elastisches Tuch das Objekt umschließt und über die eigentliche Form des Objekts. Die Bounding-Geometrie ist eine stark vereinfachte Form des eigentlichen Objektes, das an den äußersten Punkten des Objektes anliegt. Je nach Form der Geometrie kann entweder eine Kugel oder ein Quader die Form des Objekts am besten repräsentieren. Die umschließende und anliegende Hülle beachtet dabei das Vorhandensein von Kanten und Rundungen, jedoch nicht Löcher und Einbuchtungen. Die genaueste Berechnung findet über die eigentliche Form des Objekts statt, welche auch die langsamste ist. Für dieses Beispiel ist die eigentliche Form des Objekts die Methode, die zu einem realistischen Verhalten führt. [XSI08]

Die Dominosteine sind als Rigid Body Objekte definiert. Rigid Body Simulationen berechnen die Kollisionen von steifen Objekten anhand deren Masse, Schwerpunkt, Reibung, etc. Wie bei Soft Body Simulationen, werden hier unterschiedliche Berechnungsgenauigkeiten ermöglicht. Für dieses Beispiel genügt eine Bounding Box für die Dominosteine.

Bei Rigid Bodies wird zwischen „Active“ und „Passive“ Rigid Bodies unterschieden. Der Unterschied zwischen „Active“ und „Passive Rigid Bodies“ ist, dass passive Objekte Kollisionen erzeugen können, aber nicht durch die kollidierenden Objekte oder andere Kräfte, wie Gravitation, beeinflusst werden. Da eine Interaktion mit dem Ball stattfinden soll, werden die Dominosteine als Active Rigid Bodies definiert. [Exp06]

In Softimage|XSI können Rigid und Soft Bodies nicht miteinander interagieren. Um die Steine durch den Ball umfallen lassen zu können, wird die Animation, welche durch die Simulation des Soft Bodies erzeugt wurde, verwendet, um damit die Position eines passiven Rigid Bodies, welcher als Bounding Geometrie um den Ball gelegt wurde, zu animieren und somit die Steine umzuwerfen.

Die Simulation kann nur erfolgen, wenn an die Objekte Kräfte angelegt werden. In diesem Fall eine Gravitationskraft, die auf den Ball und die Steine einwirken. Da die Rigid- und Soft Body Simulationen getrennt berechnet werden, benötigen diese jeweils eine eigene Gravitationskraft.

Das Ergebnis der Simulation findet sich auf der beiliegenden CD-ROM unter Beispiele/Rigid_legacy.m2v

6.2.1.2 ICE

In ICE können nur Partikel aktive „Rigid Bodies“ sein. Dies erfordert einen völlig anderen Ansatz in der Umsetzung der Simulation, da sowohl die Steine als auch der Ball als Partikel simuliert werden müssen. Zusätzlich können nur Partikel ein und derselben „Point Cloud“ aktiv mit sich selber kollidieren. Weitere Einschränkungen, wie das Fehlen einer formgenauen Berechnung der Kollision, machen einen Workaround bei der Kollisionsabfrage mit der Schale notwendig. Eine Implementierung eines Softbodysystems fehlt in ICE gänzlich und das geforderte Verhalten des Balles kann nicht realisiert werden.

Mit der Einführung von ICE wurde gleichzeitig eine neue Art von Grundobjekten eingeführt. „Point Clouds“ bestehen lediglich aus den Vertices, also den Eckpunkten, eines Grundobjekts, die bearbeitet werden können.

An einer leeren Point Cloud wird der ICE Tree Operator ausgeführt. In diesem Beispiel werden zwei ICE Trees benötigt: einmal für das Emittieren von Partikeln und Simulieren von Rigid Bodies und einmal für das Setzen der Steine in der Schale. Der emittierende ICE Tree befindet sich aufgrund der Rigid Body Simulation in der Simulations Region des

Operator Stack, was automatisch geschieht, wenn ein simulierender ICE Tree erstellt wird. Dabei gibt es unterschiedliche Emitter Typen: Emittieren von Null Objekten, Positionen im Raum, Oberflächen, Kurven, Schnittpunkte von Objekten, Volumina und das Emittieren einer bestimmten Anzahl an Partikeln auf einmal. In diesem Beispiel wird der „Emit from Position“ Typ verwendet, der die Partikel von einem angegebenen Punkt im Raum emittiert. Für dieses Beispiel wird nur ein Partikel in Form eines Balles emittiert.

Das Rigid Body Verhalten der Partikel in der „Point Cloud“ wird erreicht durch den Rigid Body Node, über dessen Ports die Hindernisse, also passive Rigid Bodies, angegeben werden und über dessen Property Page die Parameter wie Reibung und Elastizität eingestellt werden können.

Abschließend muss dem Simulierenden ICE Tree noch eine Kraft hinzugefügt werden, die auf die Partikel einwirkt, in diesem Fall eine Gravitationskraft. Dies geschieht durch den „Add Forces“ Node und den „Gravity“ Node. Der „Add Forces“ Node ist eigentlich ein Compound, in dessen inneren die „Force“ Eigenschaften der Partikel ausgelesen werden, ein fester Wert oder ein von einem anderen Node kommender Wert addiert wird und anschließend der neue „Force“ Wert in die Partikeleigenschaften geschrieben werden. Wird nur ein „Add Force“ Node an den ICE Tree angehängt, so bewegen sich die Partikel in die Richtung, die in der Property Page des Nodes angegeben werden, jedoch ohne Berücksichtigung auf die Masse der Partikel. Masse kommt durch den „Gravity Force“ Node dazu, der ebenfalls ein Compound ist, bestehend aus einem „Get Data“ Node, der die Masse der Partikel ausliest, einem 3D Vektor der die Richtung und Stärke der Gravitationskraft angibt sowie einem „Multiply by Scalar“ Node der als Wert den 3D Vektor aufnimmt und als Faktor die Masse der Partikel. Das Ergebnis wird an den „Add Forces“ Compound weiter gegeben.

Abbildung 23 zeigt den vollständigen ICE Tree in der Simulationsregion der „Point Cloud“.

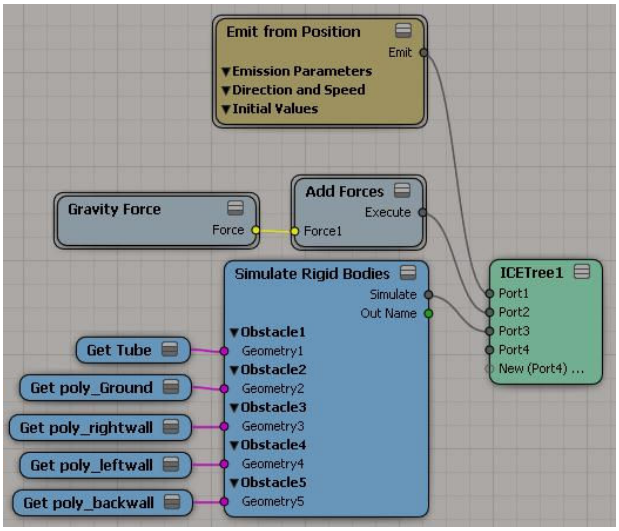

Abbildung 22 ICE Tree - Simulate Rigid Bodies

Der ICE Tree für die Steine befindet sich in der Modeling Region, da dieser ICE Tree lediglich für das Platzieren der Steine zuständig ist. Die Idee ist, ein Gitter aus Punkten als Referenz für die Positionen der Dominosteine zu nehmen.

Um beim Einrichten des Referenz Gitters die globale Position der Punkte richtig im ICE Script einlesen zu können, muss die Rotationstransformation gefreezed werden, was automatisch einen „Center“ Operator *(Die Transformation des Drehpunktes des Objekts)* im Operator Stack des Referenz Gitters erstellt. Dieser übernimmt die „Global Transform“ Werte auf die Werte des „Center“ Operator welche vom ICE Script ausgelesen werden.

Der ICE Tree besteht aus zwei Bereichen: Ein Ast des Trees ist für das Setzen der Partikel und deren Eigenschaften zuständig und ein Ast für die Instanziierung der Dominosteine.

Die eingelesenen Werte werden der Point Cloud entsprechend der 3D Vektordaten hinzugefügt und deren Eigenschaften gesetzt. Um mit den Dominosteinen eine Simulation ausführen zu können, müssen diesen Eigenschaften eingehängt werden, die für die Berechnung notwendig sind, wie zum Beispiel Masse und Kraft.

Das Ergebnis der Simulation findet sich auf der beiliegenden CD-ROM unter Beispiele/Rigid_ice.m2v

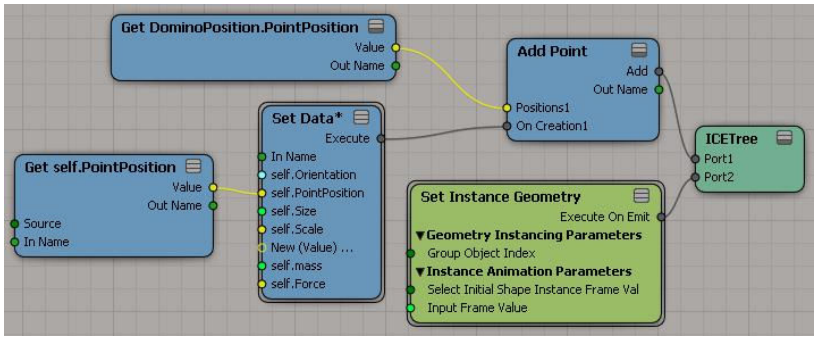

Abbildung 23 ICE Tree - Steine platzieren

6.2.2 Particle und Particle Shader

In dieser Szene ist eine brennende Zigarette in einem Aschenbecher unter einer kleinen Stehlampe positioniert. Der Fluss des aufsteigenden Rauches von der Zigarette soll dabei durch die Lampe gestört werden. Dabei geht es darum, ein möglichst realistisches Verhalten des Rauches zu erreichen. Abbildung 25 zeigt ein gerendertes Standbild der Szene ohne simulierten Rauch.

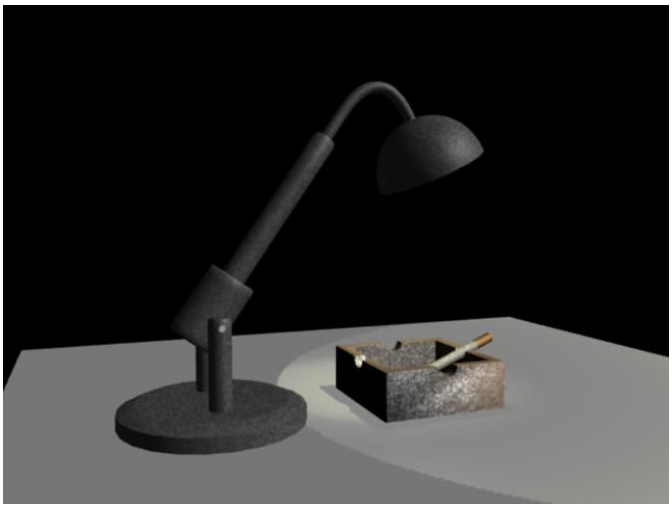

Abbildung 24 Szenenaufbau

6.2.2.1 Herkömmliche Methode

Es gibt zwei unterschiedliche Ansätze für das Simulieren von Rauch mit der herkömmlichen Methode. Dier erste Methode ist das Verwenden eines Volume Shader der die Simulation des Rauches übernimmt.

Der Volume Shader simuliert Rauchpartikel die sich innerhalb einer Geometrie befinden und sich vom kleinsten, lokalen Y-Wert bis zum größten Y-Wert ausdehnen. Die Turbulenzen werden dabei durch Fraktale realisiert. Über die Shader Parameter lassen sich

Dichte, Farbe, Turbulenzen und Berechnungsgenauigkeit des Rauches einstellen. Da es sich dabei um kein Partikel System im Sinne von über XSI bearbeit- und beeinflussbaren Partikeln handelt, sind auch keine Interaktionen mit Geometrien oder Kräften wie Gravitation oder Wind möglich. Dadurch ergibt sich eine niedrige Renderzeit auch bei komplex wirkenden Rauchsimulationen. Für dieses Beispiel ist diese Methode aufgrund fehlender Interaktionsmöglichkeiten nicht brauchbar.

Abbildung 25 Smoke Shader

Die zweite Methode verwendet das Partikelsystem von XSI. Das Partikelsystem besteht aus mehreren Komponenten:

Particle Simulator: ist ein Operator der für das Verhalten der Partikel verantwortlich ist. Der Operator überprüft eventuell einwirkenden Kräfte oder Kollisionen mit einer Geometrie und lässt diese in die Berechnung der Partikelbewegung einfließen. Pro Partikelsimulation wird ein Operator benötigt.

Particle Cloud: repräsentiert den Partikelsimulator in einer für den Benutzer verwertbaren Form.

Emitter: gibt an, wo und wie die Partikel erzeugt werden. Per Default werden die Normalen der Polygone verwendet. In diesem Beispiel werden die am Ende des Zylinders nach oben zeigenden Polygone ausgelöst und als Emitter gesetzt. Da es sich um ein rundes Objekt handelt, werden die Partikel entsprechend der Polygonnormalen aufgefächert in den Raum emittiert. Um ein gerichtetes Emittieren zu erzeugen, muss der Emitter auf eine globale Richtung eingestellt werden.

Partikel Typ: definiert die Eigenschaften der Partikel wie Masse, Geschwindigkeit, Lebensdauer und Anzahl der Partikel. Da Rauch sich nach dem entstehen mehr und mehr diffundiert, wird ein maximales Alter der Partikel gesetzt bei dem sie gelöscht werden.

Die hier angewendete Methode entspricht nicht einer echten Simulation von Rauch, die auf der Simulation von Flüssigkeiten basiert, sondern versucht lediglich ein realistisches Aussehen von Rauch zu erzeugen.

Die emittierten Partikel bewegen sich per Default geradlinig von der Emitter-Fläche weg. Um die durch aufsteigende Warmluft erzeugten Turbulenzen simulieren zu können, werden verschiedene Kräfte auf die Partikel angewendet. Eine „Turbulence“ Kraft sorgt für die notwendigen unregelmäßigen Bewegungen der Partikel. Die Bewegung wird erzeugt durch eine Rauschfunktion, die in allen Achsen der Partikelpositionen angewendet wird.

Um eine Interaktion der Partikel mit der Lampe zu bekommen, werden die entsprechenden 3D Objekte als „Obstacle“ definiert. Wie bei Rigid Bodies, ist auch hier eine Angabe der benötigten Berechnungsgenauigkeit erforderlich. Da die Rauchpartikel nicht mit der Oberfläche reagieren sollen, außer nicht durch sie hindurch zu treten, werden sämtliche Parameter wie Reibung und Elastizität auf null gesetzt.

Da auf die Partikel keine weitere Kraft außer der Turbulenz einwirkt, fließen die Partikel an der Lampe entlang und halten die Richtung in der sie durch die Kollision gebracht wurden. Damit diese nach der Kollision weiter aufsteigen, benötigt es eine Gravitationskraft welche die Partikel nach oben zieht.

Eine realistische Rauchsimulation besteht nicht nur aus einer komplexen Simulation der Partikelbewegungen, sondern auch aus einem realistischen Shader für die Partikel.

Per Default ist ein Billboard Shader auf den Partikeln. Ein Billboard Shader zeigt mit der Normalen immer zur Kamera beziehungsweise zu einem angegeben Punkt und erscheint so zu jedem Zeitpunkt gleich, unabhängig der Kameraposition. Die Partikel sind dabei mit einem konstanten Material belegt die zum Rand hin durchsichtig werden. Für dieses Beispiel werden die Default Shader auf den Partikeln behalten.

Das Ergebnis der Simulation findet sich auf der beiliegenden CD-ROM unter Beispiele/particle_legacy.m2v

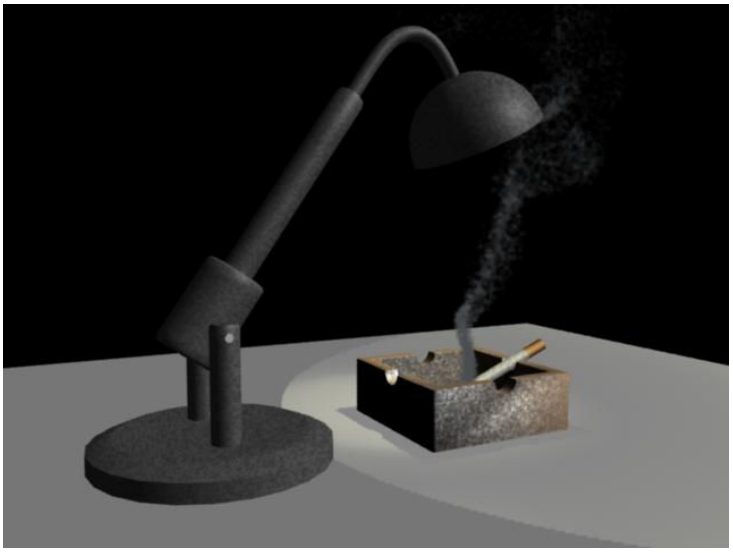

Abbildung 26 Particle Simulation mit herkömmlicher Methode

6.2.2.2 ICE

Im Gegensatz zur herkömmlichen Methode, wird in ICE das Verhalten der Partikel zum Zeitpunkt der Erzeugung vom Benutzer festgelegt.

Der ICE Tree lässt sich grob in vier Bereiche unterteilen:

1. Emitter: Der Emitter erzeugt die Partikel, wie bei der herkömmlichen Methode, von den ausgelösten Polygonen am Ende der Zigarette. Es muss ebenfalls eine Richtung angegeben werden, da die Partikel ansonsten entlang der Polygonnormalen emittiert werden. Dem Emitter-Node wird die Eigenschaft „Particle Age Limit“ übergeben, welche im ICE Tree von anderen Nodes als Ausgangswert weiterverwendet werden kann. In diesem Beispiel werden die Partikel beim Erreichen dieses Wertes über den Node „Delete Particle at Age Limit“ gelöscht.
2. Turbulenz: Die Turbulenz wird bei diesem Beispiel nicht durch Anlegen einer Kraft, sondern durch Setzen der Punkte nach vorgegebenen Regeln erreicht. Dazu werden die aktuellen Punktpositionen eingelesen, welche als Ausgangspunkt für die Turbulenzen dienen. Die Turbulenz bewegt dabei den Punkt um den angegebenen Grundwert entsprechend der eingestellten Stärke und Art der Turbulenz. Ein weiterer Parameter ist das Zentrum der Turbulenzen, die in globalen Koordinaten angegeben werden. Um die aufsteigende Rauchsäule lebendiger zu gestalten, wird das Zentrum der Turbulenz per Zufall rundum einen Wert verschoben, dessen Varianz durch das Alter der Partikel vorgegeben wird.

3. Kollision: Die Kollision wird durch einen „Bounce off Surface“ Compound realisiert. Dieser Compound verwendet keine Rigid Body Simulationen, sondern überprüft die Position der Partikel zum angegebenen Objekt und errechnet im Falle einer stattgefundenen Kollision das Verhalten des Partikels in den nachfolgenden Frames aufgrund der eingestellten Parameter, wie zum Beispiel Anzahl der Aufpralle oder Winkel des Zurückprallens. Um eine fließende Bewegung zu bekommen, werden diese Parameter auf null gestellt, was dazu führt, dass die Partikel an der Oberfläche entlang gleiten.
4. Post Kollision: Da die Partikel beim Verlassen der Oberfläche die Richtung beibehalten, die sie durch die Kollision eingenommen haben, wird eine Gravitationskraft in die entgegengesetzte Richtung angelegt. Dies geschieht zu einem definierten Zeitpunkt.

Neben diesen Bereichen gibt es noch einen weiteren Node der für die Farbe der Partikel entsprechend dem Alter der Partikel zuständig ist. Der Node sorgt für einen Farbverlauf bei den emittierten Partikeln, der dazu dient, um die höhere Dichte der Partikel kurz nach dem Austreten aus der Zigarette realistisch darstellen zu können. Ein weiterer Faktor für den Realismus ist die Anzahl der Partikel: je höher die Anzahl und je kleiner die Partikel, umso detailierter und realistischer wirkt die Simulation der Partikel und umso aufwändiger wird die Berechnung. Abbildung 27 zeigt den vollständigen ICE Tree der Rauchsimulation.

Als Shader für die Partikel, die in Kugelform emittiert werden, dient der ICE Particle Volume Shader (Siehe Kapitel 5). Der Shader übernimmt die im ICE Tree eingestellten Werte der Partikel auf die Farbwerte des Volumenmaterials.

Das Ergebnis der Simulation findet sich auf der beiliegenden CD-ROM unter Beispiele/particle_ice.m2v

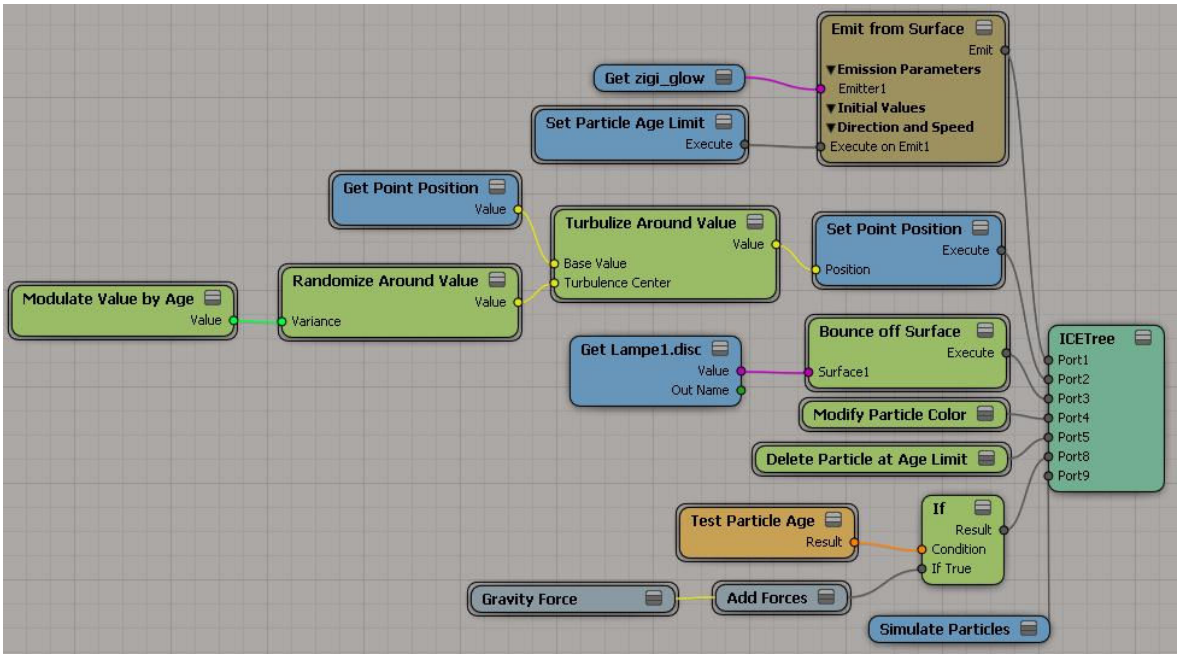

Abbildung 27 ICE Tree – Rauch

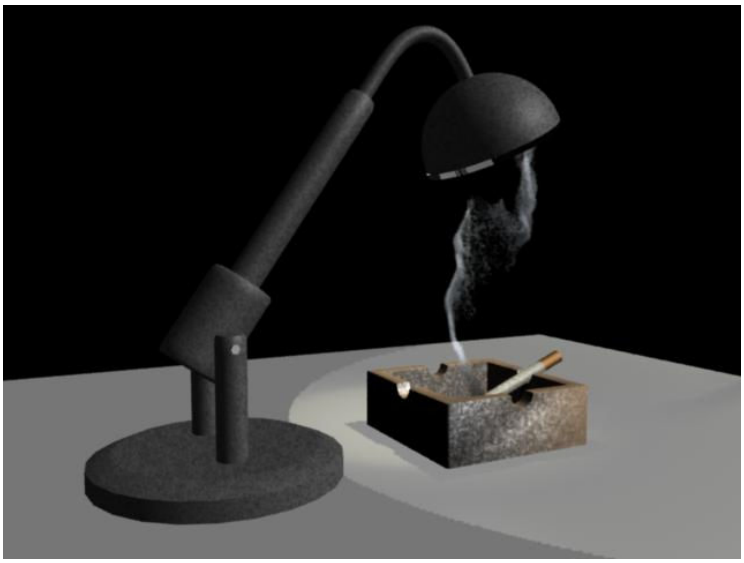

Abbildung 28 Particle Simulation mit ICE

6.2.3 Modeling

In diesem Beispiel soll die Oberfläche eines Geometrieobjekts, ein Würfel, mit Details versehen werden, welche prozedural erstellt werden. Diese Art der Herausarbeitung von Details ohne speziellen Zweck oder Bedeutung wird Greeble genannt. [Wik09]

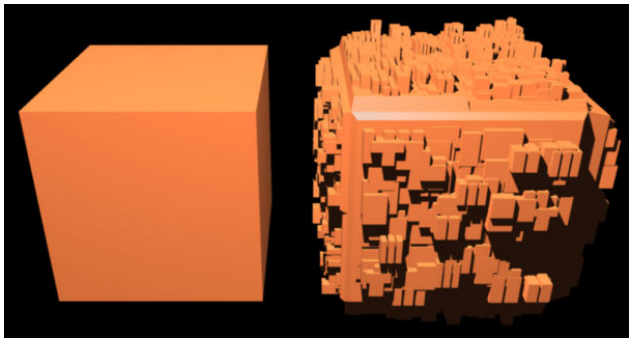

Abbildung 29 Würfel ohne und mit Greeble [Wik09]

6.2.3.1 Herkömmliche Methode

Um eine Oberfläche mit Details zu versehen gibt es zwei unterschiedliche Methoden. Zum einen kann die Oberfläche mit den herkömmlichen Tools in kleinere Polygonabschnitte unterteilt und dann extrudiert werden. Dies kann, je nach gewünschtem Detailreichtum, sehr arbeitsintensiv sein.

Zum anderen kann die Oberfläche mit Hilfe eines Scripts gestaltet werden. Der zugrunde liegende Algorithmus ist dabei relativ einfach und ist ähnlich der manuellen Methode:

Ein gewähltes Polygon wird unterteilt. Per Zufall wird eine Anzahl der neuerstellten Polygone ausgewählt und diese wiederum um Zufallswerte extrudiert. Anschließend werden wieder Polygone per Zufall ausgewählt und der Vorgang des Unterteilens und Extrudierens wird wiederholt. Entsprechend des gewünschten Detailreichtums wird diese Schleife beliebig oft wiederholt.

Das hier präsentierte Beispiel basiert auf dem Greeble Plug-In für Softimage|XSI von Michael Gangolf. Für dieses Beispiel wurde der für die Ausführung essentielle Code aus der Plug-In Umgebung herausgelöst und stark vereinfacht und angepasst, um das dahinterliegende Prinzip nachvollziehbar erläutern zu können.

Das Script wird nach der Selektion eines Polygons im Scripteditor ausgeführt. Für den interessierten Leser findet sich eine detailierte Erläuterung zum Programmcode als Kommentar (grün) im Script.

```
Subdivision=2; //Iterationen bei der Polygonunterteilung
Threshold=9; //Wie viele Polygone Extrudiert werden sollen
height_min=0.01;//Minimaler Extrudier wert
height_max=0.9;//Maximaler Extrudier wert
loop=3; //wie oft die Operation ausgeführt werden soll - bestimmt den
Detailreichtum

if (selection(0).type != "polySubComponent") {
        logmessage("Only polygons!",siError);
}
//Subidivide Polygons Operator mit definierter Anzahl an Iterationen
subd=ApplyTopoOp("SubdividePolygon", selection(0));
SetValue(subd+".subdivisions", Subdivision, null);

//übergib der Variable p_all eine Collection der neuen Polygone
p_all = selection(0).subcomponent.componentcollection;

//Führt für jede Stufe einmal folgenden Code aus
for (j=0;j<loop;j++){

    deselectall();

    //Nimmt alle Polygone der Collection
    //und selektiere die Polygone, bei
    //dem der Zufallszahlgenerator einen Wert
    //über dem definierten Schwellwert generiert.
    //Je niedriger der Schwellwert, desto mehr
    //Polygone werden selektiert.
    for (i=0;i<p_all.count;i++){

        //random() generiert Zufallszahl zwischen 0 und 1
        //daher darf der Schwellwert 9 nicht übersteigen
        if (Math.random()*10>Threshold) {
                addtoselection(p_all.item(i));
        }
    }

    //Legt die neu selektierten Polygone in der Variable p_sub ab.
    p_sub=selection(0).subcomponent.componentcollection;

    //Wenn Polygone selektiert wurden, dupliziere diese
    //und platziere sie entsprechend der zufallsgenerierten
    //Polygonlokalen y-Position (entspricht Extrude Op)
    if (p_sub.count>0) {

        var  rand = Math.random()*height_max+height_min;
        dubl=DuplicateMeshComponent(selection(0),0,rand);
    }

    //Unterteile die extrudierten Polygone und übergib
    //die Collection an Variable p_all,
    //solange die Anzahl der gewünschten Durchläufe
    //noch nicht erreicht wurde.
    if (j<loop-1){

        subd=ApplyTopoOp("SubdividePolygon", selection(0));
        SetValue(subd+".subdivisions", Subdivision, null);
        AddToSelection(p_all, null, true);

        p_all=selection(0).subcomponent.componentcollection;
    }
}
```

Abbildung 30 Greeble-Script auf Würfel angewendet

6.2.3.2 ICE

Da ICE das Ausführen von externen, also nicht als ICE Node verfügbaren Operatoren unmöglich macht, muss ein anderer Ansatz gewählt werden, der keinen Extrude Operator verwendet.

Die Idee ist, die Oberfläche einer Geometrie als Emitter von Partikeln zu setzen und den Partikeln Geometrieinstanzen zu übergeben, welche über einen Zufallswert skaliert werden. Auf diese Art und Weise sind sehr schnell brauchbare Ergebnisse zu erzielen.

Dabei werden keine Partikelsimulationen erzeugt, sondern lediglich der Emitter Compound verwendet um ein kompaktes und mächtiges Werkzeug für die Partikelsteuerung zu bekommen. Durch das Fehlen einer Partikelsimulation bleiben die emittierten Partikel am Emitter-Objekt haften. Beim Emittieren der Partikel muss darauf geachtet werden, dass die Partikel alle zu einem Zeitpunkt emittiert werden. Um Partikel verwenden zu können, muss in der Szene eine leere „Particle Cloud“ erstellt werden, auf der der ICE Tree angewendet werden kann. Der Würfel dient dabei nur als Emitter Objekt. Zusätzlich benötigt die Szene eine oder mehrere Geometrien, die auf die Partikel instanziiert werden können.

Die Nodes welche das Aussehen des Greebels beeinflussen, führen alle in den Emitter Compound, da die Werte zum Zeitpunkt der Erzeugung gesetzt werden müssen.

Um die Größe der Partikel unterschiedlich zu gestallten, wird der Wert über einen Zufallszahlengenerator gesteuert. Gleichzeitig wird auch die Skalierung der Partikel in alle drei Achsen innerhalb eines Wertebereiches generiert.

Dem Emitter wird eine Geometrie zur Instanziierung übergeben, die beliebig komplex ausfallen kann. Des Weiteren ist es hier möglich, eine Gruppe unterschiedlicher Objekte als Instanzen zu verwenden, welche wiederum über einen Zufallszahlengenerator den Index steuern, der für die Auswahl der Geometrie in der Gruppe zuständig ist.

Zuletzt wird die zufällige Rotation der Partikel gesetzt. Um die Rotation in diesem Beispiel immer in 90 Grad Schritten durchzuführen, wird der ganzzahlige Wert des Zufallszahlengenerators mit 90 multipliziert, das Ergebnis in einen skalaren Wert konvertiert. Dieser Wert kann nun in einen Rotationswert konvertiert werden, der dem Partikel übergeben werden kann. Abbildung 30 zeigt den gesamten ICE Tree des Beispiels.

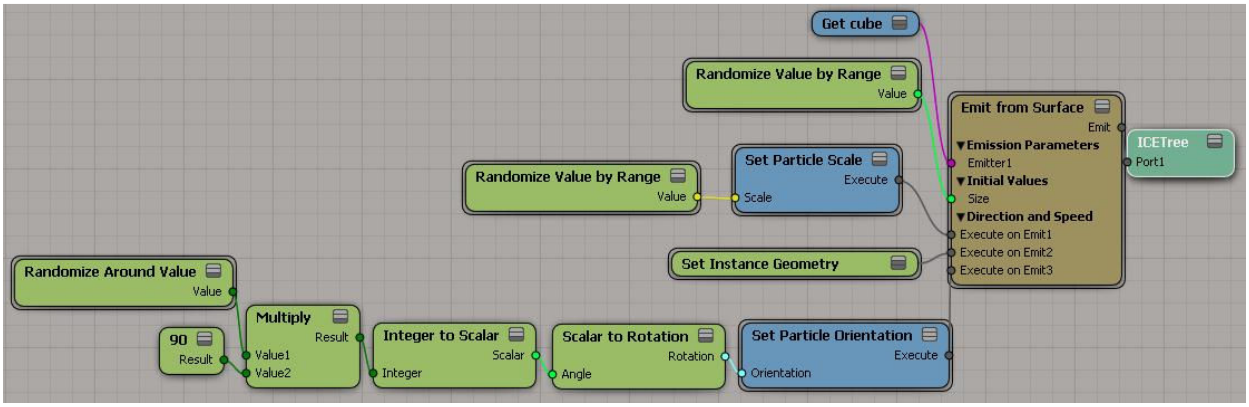

Abbildung 31 ICE Tree - Greeble

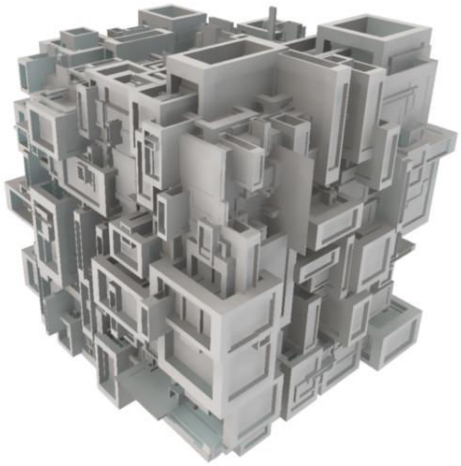

Abbildung 32 ICE Tree auf Würfel angewendet. Instanzgeometrie nicht sichtbar.

7 Analyse und Fazit

7.1 Analyse und Beantwortung der Forschungsfragen

Bereits während der theoretischen Bearbeitung dieses Themas stellte sich heraus, dass ICE nicht für alle Bereiche, für die bisher Scripting eingesetzt wurde, als Ersatz dienen kann. Für die in Kapitel 4 beschriebenen Einsatzgebiete von Scripting in XSI ergeben sich folgende Erkenntnisse:

Pipeline Scripting: ICE kann zwar sämtliche Eigenschaften des XSI Objekt Modells einlesen, jedoch fehlen zur weiteren Verarbeitung die entsprechenden Nodes, die eine Formatumwandlung oder ein Exportieren der Daten in eine Datei ermöglichen. Einzige Möglichkeit Daten aus ICE in eine Datei zu schreiben ist das Cachen von Simulationsdaten.

Batch Scripting: da ICE nur in der grafischen Benutzeroberfläche von ICE ausgeführt werden kann, ist der Einsatz für Batch Scripting nicht möglich, wobei ICE jedoch Potential für Stapelverarbeitung hätte. Zum Beispiel zum Berechnen komplexer Simulationen und Cachen der Simulation für eine größere Zahl an Szenen.

Erweiterungen für Szenenelemente und XSI Features: auch hier ist das Fehlen von entsprechenden Nodes ein Hindernis. Custom Properties, Filter, Commands und Events sind nicht möglich in ICE. Eine Ausnahme bilden die Custom Operator. Da ICE selbst als Operator fungiert, ist jeder ICE Tree quasi ein selbst erstellter Operator. Über Compounds lassen sich diese auch praktisch abspeichern und können dadurch schnell wieder verwendet werden. In Kombination mit einem Custom Command könnte so schnell ein Tool für verschiedenste Bereiche der Animation oder Modellierung erstellt werden.

Bei der Erarbeitung der praktischen Beispiele ergaben sich folgende Erkenntnisse:

Die Simulation von Soft- und Rigid Bodies ist in der herkömmlichen Methode sehr ausgereift und führt auch ohne lange Einarbeitungszeit schnell zu einem guten Ergebnis. In ICE muss zuerst das Prinzip der Funktionsweise von ICE verstanden werden, um erste Ergebnisse sehen zu können. Mit zunehmender Komplexität der Simulationsumgebung, also Anzahl und Form der Hindernisse und Kräfte, wird der Aufwand entsprechend hoch und die Verwendung von ICE verlangsamt den Workflow erheblich. Zudem kann auf eine vorgefertigte Implementierung eines Soft Body Systems nicht zurückgegriffen werden.

Zwar stellte sich im Zuge der Recherchen heraus, dass eine eigene Simulation eines Soft Body Systems möglich ist, sich aber aufgrund des Aufwands für die Implementierung dieses komplexen Algorithmus kaum für den Einsatz von ICE steht.

Im Bereich der Partikel Simulation konnten mit ICE sehr schnell gute Ergebnisse erzielt werden. Die Bedienung ist auf niederem Niveau sehr intuitiv und übersichtlich. Mit steigender Komplexität des Algorithmus verlangt der Umgang mit ICE aber ein grundlegendes Verständnis programmiertechnischer Konzepte. Gegenüber der herkömmlichen Methode bietet ICE eine sehr zentrale Kontrolle der Partikel über den ICE Tree, wo sämtliche Parameter der an der Point Cloud anliegenden Operationen angepasst werden können.

Im Modeling Bereich zeigte sich, dass ICE nur bedingt Operationen zur Oberflächengestaltung bietet. Diese beziehen sich allesamt auf das Verformen der Objekte und ermöglichen kein Extrudieren oder Hinzufügen von Polygonen. Die Umsetzung des Greeble-Scripts in ICE konnte daher nicht nach dem gleichen Prinzip erfolgen, wie dies in JScript der Fall war. Die eingesetzte Methode zeigte jedoch, dass in ICE nicht nur sehr schnell ein komplexer Effekt erstellt werden kann, sondern dieser im Vergleich zur herkömmlichen Methode auch einfacher zu warten, erweiterbar und im Gegensatz zum Script auch animierbar ist. Es zeigte sich im Zuge der Recherchen, dass für spezielle Modelingtools die auf die Bearbeitung von Polygonen ausgelegt sind, nicht auf herkömmliches Scripting verzichtet werden kann.

Im Folgenden findet sich die Beantwortung der Forschungsfragen.

Kann ICE das Scripting ersetzen, und wenn nicht, wo sind die Grenzen von ICE?

ICE kann das Scripting in Softimage|XSI nicht in allen besprochenen Bereichen ersetzen. Grenzen zeigen sich auf beim Verarbeiten von Daten aus dem Object Modell die nicht für den Einsatz in ICE gedacht sind. Des Weiteren stehen keine Export und Importfunktionen in ICE bereit.

Kann ICE die vorhandenen Simulationsmethoden wie z.B. Soft Bodies ablösen?

Das Rigid Body System in ICE ist eine vereinfachte Version des herkömmlichen Systems. Abzüge wurden bei der Berechnungsgenauigkeit bei Kollisionen gemacht und aktive Kollisionen sind nur innerhalb einer Point Cloud möglich.

Eine Implementierung eines Soft Body Systems in ICE wurde nicht realisiert, und die Umsetzung eines Soft Body Algorithmus wäre zu aufwändig, um die Methode in ICE der herkömmlichen vorzuziehen.

Welche Vorteile bietet ICE gegenüber dem herkömmlichen Workflow der Effekterstellung?

Ein großer Vorteil ist die Wiederverwendbarkeit der erstellten Effekte die in anderen Projekten als Ausgangsbasis für Effekte dienen oder gleich zum Einsatz kommen können.

Ein weiterer Vorteil ist die einfache und zentrale Kontrolle der Effekt bestimmenden Parameter über die Nodes im ICE Tree. Parameter können in Echtzeit verändert werden und über Compounds in Property Pages zusammengefasst werden. Eine erneute Ausführung von Scripten oder gar erneutes Kompilieren von Effekten ist nicht notwendig.

Sollten die herkömmlichen Deformer und Effekte nicht die gewünschten Parameter bieten, können diese in ICE nachgebaut und um den benötigten Parameter erweitert werden. Dies erlaubt eine maximale Anpassung an die Bedürfnisse und Anforderungen des Benutzers.

Durch die Unterstützung von Multithreading der Nodes, ist eine höhere Simulationsgeschwindigkeit möglich, welches sich vor allem bei der Simulation von Partikel auswirkt.

7.2 Fazit

Beim Einarbeiten in ICE verläuft die Lernkurve steil, da nur wenige Handgriffe notwendig sind um zu einem ersten Ergebnis zu kommen. Die vorhandene Dokumentation ist umfangreich und ermöglicht auch Digital Artists, die keine Programmierkenntnisse besitzen, einen schnellen Einstieg. Des Weiteren ist eine solide Community um ICE vorhanden, die sich mit der Entwicklung von Compounds und selbst Entwickelten Nodes beschäftigt. Auf weiterführende Literatur konnte zum Zeitpunkt der Erstellung dieser Arbeit nicht zurück gegriffen werden, da das ICE System erst seit ca. 5 Monaten auf dem Markt erhältlich war.

Bei komplexeren Effekten und Simulationen ist ein grundlegendes Verständnis programmatischer Konzepte erforderlich und mathematische Teilgebiete wie Matrizen und Vektorrechnung sollten dem Benutzer nicht fremd sein.

ICE bietet eine gute Ergänzung zum herkömmlichen Scripting und ermöglicht schnelle Ergebnisse ohne aufwändigen Code entwickeln zu müssen. Durch das Fehlen eines Kompilier- oder Interpretiervorgangs des ICE Trees fällt zeitaufwändiges Debugging und Testen weg und der Digital Artist kann sich voll und ganz auf die Logik des zu implementierenden Effekts konzentrieren.

7.3 Mögliche weiterführende Arbeiten

Custom ICE Nodes: Da in Softimage|XSI eigene, kompilierte Nodes verwendet werden können, wäre es interessant zu erfahren, was dabei beachtet werden muss und welche Möglichkeiten dadurch gegeben sind.

Schnittstellen zwischen ICE und Scripting: Ist es möglich, eine Schnittstelle zwischen ICE und Scripting zu erstellen, um aus einem ICE Tree herkömmliche Scripten steuern und Daten austauschen zu können? Gibt es eine Möglichkeit, den Flaschenhals der Ausführungsgeschwindigkeit bei Scripten zu umgehen? Was muss dieser Node für Eigenschaften aufweisen und wie werden diese implementiert?

Implementierung von Flüssigkeitssimulationen: Es gibt bereits Custom ICE Nodes die es erlauben, schnelle und realistische Flüssigkeitssimulationen zu erstellen. Ist es möglich, diese komplexen Algorithmen in ICE zu implementieren?

Erweiterungen zum Prozeduralen Modeling: Ist es möglich, ein Set von Custom ICE Nodes zu erstellen, die es ermöglichen auf die Polygone und deren Operatoren zugreifen zu können? Wie könnte prozedurales Modeling in ICE aussehen?

8 Literaturverzeichnis

[Akt09] Aktive Inhalte: ActiveX. *Bundesamt für Sicherheit in der Informationstechnik.* [Online] Bundesamt für Sicherheit in der Informationstechnik. [Zitat vom: 11. 2 2009.] http://www.bsi.de/fachthem/sinet/gefahr/aktiveinhalte/definitionen/activexcontrols.htm.

[Bun09] Aktive Inhalte: JavaScript/JScript. *Bundesamt für Sicherheit in der Informationstechnik.* [Online] Bundesamt für Sicherheit in der Informationstechnik. [Zitat vom: 11. 2 2009.] http://www.bsi.de/fachthem/sinet/gefahr/aktiveinhalte/definitionen/javascript.htm.

[Akt091] Aktive Inhalte: VB Script. *Bundesamt für Sicherheit in der Informationstechnik.* [Online] Bundesamt für Sicherheit in der Informationstechnik. [Zitat vom: 11. 2 2009.] http://www.bsi.de/fachthem/sinet/gefahr/aktiveinhalte/definitionen/VB Script.htm.

[Aut09] Autodesk. 2009. Autodesk Softimage ICE. *Autodesk Softimage.* [Online] Autodesk, 2009. [Zitat vom: 24. 2 2009.] http://www.softimage.com/products/xsi/ice/.

[Aut08] Autodesk. 2008. ICE Attribute Reference. *XSI Wiki.* [Online] Autodesk, 2008. [Zitat vom: 18. 3 2009.] http://softimage.wiki.softimage.com/index.php/ICE_Attribute_Reference.

[Bli02] Blieberger, Johann, Burgstaller, Bernd und Schildt, Gerhard-Helge. 2002. *Informatik Grundlagen.* Wien : Springer-Verlag, 2002.

[Ver08] Cinema 4D. *Wikipedia.* [Online] 19. 12 2008. [Zitat vom: 14. 1 2009.] http://de.wikipedia.org/wiki/Cinema_4D.

[Cod08] CodeGear. 2008. COM-Schnittstellenzeiger. *Codegear.* [Online] 2008. [Zitat vom: 5. 12 2008.] http://docs.codegear.com/docs/radstudio/radstudio2007/RS2007_helpupdates/HUpdate4/DE/html/devwin32/ooccominterfacepointers_xml.html.

[Dig05] Digital Production pipelines: examining structures and methods in the computer effects industry. 2005 *Texas Digital Library.* [Online] 29. 8 2005. [Zitat vom: 13. 2 2009.] http://repositories.tdl.org/tdl/handle/1969.1/2406.

[Har03] Ernst, Hartmut. 2003. *Grundkurs Informatik.* Braunschweig/Wiesbaden : Friedr. Vieweg & Sohn Verlagsgesellschaft mbH, 2003.

[Exp06] ExpertRating. 2006. Expertrating, MAYA Tutorial, Dynamics and Special Effects. *Expertrating.com.* [Online] ExpertRating, 2006. [Zitat vom: 15. 3 2009.] http://www.expertrating.com/courseware/MayaCourse/MAYA-Dynamics-SpecialEffects-1.asp.

[Fra08] Frank Schullenburg, et al. 2008. Stereoskopie . *Wikipedia.* [Online] 4. 12 2008. [Zitat vom: 6. 12 2008.] http://de.wikipedia.org/wiki/Stereoskopie.

[Gan07] Ganster, Björn und Klein, Reinhard. 2007. Computer Graphik, Universität Bonn. [Online] 4 2007. [Zitat vom: 9. 12 2008.] http://cg.cs.uni-bonn.de/.

[GNU06] GNU Make - Gnu Project - Free Software Foundation (FSF). 2006 *GNU.* [Online] Free Software Foundation, Inc, 2006. [Zitat vom: 3. 2 2009.] http://www.gnu.org/software/make/.

[Tho08] Güttler, Thomas. 2008. Python, Programmieren macht Spaß. [Online] 2008. [Zitat vom: 11. 2 2009.] http://www.thomas-guettler.de/vortraege/python/einfuehrung.html.

[Mic01] Isner, Michael. 2001. Scripting. *Isner.com.* [Online] 10 2001. [Zitat vom: 12. 2 2009.] http://www.isner.com/tutorials/commands_vs_object_model2.htm.

[ITW09] ITWissen. **2009** *COM :: Component Object Model :: Definition :: IT Lexikon.* [Online] Datacom Buchverlag GmbH, 2009. [Zitat vom: 9. 2 2009.] http://www.itwissen.info/definition/lexikon/component-object-model-COM.html.

[Don811] Knuth, Donlad. 1981. *The Art of Computer Programming.* s.l. : Addison-Wesley, 1981.

[Arn03] Königsmarck, Arndt von. 2003. *CInema 4D 8.1.* München : Addison-Wesley Verlag, 2003.

[Lev03] Levi, Paul und Rembold, Ulrich. 2003. *Einführung in die Informatik.* s.l. : Carl Hanser Verlag München Wien, 2003.

[Mar08] Lipp, Markus. 2008. *Interactive Copmuter generated Architecture.* Wien : s.n., 2008.

[Mah04] Mahintorabi, Keywan. 2004. *Maya 6.* Bon : mitp-Verlag, 2004.

[Max08] Maxon. 2008. Maxon Geschichte. *Maxon.* [Online] 2008. [Zitat vom: 13. 1 2009.] http://www.maxon.net/pages/contact/history_d.html.

[New09] NewTek. 2009. NewTek . *NewTek.* [Online] 2009. [Zitat vom: 17. 1 2009.] http://www.newtek.com/newtek/focus.php.

[Ore00] Orenstein, David. 2000. QuickStudy: Application Programming Interface (API). *Computerworld.* [Online] 20. 1 2000. [Zitat vom: 3. 12 2008.] http://www.computerworld.com/action/article.do?command=viewArticleBasic&articleId=43487.

[Per09] Perl (Programmiersprache). *Wikipedia.* [Online] [Zitat vom: 11. 2 2009.] http://de.wikipedia.org/wiki/Perl_(Programmiersprache).

[Plu08] Plug-in (Computing). **2008** *Wikipedia.* [Online] 23. 11 2008. [Zitat vom: 5. 12 2008.] http://en.wikipedia.org/wiki/Plug-in_(computing).

[Pro08] Programmierschnittstelle. 2008. *Wikipedia.* [Online] 29. 10 2008. [Zitat vom: 5. 12 2008.] http://de.wikipedia.org/wiki/Programmierschnittstelle.

[Pyt09] Python (Programmiersprache). *Wikipedia.* [Online] [Zitat vom: 11. 2 2009.] http://de.wikipedia.org/wiki/Python_(Programmiersprache).

[Raa05] Raab, Rüdiger Manfred Karl Anton. 2005. *Shaderwriting für MentalRay und Softimage XSI.* St. Pölten : s.n., 2005.

[Guy01] Rabiller, Guy. 2001. Guy Rabiller - 3D technical Director. *Guy Rabiller - 3D technical Director.* [Online] 2001. [Zitat vom: 3. 2 2009.] http://www.grabiller.com/site/index.php?lang=eng&url=tut_script_commandsvsobjects_eng.html.

[Raf05] Raffaeld. 2005. Image:Expr5.jpg. *XSI-Wiki.* [Online] 26. 9 2005. [Zitat vom: 2. 12 2008.] http://softimage.wiki.softimage.com/index.php/Image:Expr5.jpg.

[SBl06] SBlair. 2006. Batch Scripting. *XSI-Wiki.* [Online] Autodesk, 4. 7 2006. [Zitat vom: 15. 2 2009.] http://softimage.wiki.softimage.com/index.php/Batch_Scripting_(XSISDK).

[SBl081] SBlair. 2008. CPP API Overview. *XSI-Wiki.* [Online] Autodesk, 2008. [Zitat vom: 3. 2 2009.] http://softimage.wiki.softimage.com/index.php/Cpp_API_Overview_(XSISDK).

[SBl082] SBlair. 2008. CPP Compilation. *XSI-Wiki.* [Online] Autodesk, 2008. [Zitat vom: 9. 2 2009.] http://softimage.wiki.softimage.com/index.php/Cpp_Compilation_(XSISDK).

[SBl07] SBlair. 2007. JScript. *XSI-Wiki.* [Online] Autodesk, 22. 8 2007. [Zitat vom: 11. 2 2009.] http://softimage.wiki.softimage.com/index.php/JScript_(XSISDK).

[SBl08] SBlair. 2008. Object Model Diagrams - XSI Wiki. *XSI-Wiki.* [Online] Autodesk, 2008. [Zitat vom: 26. 1 2009.] http://softimage.wiki.softimage.com/index.php/Object_Model_Diagrams.

[SBl083] SBlair. 2008. Python. *XSI-Wiki.* [Online] Autodesk, 14. 11 2008. [Zitat vom: 11. 2 2009.] http://softimage.wiki.softimage.com/index.php/Python_(XSISDK).

[SBl071] SBlair. 2007. VB Script. *XSI-Wiki.* [Online] Autodesk, 30. 8 2007. [Zitat vom: 11. 2 2009.] http://softimage.wiki.softimage.com/index.php/JScript_(XSISDK).

[Max03] Schönherr, Maximilian. 2003. Maya Pit - My First Expression/Meine erste Expression. *Maximilian Schönherr.* [Online] 10. 2 2003. [Zitat vom: 29. 3 2009.] http://www.maxschoenherr.de/animation/Maya/MayaPit/myFirstExpression/pit14myFirstExpression.html.

[Sco08] Scott, Dean A. und Vost, Ben. 2008. Lightwiki. *Lightwave History* . [Online] 2008. [Zitat vom: 17. 1 2009.] http://www.lightwiki.com/LightWave_History.

[Rob99] Sedgewick, Robert. 1999. *Algorithmen in C++.* s.l. : Addison-Wesley, 1999.

[Sid08] SideEffects. 2008. Leading the 3D Animation Industry. *Side Effects Software Inc.* [Online] 2008. [Zitat vom: 9. 12 2008.] http://www.sidefx.com/index.php?option=com_content&task=view&id=22&Itemid=51.

[Sid081] SideEffects. 2008. Press. *Side Effects Software Inc.* [Online] 2008. [Zitat vom: 9. 12 2008.] http://www.sidefx.com/index.php?option=com_content&task=blogcategory&id=41&Itemid=55.

[And05] Skowronski, Andrew. 2005. XSIBlog - Blog Archive - Becoming a Scriptmaster. *XSIBlog.* [Online] 24. 11 2005. [Zitat vom: 3. 2 2009.] http://www.xsi-blog.com/archives/67.

[Sof09] Softimage (Company). *Wikipedia.* [Online] 14. 1 2009. [Zitat vom: 28. 1 2009.] http://en.wikipedia.org/wiki/Softimage.

[Sof091] Softimage XSI. *Wikipedia.* [Online] 21. 1 2009. [Zitat vom: 27. 1 2009.] http://en.wikipedia.org/wiki/Softimage_XSI.

[Sof08] Softimage. 2008. *XSI SDK Documentation.* [Kompilierte HTML Hilfe Datei] s.l. : Softimage, 2008.

[Tic08] Ticca, Fabio. 2008. Pipeline-Scripting in XSI. *Digital Production.* 2008, 4.

[Tle08] Tleisher. 2008. Maya - CGWiki. *CGWiki.* [Online] 16. 3 2008. [Zitat vom: 11. 12 2008.] http://wiki.cgsociety.org/index.php/Maya.

[War07] Warner, Steve, Phillips, Kevin und Albee, Timothy. 2007. *Essential Lightwave 9.* Plano, Texas : Wordware Publishing Inc., 2007.

[Wik09] Greeble. *Wikipedia.* [Online] Wikipedia, 2 2009. [Zitat vom: 21. 3 2009.] http://en.wikipedia.org/wiki/Greeble.

[Nik08] Wirth, Niklaus. 2008. *Grundlagen und Techniken des Compilerbaus.* München : Oldenbourg Wissenschaftsverlag GmbH, 2008.

[XSI08] ***XSI Documentation***. [kompilierte HTML Hilfe] 2008.