HOCHSCHULE FURTWANGEN UNIVERSITY HFU

Fakultät
Digitale Medien

SS 2013

Bachelorthesis

Dominik Steffen

Implementierung der Half-Edge Datenstruktur zu Manipulation und Handling Dreidimensionaler Meshes insbesondere durch den Einsatz von LINQ und Lambda Ausdrücken in Microsofts C#.

Prof. Christoph Müller
Prof. Wilhelm Walter

Abstract

Diese Arbeit zeigt, wie eine Implementierung der Half-Edge Datenstruktur für das Handling von Mesh Daten umgesetzt werden kann. Die Implementierung des vorgestellten Konzeptes erfolgte in C# und ist in die Echtzeit 3D Engine Fusee der Hochschule Furtwangen integriert. Geometrie Daten können mit der Datenstruktur im Speicher gehalten und manipuliert werden. Im Kern dieser Thesis wird überpüft wie sich LINQ Abfragen in der Datenstruktur realisieren lassen, ob sie zur Laufzeit Auswirkungen auf die Performance des Systems haben. Ein Ausblick erörtert wie LINQ die zukünftige Formulierung von Geometrie manipulierenden Algorithmen im Fusee Projekt beeinflussen könnte. Die Implementierung von LINQ For Geometry bietet Iteratoren als Schnittstelle für Benutzer des Systems, mit welchen sich neue komplexe Transformations Algorithmen realisieren lassen. Es ist angedacht, das Projekt zukünftig eventuell als Datenstruktur für einen Mesh Editor auf Basis der Fusee Engine einzusetzen. Das Ergebnis, der Code der Arbeit, ist auf GitHub frei zugänglich und befindet sich in einem seperaten Branch des Fusee Projektes.

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Bachelorthesis selbständig und ohne unzulässige fremde Hilfe angefertigt habe. Alle verwendeten Quellen und Hilfsmittel die sowohl zum schreiben dieser Arbeit als auch zum Entwickeln des dazugehörigen Quellcodes benutzt wurden, habe ich angegeben.

Dominik Steffen, Küssaberg den 9. August 2013

LINQ FOR GEOMETRY / LFG

„Focused, hard work is the real key to success. Keep your eyes on the goal, and just keep taking the next step towards completing it. If you aren't sure which way to do something, do it both ways and see which works better.“

- John Carmack, co-founder of idSoftware

Inhaltsverzeichnis

Abstract 2

Eidesstattliche Erklärung 3

Inhaltsverzeichnis 5

1 Einleitung 1

1.1 Motivation . . . 1

1.2 Ziele von LINQ For Geometry . . . 2

1.3 Gliederung dieser Arbeit . . . 2

1.4 Verwendete Software . . . 3

2 Hauptteil 4

2.1 Die Half-Edge Datenstruktur . . . 4

2.1.1 Die Half-Edge Datenstruktur in LINQ For Geometry . . . 5

2.1.2 Vorteile der Half-Edge Datenstruktur . . . 9

2.1.3 Nachteile der Half-Edge Datenstruktur . . . 10

2.2 Aktueller Forschungsstatus auf dem Gebiet der 3D Mesh Datenstrukturen . . . 12

2.2.1 Probleme in der aktuellen Forschung . . . 13

2.3 Einführung zu LINQ in C# . . . 13

2.3.1 Was ist LINQ to Objects genau? . . . 15

2.3.2 Abfragesyntax in LINQ . . . 16

2.3.3 Methodensyntax in LINQ . . . 16

2.4 Einführung zu Lambda Ausdrücken in C# . . . 19

2.4.1 Lambda Ausdrücke in Verbindung mit LINQ . . . 19

2.5 Das Software Projekt LINQ For Geometry . . . 19

2.5.1 Konzept zu LINQ For Geometry . . . 21

2.5.2 Furtwangen University Simulation and Entertainment Engine . . . 22

2.6 Die Vorteile in der Entwicklung mit managed Programmiersprachen . . . 23

2.6.1 Warum die Programmiersprache C# für LINQ For Geometry? . . . 24
2.6.2 Der Link von LINQ zur Half-Edge Datenstruktur 25
2.7 Der Import von Geometriedaten im „Wavefront Object“ Format 25
2.7.1 Warum das Wavefront Object Format 26
2.7.2 Der Importer für das Wavefront Format oder auch Face basierter Import - Edge basiertes Handling 27
2.8 Initialisierungsablauf und Aufbau der Half-Edge Datenstruktur in LINQ For Geometry 27
2.9 Implementierung und Funktion der Handler für die einzelnen Komponenten der HES 29
2.9.1 Beispiel eines Handler Konstruktes und seiner Implementierung 29
2.10 Pointer Container und ihre Aufgabe in LINQ For Geometry . . 31
2.10.1 Half-Edges Pointer-Container 32
2.10.2 Edges Pointer-Container 33
2.10.3 Vertices Pointer-Container 33
2.10.4 Faces Pointer-Container 34
2.11 Die Geometrie in LINQ For Geometry 36
2.12 Anwendungsszenarien von LINQ For Geometry 37
2.13 LINQ Abfragen und ihr Verhalten bei der Selektierung großer Datenmengen in LINQ For Geometry 38
2.14 Stern- und Umlaufenumeratoren (Iteratoren/Enumeratoren) . . 39
2.14.1 Die Verwendung von LINQ und Lambda in den Enumeratoren 42
2.14.2 Unterschiedliche Implementierungen von Iteratoren kurz dargestellt 42
2.14.3 Die Geschwindigkeit der Half-Edge Datenstruktur in den Enumeratoren 44
2.14.4 Zusammenfassung der Erkenntnisse aus den Messwerten 50
2.14.5 Benchmarking mit verschiedenen 3d Modellen 51
2.15 Manipulation von Mesh Daten in der Datenstruktur 53
2.15.1 Manipulation von Edges und Half-Edges 53
2.15.2 Manipulation von Vertices 53
2.15.3 Manipulation von Faces 54
2.16 Beispielhafte Implementierung von Standard Algorithmen zur Geometriemanipulation als Modul 54
2.16.1 Transformation mit einer Transformationsmatrix 54
2.17 Berechnungen für die Darstellung eines Meshes in der Fusee Engine 56

3 Schluss **59**
3.1 Fazit 59
3.2 Ausblick auf die zukünftige Verwendung von LINQ For Geometry 60
3.3 Wo ist der Code und die Dokumentation zu LINQ For Geometry zu beziehen und einzusehen? 61

Verzeichnis der Quellcode Beispiele **63**

Tabellenverzeichnis **64**

Abbildungsverzeichnis **65**

Literaturverzeichnis **68**

4 UML Diagramme **69**
4.0.1 UML Diagramme zur Implementierung von LINQ For Geometry 69

1 Einleitung

1.1 Motivation

Um 3D Modelle in Echtzeit Engines darzustellen werden diese meistens in einer Face basierten Datenstruktur gespeichert. Durch diese Art der Speicherung ist es zwar möglich, aber sehr aufwendig, performante Algorithmen zur Manipulation der Daten in Echtzeit zu formulieren. Sollte dies gelingen sind die gewählten Algorithmen meist kompliziert und der Einstieg für neue Entwickler fällt meist schwer. Eine Face basierte Datenstruktur bietet kaum Vernetzungen der enthaltenen Elemente und erschwert das Formulieren von Lösungen durch die Notwendigkeit vieler Schleifen. Die Laufzeiten werden lang und Kanten basierte Operationen lassen sich durch eine solche Struktur gar nicht oder durch erheblichen rechnerischen Aufwand realisieren. Die Half-Edge Datenstruktur ist eine Datenstruktur welche diese Probleme umgehen kann. Elemente eines Meshes sind untereinander vernetzt und Iterationen auf dem Mesh können schnell ausgeführt werden.

Um diese Datenstruktur in `C#` zu implementieren und den Zugriff darauf zu verbessern, beschäftigt sich diese Arbeit mit der Frage wie die Half-Edge Datenstruktur in `C#` mit LINQ und Lambda Support umgesetzt werden kann und wo die Vorteile und Nachteile eines solchen Projektes liegen. LINQ For Geometry ist ein Projekt, welches sich während seiner Entwicklung stetig weiter entwickelt hat. Zuerst wurde nur eine Datenstruktur entwickelt. Um sie allerdings zu testen und den Nutzen zu erforschen wurde die Datenstruktur in eine Echtzeit 3D Engine namens Fusee integriert.

Dieser Text beschreibt auch, einige der wichtigsten Iteratoren für die Manipulation einzelner geometrischer Daten oder ganzer Models/Meshes zur Laufzeit. Es wird geprüft, ob LINQ Abfragen in jeder Situation verwendet werden können oder ob mit Blick auf die Performance des Systems mit Problemen zu rechnen ist. Es werden einige standard Geometrie Manipulationen vorgestellt und ihre Implementierung im Projekt wird beschrieben. Schliesslich möchte diese Arbeit einen Ausblick darauf geben was mit solch einem System in einer

Engine wie Fusee getan werden kann und was die möglichen Einsatzgebiete von LINQ For Geometry in Zukunft sein könnten.

Der Autor dieser Arbeit interessiert sich bereits seit einigen Jahren für 3D Echtzeit Engines und Interaktive Software und der Funktionsweise der einzelnen Bereiche hinter einer solchen Engine. Im Laufe des Studiums wurden einige kleine Softwareprojekte und Spiele entwickelt und so war es der nächste logische Schritt etwas tiefer in die Computergrafik und Mesharchitektur einzusteigen. Menschen wie John Carmack (idSoftware), John Romero, Markus „Notch“ Persson und viele andere, waren eine Inspiration während des Studiums und haben den Schreiber dieser Arbeit stetig weiter motiviert.

1.2 Ziele von LINQ For Geometry

LINQ For Geometry hat es sich als Ziel gesetzt die Half-Edge Datenstruktur in einem C# Projekt einmal grundlegend zu implementieren und herauszufinden wie sehr C# 's LINQ Abfragen in einem solchen System von Nutzen sind. Die Entwicklung des Softwareprojekts stand hierbei im Vorderung und entwickelte sicher stetig weiter. LINQ For Geometry wurde mit der Zeit in die Echtzeit 3D Engine Fusee integriert und besitzt aktuell einen eigenen Branch im Fusee Projekt. Das Projekt ist vorerst dazu gedacht, Daten von 3D Meshes im Speicher zu halten und ihre Werte über bereitgestellte Iteratoren und Transformationen zu manipulieren. Diese Arbeit zielte nicht darauf ab, ein komplett abgeschlossenes Projekt zu erschaffen. Die von LINQ For Geometry angebotenen Iteratoren sollen viel mehr als eine Ausgangslage darstellen und wiederverwendet werden um in Zukunft durch ihre Kombination aus bestehenden Methoden neue Möglichkeiten der Transformation in LFG zu schaffen oder bereits bekannte bewährte Algorithmen zu implementieren.

1.3 Gliederung dieser Arbeit

Zunächst soll ein Ausblick darüber gegeben werden, was eine Half-Edge Datenstruktur eigentlich ist und wo ihre Vor- und ihre Nachteile liegen. Es folgt eine Einführung in LINQ to objects und den nutzen dieses C# Sprachkonstruktes für das LINQ For Geometry Projekt. Natürlich soll auch die Implementierung des Projekts und die Probleme während der selben nicht ausser Acht gelassen werden. Später wird dieser Text noch auf die Vor- und Nachteile der Half-Edge Datenstruktur in Verbindung mit LINQ und der Verwendung der beiden im Projekt eingehen. Das Projekt wurde mit Hilfe des Visual Studio 2010 Profiles zur Laufzeit analysiert und aus den Ergebnissen wurden Hinweise für die Verbesserung des Codes gezogen. Benchmarking Ergebnisse belegen die dort

getroffenen Entscheidungen und die gewählten Methoden. Ein Abschnitt über die wichtigen Iteratoren und Transformationen, wie das Triangulieren eines Meshes, schliesst den Hauptteil dieser Arbeit ab. Der Schluss fasst nochmals die wichtigsten Einzelheiten und Erkenntnisse zusammen und gibt einen Ausblick auf die möglicherweise zukünftigen Verwendungszwecke von LINQ For Geometry.

1.4 Verwendete Software

- Microsoft Visual Studio 2010,
  verwendet als Entwicklungsumgebung für das Softwareprojekt.
- Die Erweiterung ReSharper in Version 7.1 für Visual Studio 2010 `http://www.jetbrains.com/resharper`
- Umlet `http://www.umlet.com/`
  Ein kostenloses Tool um UML Diagramme zu erstellen.
- GitHub und die GitShell `www.github.com`
  Verwendet als Versionskontrollsystem und als Distributionswerkzeug für den Quellcode.
- TexWorks `www.tug.org/texworks/`
  Zum Schreiben dieser Arbeit.

2 Hauptteil

2.1 Die Half-Edge Datenstruktur

Die Half-Edge Datenstruktur, oft auch als „Doubly-connected edge list“ bezeichnet, ist eine Datenstruktur für planare Graphen. Sie ist im Allgemeinen sehr flexibel einsetzbar. Diese Thesis Arbeit setzt die Half-Edge Datenstruktur dazu ein, die wichtigsten Informationen eines dreidimensionalen Computermodells zu repräsentieren und sie für eine spätere Verwendung im Speicher bereit zu halten. Sehr oft werden Informationen eines Computermodells ohne relationale Verbindungen gespeichert. Bei diesen so genannten Face basierten Datenstrukturen handelt es sich um sehr einfache Datenstrukturen, die keinen großen Spielraum für Optimierungen und Erweiterungen lassen. Bei Iterationen und geometrischen Transformationen sind Face basierte Strukturen wesentlich langsamer als eine Edge basierte Struktur (dazu später mehr) und werden meist nicht für geometrische Operationen, sondern nur zum Speichern der Daten im Primär oder Sekundärspeicher des Rechners verwendet. Die Half-Edge Datenstruktur wird für dieses Projekt sogar ein wenig bezüglich ihres Umfangs erweitert ohne aber die grundsätzlichen Gegebenheiten der Datenstruktur zu beeinflussen.

Ein Kernelement der Half-Edge Datenstruktur und sicherlich ihre bekannteste und herausragendste Eigenschaft ist, dass jede Kante im gespeicherten Modell (Mesh) als Kombination aus zwei Half-Edges repräsentiert wird. Zwei Half-Edges zusammen ergeben also eine komplette Kante.

Die Einsatzgebiete der Half-Edge Datenstruktur sind vielfältig. Sie wird wie von Mark de Berg, Otfried Cheong, Marc van Kreveld und Mark Overmars in Computational Geometry [BCKO08] (Auflage 3) beschrieben

> „dazu eingesetzt Geografische Gegebenheiten etc. auf digitalen Landkarten als Thematische Karten Overlays dazustellen.“ [BCKO08, S. 29]

Ebenso häufig findet man sie in der 3D Computergrafik, wenn auch selten in

größeren Projekten. Meist wird sie als Datenstruktur für kleinere Softwarelösungen herangezogen. Ein Beispiel ist hier die Demo Szene [1] der Computergrafik. Das Entwickler Team „farbrausch“, das verantwortlich ist für Projekte wie das Preis gekrönte „kkrieger“ (ein minimalistischer Ego Shooter mit einer Gesamtgröße von 96kb) hat die Half-Edge Datenstruktur für ihr Demo Projekt „debris“ genutzt. [2]

2.1.1 Die Half-Edge Datenstruktur in LINQ For Geometry

Häufig wird in der Thematik die Bezeichnung „Doubly-connected Edge List“ auch als Synonym für die Bezeichnung „Half-Edge Data Structure“ verwendet. Im Hinblick auf die Implementierung dieser Arbeit ist das so nicht ganz richtig. Dieser Text bezieht sich im Falle der „Doubly-connected Edge List“ immer auf die Darstellung von [BCKO08]. Es existieren, besonders in diesem Projekt, einige wenige feine Unterschiede der beiden Datenstrukturen.
Die Half-Edge Datenstruktur besteht grundsätzlich aus vier verschiedenen Datensätzen (data records). Dazu zählen:

- Vertex, ein Punkt im dreidimensionalen Raum.
- Face, zu betrachten als Polygon (Geometrisches Vieleck mit genau oder mehr als 3 Eckpunkten). Erweitert um zusätzliche Informationen.
- Half-Edge, der wichtigste Datensatz in der Datenstruktur. Eine halbe Kante welche in der Half-Edge Datenstruktur die meisten Informationen über ein dreidimensionales Modell (Mesh) enthält.
- Edge, eine Kante welche aus zwei Halbkanten besteht.

  „Jeder Datensatz (Half-Edge, Vertex und Face) der Datenstruktur kann ebenfalls dazu benutzt werden Informationen zu speichern welche nicht im direkten Zusammenhang mit den geometrischen Informationen des Meshes stehen. Es ist in etwa möglich, Informationen zur Beleuchtungsberechnung (Vertex- und Facenormalen) oder Textur Koordinaten eines Meshes in den Datensätzen zu speichern. Diese zusätzlichen Informationen werden im folgenden als Attributs-Informationen bezeichnet.“ [BCKO08, S. 31]

Zu beachten ist hierbei, dass die Half-Edge Datenstruktur durch ihr wichtigstes Element auch als gerichteter Graph betrachtet werden kann. Jede Half-Edge, in der Graphentheorie als Bogen bezeichnet, ist genau einem Vertex, in der

[1] Eine Demo ist in der Demo Szene als Digitale Kunst zu verstehen. Meist ist eine Demo eine Form der Computergrafik Echtzeit-Animationen inklusive Musikuntermalung.
[2] Viele farbrausch Projekte sind zu finden unter http://www.farb-rausch.de/ (Stand: Juli 2013)

Graphentheorie als Knoten bezeichnet, zugeordnet. Dieser Zustand ergibt sich durch die Speicherung des Pointers an jeder Half-Edge, der auf den Vertex verweist auf den die Half-Edge zeigt.

> „Ein gerichteter Graph $G = (V, E)$ besteht aus einer Knotenmenge V und einer Bogenmenge E, sodass jedem Bogen (jeder Kante) $e = (u, v)$ eindeutig ein geordnetes Paar (u, v) von Knoten aus V zugeordnet ist.“ [Tit11, S. 127]

> In der Half-Edge Datenstruktur sind die Half-Edges also Kanten mit einer Orientierung. Der Vertex dem sie entspringen ist dabei als der Ursprung zu bezeichnen, während der Vertex auf den sie zeigen und auf den sie einen Zeiger enthalten, als Ziel (Destination) betrachtet wird. vgl. [BCKO08, S. 31]

> „Because half-edges are oriented we can speak of the *origin* and the *destination* of a half-edge.“ [BCKO08, S. 31]

Die Half-Edge Datenstruktur ist also mathematisch betrachtet ein planarer Graph. Laut Eulerscher Polyederformel beschrieben von Peter Tittmann [Tit11, S. 47–50] ergibt sich folgender Sachverhalt. $v + f = k + 2$ v entspricht der Anzahl der Punkte oder Vertices in einem Graphen, f die Anzahl der Flächen oder Faces und k entspricht der Anzahl der Kanten. Wendet man diesen Ausdruck auf einen Würfel an, so ensteht die folgende Gleichung $8 + 6 = 12 + 2$. Es ist also hier tatsächlich so, dass ein Würfel als planarer Graph betrachtet werden kann. Der gleiche Sachverhalt ergibt sich nun in der Repräsentation eines Würfels in der Half-Edge Datenstruktur.

Doubly Connected Edge List

Die doubly connected edge list (kurz DECL) wie in [BCKO08] erwähnt ist der hier verwendeten Half-Edge Datenstruktur ähnlich. Die Strukturen haben nur ein paar wenige Unterschiede. Zuerst einmal ein paar Worte zu den Gemeinsamkeiten.

> Die Doubly-connected edge list besteht grundsätzlich ebenfalls aus drei verschiedenen Typen von Datensätzen. Dazu zählen, wie auch in der Half-Edge Datenstruktur , Vertices, Faces und Half-Edge Datensätze. vgl. [BCKO08, S. 31]

Somit ist bereits klar, dass sich die Datenstruktur in der Implementierung durch den in diesem Projekt hinzugekommenen data record „Edge“ unterscheidet. Die Doubly-connected edge list unterscheidet sich weiterhin im Blick auf die Zeiger, welche in einem data record gespeichert werden. Während die hier

implementierte Half-Edge Datenstruktur in einem Half-Edge record folgende Informationen speichert:

- Zeiger auf ihre Twin Half-Edge, auch Nachbar Half-Edge genannt.
- Zeiger auf die nächste Half-Edge im Uhrzeigersinn (Ist variabel implementierbar. LINQ For Geometry implementiert die Zeiger im Uhrzeigersinn)
  - Sowohl in OpenGL als auch in Direct3D kann eingestellt werden ob die Ausrichtung von Faces anhand des Uhrzeigersinns (CW) oder gegen den Uhrzeigersinn (CCW) bestimmt wird. Direct3D benutzt hier die CW Richtung als Standard[3], OpenGL benutzt als Standard die CCW order. [4]
- Zeiger auf den Vertex auf den die Half-Edge zeigt
- Zeiger auf das Face zu welchem die Half-Edge gehört

Die Doubly-connected Edge List laut [BCKO08] speichert ein wenig mehr Informationen.

- Zeiger auf ihre Twin Half-Edge.
- Zeiger auf die nächste Half-Edge im Uhrzeigersinn.
- Zeiger auf die vorige Half-Edge im Uhrzeigersinn.
- Zeiger auf den Vertex von dem die Half-Edge ausgeht.
- Zeiger auf das Face zu welchem die Half-Edge gehört.

Somit ist zu erkennen, dass sich die hier verwendete Struktur von der DCEL auf jeden Fall dadurch unterscheidet, dass sie a) Keinen Pointer auf die vorige Half-Edge speichert und b) Ihr Ziel anstatt ihres Ursprungs speichert.
Allerdings sind die Quellen hierzu nicht immer konsistent da beide Datenstrukturen nirgendwo in einer definitiven Form beschrieben werden. So können sich verschiedene Implementierungen der gleichen Datenstruktur in Feinheiten unterscheiden. Auch die Namen der Doubly-connected Edge List und der Half-Edge Datenstruktur wird wie schon genannt oft synonym, oft aber auch als Unterscheidung benutzt. Das LINQ For Geometry Projekt vergleicht hier um einen festen Bezugspunkt zu erhalten also immer die Doubly-connected edge list in der Darstellung von [BCKO08, S. 30 und Folgende] mit der eigenen Implementierung der Half-Edge Datenstruktur.
Diese Verbindungen und Beziehungen der data records in der Half-Edge Datenstruktur werden im nächsten Abschnitt genauer erläutert.

[3] `http://msdn.microsoft.com/en-us/library/windows/desktop/bb204882(v=vs.85).aspx` (Stand: 23. Juli 2013)
[4] `http://www.opengl.org/wiki/Face_Culling` (Stand: 23. Juli 2013)

Verbindungen und Beziehungen in der Half-Edge Datenstruktur

Die Half-Edge Datenstruktur bedingt einige Vernetzung unter den einzelnen data records. Es folgt eine Auflistung, welche Zusammensetzung an Zeigern jeder der vorhandenen „Data Records“ enthält. Anhand einer Skizze eines viereckigen Polygons(Abbildung 2.1), werden die Beziehungen der „Data Records“ untereinander illustriert.

Es folgt eine Auflistung der einzelnen, in der Half-Edge Datenstruktur existierenden, „Data Records“.

- Half-Edge
- Edge
- Face
- Vertex

Die Half-Edge ist damit der wichtigste Informationsträger in der Datenstruktur. Sie enthält als einziges Element Verbindungen zu allen anderen Elementen der Datenstruktur.
Diese Informationen sind an einer Half-Edge gespeichert.

- Ein Zeiger auf ihre twin Half-Edge (Der Nachbar der Half-Edge).
- Ein Zeiger auf den Vertex auf den sie hinführt.
- Ein Zeiger auf das zugehörige Face.
- Ein Zeiger auf die nächste Half-Edge im Uhrzeigersinn.

Eine Edge enthält gerade noch zwei Informationen:

- Ein Zeiger auf die erste Half-Edge der Edge.
- Ein Zeiger auf die zweite Half-Edge der Edge.

Ein Face speichert nur eine Information:

- Ein Zeiger auf eine beliebige das Face begrenzende Half-Edge. Vorzugsweise die erste für das Face hinzugefügte Half-Edge.

Der Vertex speichert eine Information:

- Eine beliebige Half-Edge die von ihm ausgeht, dabei wird meist die erste Half-Edge benutzt die an den Vertex angelegt wird.

Abbildung 2.1: Diese Illustration zeigt die Verbindung der data records an einem beliebigen Polygon durch Pfeilverbindungen.

An dieser Stelle sei bereits vorweg genommen, dass LINQ For Geometry in seiner Implementierung diese data records um ein paar Beziehungen, welche durch die Verbindung zur Echtzeit 3D Engine Fusee nötig werden, erweitert.

2.1.2 Vorteile der Half-Edge Datenstruktur

Die vielen Verbindungen unter den einzelnen Komponenten in der Half-Edge Datenstruktur machen es möglich sehr schnelle Iterations Algorithmen zu implementieren. Die Normale[5] eines Faces kann sehr effizient berechnet werden. Es können auch die Normalen für beliebige Faces unabhängig vom Rest des Meshes berechnet werden. Man benötigt dafür lediglich den Index eines Faces als Einstiegspunkt in die Berechnung.

[5]Normale: Orthogonaler Vektor auf einer Ebene. In der Computergrafik oft dazu benutzt um herauszufinden welche Seite des Faces „vorne“ ist. Ebenfalls oft für Beleuchtungsberechnung benutzt.

Subdivision Surface Algorithmen wie Catmull-Clark oder Loop können mit der Datenstruktur (oder jeder anderen Edge basierten Datenstruktur) besonders „kostengünstig“ realisiert werden. Catmull-Clark benötigt dazu ein Mesh aus Quads während man mit Loop ein Mesh aus Triangles verarbeiten könnte. Beide Mesh Typen, Quads und Triangles, werden vom LINQ For Geometry Projekt unterstützt. Die Fusee Engine, in welche das LINQ For Geometry Projekt integriert ist, kann im Moment allerdings nur mit Meshes bestehend aus Triangles arbeiten. Um trotzdem mit Fusee zu arbeiten, trianguliert dieses Projekt ein Mesh vor der Übergabe an Fusee.

Die Half-Edge Datenstruktur ist also grundsätzlich sehr flexibel. Es bedingt praktisch nicht, dass die Anzahl der Eckpunkte jedes Faces eines Meshes gleich groß ist. LINQ For Geometry allerdings, in seiner jetzigen Form, erwartet entweder Triangulated Meshes, Meshes aus Quadlaterals (Quads) oder eine Mischung aus beidem. Dies ist dem Umstand geschuldet, dass ein Mesh zur weiteren Verwendung in Fusee konvertiert und verarbeitet werden muss.

2.1.3 Nachteile der Half-Edge Datenstruktur

Es gibt selbst in einer so flexiblen Datenstruktur wie der Half-Edge Datenstruktur Nachteile, die während der Planungsphase oder Implementierung auftreten können. Durch ihren Aufbau und die verschiedenen Verbindung welche während einer Initialisierung aufgebaut werden müssen ist die Datenstruktur sehr komplex. Es erfordert einiges an Zeit und Überlegungen ein System für eine Implementierung so zu entwerfen, dass es alle Anforderungen an eine Echtzeit 3D Datenstruktur erfüllt. Vor einer Implementierung der Half-Edge Datenstruktur sollte also gut überlegt sein, ob sie für das gestellte Problem eine vernünftige Art der Lösung darstellt. In LINQ For Geometry ist die Datenstruktur eine sehr elegante Lösung. Sie ist hinreichend flexibel um eine mögliche Lösung für die Problemstellung „(Editierbare) Echtzeit 3D Datenstruktur“ darzustellen. Die Half-Edge Datenstruktur ist aber nicht als Universallösung zu betrachten. Wenn der Entwickler nur einige wenige spezielle Operationen auf einem Mesh ausführen möchte, so gibt es möglicherweise eine effizientere und schnellere Datenstruktur um dies zu bewerkstelligen.
Ein auffälliger Unterschied zu einfacheren Face basierten Datenstrukturen ist der massiv erhöhte Overhead, der bei der Erstellung der Datenstruktur entsteht. Im nächsten Abschnitt wird ein kleines Beispiel exemplarisch den Speicherverbrauch der Half-Edge Datenstruktur bestimmen.

Speicherverbrauch im Gegensatz zu Face basierten Datenstrukturen

Der Overhead, der bei der Repräsentation eines Meshes für eine Half-Edge Datenstruktur entsteht, ist um einiges höher als der Overhead einer Face basierten Lösung. Sollte man also nur einfache Aufgaben auf einem Mesh erledigen wollen, ist an sehr wenig Speicher gebunden, oder einige wenige Operationen nur einmal ausführen wollen so wäre es wohl ratsam eine andere Datenstruktur dafür zu nutzen. Allerdings ist es heute in der Computergrafik meistens so, dass genügend Speicherplatz im Primärspeicher unserer Rechner und Grafikkarten zur Verfügung steht um diesen Sachverhalt zu verschmerzen.

Es folgt die beispielhafte Berechnung des Speicherbedarfs eines Würfels. Man gehe davon aus, jedes gespeicherte Element (Daten in Form von Vertices, Faces, Edges, Half-Edges) wird mit 32bit repräsentiert. Jeder Zeiger (Pointer) wird mit 8bit repräsentiert. In diesem Fall könnte man den Speicherverbrauch zur Laufzeit des Programms im Hauptspeicher des Rechners (RAM) eines einfachen sechs Seitigen Würfels wie folgt beschreiben.

- 24 Half-Edges jede beansprucht 32 bit
    1. Ein Vertex Pointer mit 8bit
    2. Ein Twin Pointer mit 8bit
    3. Ein Next Pointer mit 8bit
    4. Ein Face Pointer mit 8bit
- 12 Edges jede beansprucht 32 bit
    1. Zwei Half-Edge Pointer mit je 8bit
- 6 Faces jedes beansprucht 32 bit
    1. Half-Edge Pointer eins mit 8bit
- 8 Vertices jeder beansprucht 32 bit
    1. Half-Edge Pointer eins mit 8bit

Daraus würden sich nun folgende Berechnungen ergeben:
Half-Edge Datenmenge: $24 * (32bit + 8bit + 8bit + 8bit + 8bit) = 1536bit$
Edge Datenmenge: $12 * (32bit + 8bit + 8bit) = 576bit$
Face Datenmenge: $6 * (32bit + 8bit) = 240bit$
Vertex Datenmenge: $8 * (32bit + 8bit) = 320bit$
In Summe sind das 2672bit = 334 Byte.

Eine Face basierte Lösung würde den Würfel so darstellen:

- 6 Faces jedes beansprucht 32 bit
    1. Vier Vertex Pointer mit 8bit pro Face
- 8 Vertices jeder beansprucht 32 bit

Der Datenverbrauch würde wie folgt berechnet:
Face Datenmenge: $6 * (32bit + 8bit + 8bit + 8bit + 8bit) = 384bit$
Vertex Datenmenge: $8 * 32bit = 256bit$
In Summe sind das 640bit = 80Byte.

Das Beispiel zeigt also, dass bei einem einfachen Mesh der Speicherverbrauch der Half-Edge Datenstruktur mit 334Byte um ca. das vierfache höher liegt als der Verbrauch einer Face basierten Lösung mit 80 Byte. Somit ist eine Face basierte Datenstruktur sparsamer was den Speicher betrifft. Allerdings tauscht man in diesem Falle auch Geschwindigkeit gegen Speicherplatz ein. Die Half-Edge Datenstruktur mag zwar komplex sein und hat einen höheren Speicherverbauch, allerdings ist sie bei Iterationen in Kantenbasierten Algorithmen auch schneller als die Face basierte Datenstruktur.

2.2 Aktueller Forschungsstatus auf dem Gebiet der 3D Mesh Datenstrukturen

Die Half-Edge Datenstruktur oder auch die Doubly-connected edge list (DCEL) werden heute zwar relativ häufig für 3D Mesh Datenstrukturen eingesetzt, allerdings ist zum Zeitpunkt dieser Arbeit nicht bekannt ob und welche großen Programme diese Datenstruktur zum Handling ihrer 3D Daten einsetzt. Weder für 3DS Max von Autodesk noch für Cinema 4D von Maxon konnten verlässliche Quellen gefunden werden. Die Open Source Software „Blender“ verwendet in der Version 2.6 die BMesh Structure. Hierbei handelt es sich um eine vom Blender Team entwickelte Datenstruktur, welche dazu gedacht ist Limitierungen einer älteren von Blender genutzen Datenstruktur zu beheben. BMesh[6] ist eine an die Radial Edge Structure [7] angelehnte Struktur. Sie ist der Half-Edge Datenstruktur ähnlich, beide weichen allerdings im Aufbau doch so stark von einander ab um sie sicher zu unterscheiden (Hier sei nur kurz angemerkt, dass BMesh nur auf komplette Kanten setzt und nicht auf Halbkanten). Wo wird die Half-Edge Datenstruktur also eingesetzt? Bei den Recherchen zu dieser Arbeit wurde festgestellt, dass viele

[6] `http://wiki.blender.org/index.php/Dev:2.6/Source/Modeling/BMesh/Design` (geprüft 30.07.2013)

[7] Weiler, K.J. : The Radial Edge Structure: A Topological Representation for Non-Manifold Geometric Modeling. in Geometric Modeling for CAD Applications, Springer Verlag, May 1986.

Studenten und Akademiker diese Datenstruktur für kleine Projekte in der Computergrafik nutzen. Es existieren einige exemplarische Implementierungen ohne die Integration in eine Engine, meistens jedoch in C++ geschrieben und dementsprechend ohne die Nutzung und Unterstützung von LINQ. Keines der gefundenen Projekte war in eine Echtzeit 3D Engine integriert. Es wurden auch keine geometrischen Transformationen auf den Datenstrukturen präsentiert. Es soll hier aber auf Ryan Holmes Arbeit [Hol12] verwiesen werden. Er hat ein interessantes Projekt Namens „The DCEL Data Structure for 3D Graphics“[8] auf Basis der Doubly-connected edge list erstellt. Es vereinfacht den Einstieg in die Thematik und erklärt interaktiv die Topologie der Half-Edge Datenstruktur und der DCEL.

Sicherlich kann diese Arbeit auch nicht fortfahren ohne die Erwähnung des OpenMesh[9] Projektes, betreut von Prof. Dr. Leif Kobbelt in der Computer Graphics Group der RWTH Aachen. OpenMesh ist eine generische Datenstruktur für die Repräsentation und Manipulation von polygonalen Mesh Daten [Kob13]. OpenMesh war in den Anfängen dieser Arbeit ein Vorbild für das LFG Projekt ohne jedoch den Anspruch, die Funktionen von OpenMesh nachzubilden. Open Mesh ist zudem Open Source, generisch und in C++ implementiert was einen Vergleich mit LFG zu schwierig macht. LFG orientiert sich in seiner Umsetzung jedoch ausdrücklich am Modell der Half-Edge Datenstruktur , wie sie durch M.Botsch, S.Steinberg, S.Bischoff und L.Kobbelt im Paper „OpenMesh - a generic and efficient polygon mesh data structure“ [BSBK02] erläutert wird.

2.2.1 Probleme in der aktuellen Forschung

Das Problem der aktuellen Forschung ist also, dass die Half-Edge Datenstruktur nur sehr selten in C# , wenn überhaupt nur testweise (meist sehr minimal oder sogar nur in Pseudocode), implementiert wurde und keine der Lösungen einen Support für LINQ Abfragen und Lambda Ausdrücke bietet. Somit ist LFG in Verbindung mit der Fusee Engine und dem LINQ Support eine neue Entwicklung auf Basis bewährter Strukturen.

2.3 Einführung zu LINQ in C#

Language Integrated Query, kurz LINQ und zu deutsch Sprachintegrierte Abfrage, wurde mit C# 3.0 und .NET 2.0 in die .NET Umgebung integriert. LINQ ist also ein Produkt von Microsoft und erinnert auf den ersten Blick

[8] Zu finden unter `http://www.holmes3d.net/graphics/dcel/`(geprüft 30.07.2013)
[9] Zu finden unter `http://www.openmesh.org/` (geprüft 29.07.2013)

in seinem syntaktischen und semantischem Aufbau stark an die Software MySQL, eine Sprache für die Verwaltung relationaler Datenbanken, welche aktuell (Stand Juli 2013) von der Oracle Corporation betreut wird. Beide Projekte unterscheiden sich jedoch wesentlich. LINQ ist in erster Linie ein Feature, das direkt auf Objekten arbeitet wohingegen MySQL auf relationale Datenbanken angewendet wird. Allerdings kann LINQ viel mehr als das. LINQ Statements werden zur „Compile Zeit“ auf die Korrektheit der Syntax überprüft und von Microsofts IntelliSense zum Zeitpunkt der Entwicklung auf Fehler untersucht. Was einen grössen Vorteil des Systems im Gegensatz zu Sprachen wie MySQL darstellt, da Statements in dieser Sprache vor Ausführung keiner syntaktischen Prüfung unterzogen werden.

LINQ kann als Layer und Bindeglied zwischen Collections aus Daten und den Programmiersprachen des .NET Frameworks betrachtet werden. Es ist hierbei möglich, LINQ in verschiedenen Bereichen zu nutzen. Es arbeitet sowohl wie bereits erwähnt auf Collections aus Daten, wie z.B. Listen aus Objekten die sich bereits im Hauptspeicher befinden, oder fungiert als Bindeglied zu SQL um Daten aus einem Persistenten Medium wie der Festplatte eines Rechners (Sekundär Speicher, HDD/SSD) auszulesen. Hierzu zählen z.B. XML Datensätzen bzw. Dokumente. Das System ist ebenso dazu in der Lage mit Filesystemen und anderen Datenquellen zu interagieren. Das bedeutet, dass LINQ sich nicht grundsätzlich auf die vorher genannten Möglichkeiten beschränkt sondern offen ist für viele Arten von Datenstrukturen. Weitere Provider sind LINQ to DataSet und LINQ to SharePoint auf die hier mangels Relevanz zum Projekt nicht weiter eingegangen wird.

Noch einmal zusammengefasst zeichnen sich als wichtigste Einsatzgebiete für LINQ also die folgenden Methoden aus. Im .NET Namespace finden sich die LINQ Methoden unter System.Linq

- LINQ to Objects ist die Basis für LINQ Abfragen in .NET. Dieser Provider kann auf alle internen .NET Datenstrukturen oder Benutzerdefinierten Datenstrukturen ausgeführt werden solange die Schnittstelle IEnumerable oder IEnumerable<T> für generische Datenstrukturen implementiert wird.

- LINQ to SQL ist dazu geeignet auf SQL Datenbanken, besonders auf den Versionen von Microsoft zu arbeiten.

- LINQ to XML ist ein Provider der für eine Zusammenarbeit mit XML genutzt wird. Es handelt sich hierbei um eine XML Programmierschnitt-

stelle. [Mic13] [10]

Diese Arbeit beschäftigt sich also hauptsächlich mit dem LINQ Provider (LINQ Anbieter) LINQ to Objects, da sich die zu verarbeitenden Daten während der Bearbeitungszeit mit LINQ bereits im Speicher des Rechners befinden. Wie erwähnt, ist LINQ ein Layer für die Programmiersprachen von .NET und kann deswegen nicht nur in C# verwendet werden wie hier, sondern auch in Microsofts VB.NET. Dieses Projekt beschränkt sich auf C# als Programmiersprache, weswegen auch alle folgenden Codebeispiele in C# geschrieben sind.

2.3.1 Was ist LINQ to Objects genau?

LINQ to Objects wie in diesem Projekt verwendet, erfordert zur Benutzung erst einmal die grundsätzliche Eigenschaft des Datensatzes eine Collection zu sein. Jede Collection aus Objekten, auf welche ein LINQ to objects angewendet wird, muss also die Schnittstelle IEnumerable<T> implementieren. Diese Objekte werden im LINQ Vokabular dann als Sequences bezeichnet. In dieser Arbeit werden fast aussschliesslich die generischen Collections von C# verwendet welche grundsätzlich auch alle das generische IEnumerable<T> Interface implementieren. Der am meisten verwendete Datentyp in dieser Arbeit ist die generischen Implementierung von List<T>.
Eine LINQ to object Abfrage besteht immer aus drei Abfrageoperationen

- Bereitstellen einer Datenquelle
- Erstellen einer LINQ Abfrage
- Ausführen der LINQ Abfrage auf der Datenquelle mithilfe einer foreach() Anweisung

Das besondere an der Formulierung von LINQ Statements ist, dass sie zum Zeitpunkt ihrer Erstellung noch keine Daten der Datenquelle abfragen, sondern erst zum Ausführungszeitpunkt des Statements damit beginnen. Das ist sehr praktisch, denn dadurch können einmal in einer Abfrage erstellte Statements wiederverwertet werden. Eine LINQ Abfrage Variable speichert nie das Ergebnis einer Abfrage. Durch diesen Umstand ist es möglich, einen Datensatz zu unterschiedlichen Zeitpunkten abzufragen. Eventuell ändert sich auch das Ergebnis der Abfrage, wenn man davon ausgeht dass sich die Collection der Daten über die Zeit verändert. Trotzdem wird immer noch die semantisch gleiche Abfrage verwendet. LINQ Abfragen werden also mit einer Verzögerung schlussendlich in einer foreach() Anweisung ausgeführt. Dabei durchläuft die foreach() Anweisung Syntaktisch die Query oder Statement

[10] `http://msdn.microsoft.com/de-de/library/vstudio/bb387098.aspx` (geprüft 01.08.2013)

Variable und nicht die Collection der Daten.

Hier folgt ein Beispiel zu einer einfachen LINQ Abfrage auf einem einfachen Array Datensatz, welcher die Schnitstelle IEnumerable<T> implementiert.

Quellcode Beispiele 2.1: LINQeasyQuery.cs - Einfaches LINQ Abfrage Beispiel

```
// Collection to work with
int[] numbers = new int[5] {0, 1, 2, 3, 4};

// This is the statement
var numQuery =
    from num in numbers
    where (num % 2) == 0
    select num;

// Here the query gets executed.
foreach (int num in numQuery)
{
    Console.Write(num);
}
```

Ein direkter Aufruf der Methoden toList<T> oder toArray<T> auf dem Statement erzwingt die sofortige Ausführung der Abfrage.

Quellcode Beispiele 2.2: LINQdirectQueryList.cs - Einfaches direktes LINQ Abfrage Beispiel [Mic13]

```
List<int> numQuery2 =
    (from num in numbers
     where (num % 2) == 0
     select num).ToList();
```

2.3.2 Abfragesyntax in LINQ

Was im vorigen Code Beispiel 2.2 als LINQ Statement bezeichnet wurde ist eigentlich ein ganz spezielles LINQ Statemtent in einer speziellen syntaktischen Schreibweise. In diesem Fall ein LINQ Statement geschrieben in der Abfragesyntax. Diese Syntax zeichnet sich durch einfache Lesbarkeit und einfache Zugänglichkeit aus. Zur Compile Zeit wird diese Syntax vom Compiler in die so genannte Methodensyntax übersetzt. Die beiden syntaktischen Varianten sind semantisch identisch auch wenn sie sich auf den ersten Blick sehr unterscheiden.

2.3.3 Methodensyntax in LINQ

Die Methodensyntax in LINQ unterscheidet sich von der Abfragesyntax dadurch, dass anstatt fester Keywords wie where, from und select um eine Abfrage zu formen, Lambda Ausdrücke verwendet werden (Lambda Ausdrücke

werden im nächsten Abschnitt näher erleutert). In der Referenzdokumentation zu LINQ (Stand Juli 2013) [Mic13] wird hauptsächlich diese Art der Syntax verwendet. Aus den dort aufgeführten Typen wird in dieser Arbeit sehr häufig der Enumerable Typ verwendet.
Hier ein Vergleich der beiden Syntaktischen Möglichkeiten zum Erstellen von LINQ Statements.

Quellcode Beispiele 2.3: LINQzweiSynt.cs - Zwei syntaktische Möglichkeiten [Mic13]

```
//Query syntax:
IEnumerable<int> numQuery1 =
    from num in numbers
    where num % 2 == 0
    orderby num
        select num;

//Method syntax:
IEnumerable<int> numQuery2 = numbers.Where(num => num % 2 == 0).OrderBy(n => n
    );
```

Beide Methoden LINQ Abfragen in C# zu schreiben werden vom JIT (Just in Time Compiler) in gleichen Code übersetzt. Sie unterscheiden sich nur selten. Manche Standardoperatoren wie Sum oder Count sind in der Abfragesyntax nicht gegeben. Dieses Projekt wird in Zukunft alle LINQ Abfragen in der Methodensyntax verwenden um den Code zu vereinheitlichen. Der Entwickler kann sich also eigentlich aussuchen, in welcher Syntax er seinen Code am liebsten schreibt. Eine gute übersicht von LINQ Standard Operatoren wurde von Don Box und Anders Hejlsberg 2007 in

> „LINQ: .NET Language-Integrated Query“ [BH07, Standard Query Operators in a Nutshell]

beschrieben. [11]

[11] Zu finden unter `http://msdn.microsoft.com/en-us/library/bb308959.aspx`

Hier sollen nur die für dieses Projekt wichtigsten Standardoperatoren aufgelistet werden.

Operator	Beschreibung
Select	Ein Projektionsoperator der auf einer Auswahlfunktion basiert.
Where	Ein Restriktionsparemeter, seine Entscheidungen basieren auf einem angegebenen Prädikatausdruck
AsEnumerable	Konvertiert eine Collection in ein Objekt vom generischen Typ IEnumerable<T>
ToArray, ToList	Konvertiert eine Collection in ein Array oder eine Liste
Contains	Prüft ob ein Element in der Collection vorhanden ist
Distinct	Entfernt Duplikate aus einer Collection

2.4 Einführung zu Lambda Ausdrücken in C#

Lambda Ausdrücke in C# sind Anonymen Funktionen sehr ähnlich. In dieser Arbeit wurden Lambda Ausdrücke in LINQ Statements im Zusammenhang mit Standardabfrageoperatoren verwendet. Syntaktisch ist diese Art Funktionen zu schreiben sehr simpel gestrickt. Das Lambda Zeichen in C# wird als => dargestellt. Es folgt immer der Parameterliste und sollte nicht mit den Vergleichsoperatoren >= und <= verwechselt werden. Das Lamba Zeichen bedeutet hier soviel wie „wechselt zu“ oder „wird zu“ und steht links des gewünschten Ausdrucks.

2.4.1 Lambda Ausdrücke in Verbindung mit LINQ

Wie bereits beschrieben werden Lambda Ausdrücke in Verbindung mit LINQ in der Methodensyntax von LINQ verwendet. Diese Syntax wird meist benutzt um Standardabfrageoperatoren wie Collection.Select(lambda hier) Collection.Where(lambda hier) und Collection.All(lambda hier) zu verwenden. In dieser Arbeit wird der Standardoperator Select() am häufigsten verwendet. Select() wendet den angegebenen Lambda Ausdruck auf jedes Element der Collection an auf die der Select operator angewendet wird und gibt, falls vorhanden, ein Ergebnis zurück. Siehe Listing 2.4
Lambda ist also eine einfache Möglichkeit Standard LINQ Abfragen aufzuwerten. Es reicht im Hinblick auf dieses Projekt meist zur Abfrage von Informationen aus. Oft kommt es vor, dass mehrere LINQ Abfragen mit Lambda Statements kombiniert werden um ein Ergebnisset zu erhalten.

Quellcode Beispiele 2.4: LambdaEasy.cs - Einfacher Lambda Ausdruck

```
// Creating a collection.
int[] numbers = new int[5] {0, 1, 2, 3, 4};

// Executing a LINQ Statement in method form with a lambda expression.
// This expression will return every number in numbers multiplied by itself.
int[] newNumbers = numbers.Select(x => x * x);
```

2.5 Das Software Projekt LINQ For Geometry

Das Softwareprojekt LINQ For Geometry (kurz LFG), welches im Rahmen dieser Thesis Arbeit entworfen und implementiert wurde, stellt eine Umsetzung der Half-Edge Datenstruktur zum Handling dreidimensionaler Meshes dar. LINQ For Geometry hat darüber hinaus den Anspruch, die Datenstruktur durch LINQ Ausdrücke und bereit gestellte Iteratoren/Enumeratoren, welche mit LINQ erweitert werden können, flexibel und zugänglich zu gestalten. Es

bietet einen Converter in das von der Fusee Engine benutze Mesh Format und kann in Echtzeit geometrische Operationen auf einem in der Datenstruktur gespeichertem Mesh ausführen. LINQ For Geometry wurde Anfangs als eigenständiges Projekt begonnen. Im späteren Verlauf wurde es in die Fusee Engine integriert um dort wahlweise Meshes in der Half-Edge Datenstruktur im Speicher bereit zu halten. Das LINQ For Geometry Projekt ist von Fusee nicht abhängig. es wäre also nach Wunsch auf andere Engines oder Programme mit wenigen Anpassungen portabel.
LINQ For Geometry erweitert die Half-Edge Datenstruktur um ein paar Funktionen um die korrekte Darstellung von Meshes in Fusee zu gewährleisten. Dazu zählen Erweiterungen der an den Kanten und Faces gespeicherten Informationen, um UV Textur Koordinaten und Face- und Vertex Normalen im Mesh zu speichern. Zusätzlich ist es in der Lage, Kanten zu verarbeiten, die an mehr als 2 Polygone grenzen. Diese Lösung ist allerdings zum Zeitpunkt Juni 2013 nur als Notlösung zu betrachten und das massive Edge Sharing (Teilen von Kanten mit vielen Faces) sollte wenn möglich bereits beim Modellieren eines Meshes vermieden werden.

Abbildung 2.2: Diese Illustration zeigt die verschiedenen Layer von LFG und deren Verbindungen.

In den nächsten Abschnitten steht die Implementierung, siehe 4.1, von LINQ For Geometry im Vordergrund und versucht ein Umfassendes Bild zu vermitteln wie die Half-Edge Datenstruktur mit Hilfe von LINQ und Lambda Support in C# aufgebaut werden kann.

2.5.1 Konzept zu LINQ For Geometry

Das LINQ For Geometry Projekt begann als Thesis Projekt, in welchem angedacht war die Half-Edge Datenstruktur in C# zu implementieren und ihre Möglichkeiten in Verbindung mit LINQ ausfindig zu machen. Nach einer Weile zeigte sich jedoch, dass es wesentlich interessanter wäre, dieses Projekt in einer echten 3D Echtzeit Engine einzusetzten und die Möglichkeiten direkt in der Engine auszuwerten. Das Projekt wurde dann im nächsten Schritt in einem extra Branch in die Fusee Engine intergiert. Der Branch in welchem sich LINQ For Geometry aktuell (25. Juli 2013) befindet ist stehts auf dem aktuellen Stand des Fusee development Branches (develop). Alle Veränderungen des Fusee Projekts wurden während dieser Arbeit sofort eingepflegt um möglichen späteren Problemen entgegen zu wirken. Diese Arbeit ist von Fusee trotz der engen Bindung als Geometry Datenstruktur fast nahtlos abgekoppelt. Die einzigen Referenzen welche LINQ For Geometry auf Fusee enthält beziehen sich auf die Mathematik Bibliothek von Fusee und das von Fusee genutze Mesh Format. LINQ For Geometry kann dadurch recht einfach von Fusee gelöst und in eine andere Engine integriert werden. Lediglich die toMesh() Methode von LFG müsste dann auf eine neue Engine angepasst werden, falls sich deren Mesh Repräsentation von der bereits in Fusee implementierten Variante unterscheidet.

LINQ For Geometry stellt wie alle interessanten Anwendungsmöglichkeiten in Fusee ein eigenes Beispiel Projekt im Projektmappenordner „Examples“ bereit. In diesem Beispiel dreht sich alles um die Verwendung von LFG als Handler für die Mesh Daten und dessen Möglichkeiten. Verschiedene Modelle können als Mesh in eine Fusee Applikation geladen werden und durch verschiedene Tastenbelegungen welche in der Methode PullUserInput() hinterlegt sind auch transformiert werden. Aktuell bietet das Beispiel Projekt an, ein Mesh Objekt zu skalieren, es zu drehen, es zu verschieben und die Berechnung der Vertex Normalen aus zu schalten. Alle Berechnungen für die Transformationen laufen dabei direkt auf der Half-Edge Datenstruktur in LINQ For Geometry ab und werden nicht durch die Fusee Engine beeinflusst. Für die später noch angesprochenen Benchmarks, verfügt das Projekt über eine Demo Funktion welche das Mesh in vorher festgelegten Zeitabständen durch die bereitgestellten Transformationen bearbeitet. Durch den frei konfigurierbaren Ablauf und die Settings der Demo biett sie eine gute Basis für Benchmarks.

In Zukunft ist es angedacht, das Projekt LINQ For Geometry möglicherwei-

se als eine 3D Editor Datenstruktur in Fusee zu verwenden. Es ist ebenfalls denkbar, das Projekt einzusetzen wenn sich Meshes in einer Applikation zur Laufzeit dynamisch verändern sollen. Durch die massive Vernetzung der einzelnen Elemente (Half-Edges, Vertices, Faces) der Half-Edge Datenstruktur ist das Projekt sehr flexibel einsetzbar. Das VisualStudio Projekt „LingForGeometry.ExternalModules“ welches mit dem LINQ For Geometry Branch von Fusee mitgeliefert wird, ist ein Projekt um die Funktionen dieser Arbeit zu erweitern ohne den Core Code verändern zu müssen. Das Projekt hat Zugriff auf alle wichtigen Iteratoren/Enumeratoren (LINQ und nicht LINQ basiert) und Collections aus Daten. Um ein Beispiel für diese Funktionen zu haben wurden alle Geometrischen Operationen und Transformationen als externe Funktionen im ExternalModules Projekt implementiert.

2.5.2 Furtwangen University Simulation and Entertainment Engine

Die Furtwangen University Simulation and Entertainment Engine (kurz Fusee) ist eine Open Source 3D Echtzeit Engine mit einem Schwerpunkt auf einer Multiplattform Unterstützung. Entwickelt wurde sie von einer Projektgruppe an der Hochschule Furtwangen unter der Leitung von Herrn Prof. Müller an der Fakultät Digitale Medien.Von der aktuellen Version 0.5 werden folgende Plattformen unterstützt.

- Windows OS
- Linux OS
- Android OS
- Die Android Konsole OUYA
- Der Chrome Browser
- Der Firefox Browser

Um eine Linux und Android Unterstützung zu erreichen ist das Projekt mit Mono, einer .NET Open Source Implementierung, kompatibel. Sogar einen Export nach JavaScript durch den JSIL Compiler von Kevin Gadd ist möglich. Somit ist Fusee ohne weitere Plugins im Browser lauffähig.

Hier eine Liste der bereits umgesetzten Features in der Version 0.5:

- Multitexturing
- Ein standard Set an Shadern ist verfügbar
- Benutzerdefinierte Shader sind möglich
- Ein Scene Management wird unterstützt
- Sound in Browsern und Betriebssystemen vorhanden

LINQ For Geometry ist als Projekt in einen eigenen Fusee Branch integiert. Nach aktuellem Stand (Juli 2013) kann man sich als Benutzer entscheiden, ob man dann ein Modell mit dem in Fusee integrierten Mesh System oder mit dem Half-Edge Datenstruktur Geometry System von LINQ For Geometry laden möchte.
Als Schnittstelle zur GPU (Graphics Processing Unit) verwendet Fusee die low level Bibliothek OpenTK[12] welche in C# geschrieben ist und einen wrapper für die OpenGL Multi platform application programming interface (API) darstellt. Weitere Informationen über die Fusee Engine sind auf der offiziellen Homepage[13] oder über GitHub[14] erhltlich.[15]

2.6 Die Vorteile in der Entwicklung mit managed Programmiersprachen

Durch die enge Verknüpfung von LINQ For Geometry mit der Fusee Engine war es sinnvoll, das Projekt in der gleichen Programmiersprache wie Fusee zu implementieren, um eine Kommunikation der beiden Projekte möglichst einfach zu gestalten. Obwohl die .NET Basis hier natürlich Spielraum für Entscheidungen zugunsten VB.NET gelassen hätte. Sinnvoll wäre das natürlich nicht gewesen. Hier soll jetzt kurz geklärt werden was „Managed Code“ genau bietet und warum LINQ For Geometry in einer solchen Sprache wie C# implementiert wurde. Natürlich einmal davon abgesehen, dass LINQ For Geometry mithilfe von LINQ implementiert wurde und LINQ eine der Säulen des Projekts darstellt.
Managed Code bietet einem Entwickler viele verschiedene Vorteile im Gegensatz zur Entwicklung in einer Nativen Programmiersprache. Im Hinblick auf die Half-Edge Datenstruktur sollte besonders das Speichermanagement und die Pointer Verwaltung betrachtet werden. Durch die in C# implementierte

[12] http://www.opentk.com/(gepr"uft30.07.2013)
[13] http://fusee3d.org/
[14] https://github.com/FUSEEProjectTeam/Fusee
[15] Quelle http://fusee3d.org/about/ (Stand 19.07.2013)

Garbage Collection ist der Programmierer von der Aufgabe befreit, das Speichermanagement selbst zu verwalten. Das bietet einen großen Vorteil in der schnellen Entwicklung von Programmiersprachen. Managed Code ist für viele Entwickler einfacher zu schreiben als native Code in Sprachen wie z.B. C++. Im Hinblick auf LINQ For Geometry fällt durch die Nutzung von C# einiges an Arbeit weg. Nicht mehr gebrauchte temporäre Listen, von denen es einige im Projekt gibt und Objekte werden automatisch durch die Garbage Collection aufgeräumt und es kommt sehr selten bis kaum zu Memory Leaks durch Programmier- oder Laufzeitfehler. Es ist so wesentlich einfacher externen Nutzern einen Zugang auf den Code des Projekts zu ermöglichen ohne dass sie Gefahr laufen, die Integrität der Datenstruktur durch Pointer Manipulationen zu gefährden. C# bietet hier also ein gewisses Maß an Sicherheit für das System.
Ein weiterer Pluspunkt ist natürlich, dass sich bereits viele Entwickler und Studenten der Hochschule Furtwangen, die später möglicherweise die Fusee Engine verwenden möchten, bereits mit C# oder Java, welches C# zumindest syntaktisch sehr ähnelt, auskennen.

2.6.1 Warum die Programmiersprache C# für LINQ For Geometry?

Wie bereits im oberen Abschnitt 2.5.2 angeschnitten, wurde LINQ For Geometry für eine Zusammenarbeit mit der Fusee Engine konzipiert. LINQ For Geometry bietet hierbei eine neue Geometry Objekt Klasse für die Fusee Engine an, ohne die tatsächliche Klasse zu überschreiben. Laut aktuellem Stand kann der Benutzer beider Projekte sich also entscheiden, welche Meshes er gerne in der Half-Edge Datenstruktur laden möchte und für welche es genügt, sie als Face basiertes Mesh im Speicher zu halten. Diese Entscheidung lässt dynamische Entscheidungen zu. Es ist somit im Bereich des Möglichen, manche Meshes als unveränderbares Fusee.Mesh zu behandeln und andere während der Laufzeit durch LFG zu manipulieren.

Damit LINQ For Geometry auf diese Art mit der Fusee Engine kommunizieren kann, musste eine gemeinsame Schnittstelle evaluiert werden. Weil beide Projekte die .NET Plattform benutzen und Fusee in C# geschrieben wurde war es naheliegend für LINQ For Geometry auch C# zu benutzen. Da der LINQ Layer ein Produkt von .NET ist, war C# die wohl beste Wahl um beide Projekte zusammen zu bringen.

2.6.2 Der Link von LINQ zur Half-Edge Datenstruktur

LINQ For Geometry ist durch die Bereitstellung einiger grundlegender Enumeratoren einfach zu erweitern. Um den Einsatz des Projekts für externe Benutzer weiter zu vereinfachen, wurde das Hauptaugenmerk darauf gelegt, LINQ für Datenabfragen auf der fertigen Half-Edge Datenstruktur zu verwenden. LINQ ist einfach verständlich und kann durch wenige Zeilen Code aufwendige traversierungs Algorithmen realisieren. Die bereits angebotenen Enumeratoren sollen im zukünftigen Einsatz von LFG unbedingt wiederverwendet werden und sind teilweise schon selbst nur durch Kombinationen aus anderen in LFG implementierten LINQ Abfragen aufgebaut. So ist es ein Ziel dieser Arbeit LINQ als Bindegleid und „Schnittstelle“ zum Projekt einzusetzen und damit die Produktivität beim Formulieren neuer Algorithmen und Funktionen zu steigern.

2.7 Der Import von Geometriedaten im „Wavefront Object“ Format

Das Projekt LINQ For Geometry benutzt das von Wavefront Technologies entwickelte offene „Wavefront Object“ Format (Dateiendung .obj) um Modelle aus 3D Modellierungssoftware zu importieren. Wavefront Technologies existiert heute nicht länger. Die Firma wurde im Jahr 1995 von Silicon Graphics augekauft und mit Alias Research zu Alias|Wavefront verschmolzen. Mittlerweile gehören beide Firmen zu Autodesk (3DS Max, AutoCAD).
Das .obj Format zeichnet sich durch seine einfache Form und durch die von Menschen lesbare Kodierung in ASCII aus. Bei diesem Format handelt es sich um ein Face basiertes Format zur Speicherung von Mesh Daten. Alle Vertices eines Meshes werden gespeichert und Faces definieren ihre Eckpunkte aus den Indizes der Vertices. Es ist ebenfalls möglich, Texturkoordinaten und Normalen zu speichern. Die optionalen Tags „mtllib“ und „mtl“ geben an ob Material Dateien vorliegen und wann und wo diese verwendet werden sollen. Diese Funktion wird in LFG und Fusee zum Zeitpunkt dieser Arbeit nicht unterstützt. Texturen werden aktuell noch als .jpg Dateien geladen. Kommentarzeilen werden mit einer Raute „#“ gekennzeichnet, Objekte und Gruppen von Objekten mit einem „o“ und „g“. Das Wavefront .obj Format ist heute eine Art Standard. Die meisten 3D Programme können es importieren und exportieren. Dazu zählen das während dieser Arbeit verwendete Cinema 4D von Maxon, sowie die Open Source Software Blender und die Fusee Engine.

2.7.1 Warum das Wavefront Object Format

Warum verwendet LFG das Wavefront .obj Format? Einer der Wichtigsten Gründe ist sicherlich, dass Fusee in seiner Standard Implementierung ebenfalls dieses Format verwendet. Um die Übersichtlichkeit des Projektes zu erhalten und den Prozess des Imports nicht zu verkomplizieren wurde während dieser Arbeit ebenfalls entschieden, den Import der Daten im .obj Format vorzunehmen. Der Importer von LINQ For Geometry ist jedoch ein Anderer als der Importer von Fusee. Hauptsächlich aufgrund einer fehlenden Schnittstelle und anderer Import Algorithmen. Das hat den Vorteil, dass ein späterer Benutzer des Systems den Importer erweitern kann, ohne den Engine Code oder den Code von LINQ For Geometry ändern zu müssen. Der Importer ist vom LFG Projekt abgekoppelt. Ein Nutzer könnte also sogar seinen eigenen Importer für ein anderes Format verwenden, solange er der Geometry Klasse das korrekte Datenformat inklusive der benötigten Datentypen zum Aufbau der Datenstruktur übergibt. Im LFG erwartet die Geometry Klasse vom Importer die Übergabe eine Collection (einfach verkettete Liste, in C# List<T>) aus „GeoFaces“.

Quellcode Beispiele 2.5: GeoFace.cs - Ein temporärer Face basierter Datentyp.

```
1    public struct GeoFace
2    {
3        internal List<float3> _LFVertices;
4        internal List<float2> _UV;
5    }
```

Ein „GeoFace“ ist ein Konstrukt, das ein Polygon (In LFG Face genannt) eines 3D Models (Meshes) repräsentiert. Jedes GeoFace selbst enthält eine Liste aus Fusee.Math float3 Vertice Daten und Fusee.Math float2 UV Koordinaten.

Im Beispiel 2.7.1 handelt es sich um einen Würfel, der ohne Normalen oder UV Textur Koordinaten gespeichert wurde. Der Würfel wurde aus der Software Cinema4D exportiert. Der Würfel besteht aus Quads. Das ist daran zu erkennen, dass jedes Face 4 Vertices referenziert.

```
1  # WaveFront *.obj file (generated by CINEMA 4D)
2
3  g Cube
4  usemtl Mat
5  v -100 -100 100
6  v -100 100 100
7  v 100 -100 100
8  v 100 100 100
9  v 100 -100 -100
10 v 100 100 -100
```

```
v -100 -100 -100
v -100 100 -100

f 3 4 2 1
f 5 6 4 3
f 7 8 6 5
f 1 2 8 7
f 4 6 8 2
f 5 3 1 7
```

2.7.2 Der Importer für das Wavefront Format oder auch Face basierter Import - Edge basiertes Handling

Zum Import Zeitpunkt eines Assets (Meshes) wird zuerst der Inhalt der im Speicher des Betriebssystems abgelegten .obj Datei komplett eingelesen. Die Datei Endung für eine Mesh Datei im Fusee Universum unterscheidet sich durch die zusätzliche Endung .model leicht von der Standard Datei Endung. So wird ein Konflikt mit den in Visual Studio benutzen Compilern und dem .obj Format für kompilierte aber noch nicht gelinkte Dateien vermieden. Eine Datei würde dann für die Verwendung in Fusee und LFG als Cube.obj.model benannt werden. Der Importer parsed jetzt alle nötigen Strings in Integer Werte und baut aus den Face Definitionen im .obj Format „GeoFace“ Objekte zusammen. Zu diesem Zeitpunkt liegt das komplette Mesh noch Face basiert vor. Es gibt noch keine Beziehungen zwischen den einzelnen Elementen und jedes Face steht für sich alleine. Die ganze Collection an Daten wird nun von der Geometry abgeholt und durch den Initialisierungslauf in die Half-Edge Datenstruktur übersetzt. Erst entsteht aus dem Face basierten Meshe die Edge basierte Datenstruktur.

2.8 Initialisierungsablauf und Aufbau der Half-Edge Datenstruktur in LINQ For Geometry

In diesem Abschnitt wird kurz aufgezeigt wie aus einem Face Basierten Mesh aus GeoFaces die Half-Edge Datenstruktur entsteht.
Die Geometry Klasse in LINQ For Geometry übernimmt die Speicherung der Datenstruktur ebenso wie die Umsetzung eines Meshes in diese. Ein Asset (Mesh) muss mit der Methode LoadAsset(String path) geladen werden. Die Load Asset Methode übernimmt den kompletten Konvertierungsprozess in die Half-Edge Datenstruktur . Eine Liste aus GeoFaces wird durch den Importer geladen und dieser Methode anschliessend zur Verfügung gestellt. Weiterhin wird nun für jedes Face in der bekannten Liste durch die Methode AddFace(GeoFace face) ein Face in der Datenstruktur eingefügt. Der genaue

Ablauf hierzu wird in UML Diagrammen dargestellt.
LINQ wird im Initialisierungslauf dafür eingesetzt, zu überprüfen, ob ein Vertice schon durch das Einfügen eines Faces in der Datenstruktur vorhanden ist, oder ob der Vertex noch eingefügt werden muss. Durch diese Abfrage wird verhindert, dass sich redundante Datensätze bilden. Eine ähnliche aber weitaus komplexere LINQ Abfrage erfolgt beim Einfügen der Kanten. Auch hier wird auf bereits bestehende Kanten geprüft. Das ist immer dann der Fall, wenn zwei Faces sich zwei Vertices teilen und das zweite Face nach dem ersten in die Datenstruktur eingefügt werden soll. Das System erkennt hier eine bereits bestehende Kante und anstatt eine neue Kante zu erstellen wird nur die freie Half-Edge der Kante verwendet.

Hier exemplarisch der verkettete LINQ Query zum Überprüfen der vorhandenen Kanten. Er funktioniert in diesem Fall ähnlich einer if Abfrage und sucht per FindIndex() den Index des ersten ,Elements welches auf das angegebene Prädikat passt. Sollte kein passendes Element gefunden werden, so wird -1 zurück gegeben.

```
_LedgePtrCont.FindIndex(
                    edgePtrCont => _LhedgePtrCont[edgePtrCont._he1]._v ==
    fromVert && _LhedgePtrCont[edgePtrCont._he2]._v == toVert ||
    _LhedgePtrCont[edgePtrCont._he1]._v == toVert && _LhedgePtrCont[
    edgePtrCont._he2]._v == fromVert)
                );
```

An dieser Stelle im Code wurde für LFG eine kleine Erweiterung implementiert. Sollte eine Kante bereits von zwei Faces belegt worden sein, jedoch eine dritte Kante ebenfalls eine Kante zwischen den zwei bereits belegten Vertices fordern, so wird eine neue Kante für das dritte Face eingerichtet. Dieser Vorgang ist nicht begrenzt. Der 3D Artist sollte jedoch darauf achten, dass keine Kanten im Mesh von mehr als 2 Faces benutzt werden. Dieses Fallback wurde eingebaut um, einen Crash der App im Falle eines Imports eines fehlerhaften Meshes abzufangen. Die Datenstruktur ist auf eine solche Handhabung in LFG eigentlich nicht ausgelegt.
In der AddFace() Methode werden ebenfalls die benötigten UV Koordinaten an den Half-Edges gespeichert. Sie werden an den Half-Edges gespeichert welche mit ihrem Vertice Handle auf die korrekten Vertices zeigen. Zuletzt werden alle Half-Edges des Faces noch miteinander verbunden. Diese Operation wird zuletzt ausgeführt, weil der Index für das Handle einer Half-Edge nicht im Vorfeld berechnet werden kann. So können in einem Modell das Face Nr 1 und das Face Nr XXX nebeneinander liegen und sich möglicherweise manche

Kanten teilen.
Ein Face muss jetzt nur noch als Face Pointer Container in das System eingefügt werden und LFG beginnt mit dem nächsten Face von vorne bis alle Faces abgearbeitet wurden.

2.9 Implementierung und Funktion der Handler für die einzelnen Komponenten der HES

Die Handler Konstrukte „Structs“ in dieser Arbeit, welchen Datentyps auch immer, können als Zeiger auf Datensätze von realen Daten betrachtet werden. Sie dienen als Indizes um Daten zu vernetzen und bieten eine Kontrollmethode, um zu Überprürfen ob ein Handler „valid“ (Zulässig, In Ordnung) ist. Ein Handler auf einen Satz von Daten kann wenn gewünscht mehrmals verwendet werden. Dies würde bei geeigneter Implementierung die Menge der zu speichernden Daten um den Faktor n (n = wiederholte Nutzung des Handlers) reduzieren.

2.9.1 Beispiel eines Handler Konstruktes und seiner Implementierung

Sie enthalten nur einen Index als Zeiger und sehr wenige Funktionen. Ein Handler speichert zur Laufzeit pro Instanz einen Index auf den realen Datensatz für den er einen Handle darstellt. Handler Structs gibt es für Edges, Half-Edges, Faces, FaceNormals, Vertices, VertexNormals und VertexUVs. Sie unterscheiden sich hierbei nur durch die Namensgebung der Structs und der Konstrukturen.

Quellcode Beispiele 2.6: HandleHalf-Edge.cs - Variablen Deklaration des „Zeigers“

```
1        internal int _DataIndex;
```

Ein Handler Struct stellt eine implizite Konvertierung des Handlers, siehe Listing 2.7, in den Datentypen Integer (int) zur Verfügung.

Quellcode Beispiele 2.7: HandleHalf-Edge.cs - Impliziter cast nach Integer

```
1        public static implicit operator int(HandleHalfEdge handle)
2        {
3            return handle._DataIndex;
4        }
```

Zusätzlich kann der Entwickler jederzeit abfragen, ob der aktuell verwendete Handler schon als valide betrachtet werden kann. Hierzu Listing 2.8 betrach-

ten. Ein Handler ist dann valide, wenn sein Index nicht kleiner als 0 ist. In dieser Arbeit werden einstweilen Handler initialisiert, für die zum Zeitpunkt der Initialisierung noch kein Index zur Speicherung bereit steht. Diese vorerst nicht validen Handler werden dann mit dem Wert -1 initialisiert und sind somit zu diesem Zeitpunkt als nicht valide also nicht verwendbar zu betrachten. Um diese später im Programm zu benutzen, muss also noch der korrekte Index, meistens der Wert einer Count Funktion auf einer Liste eingefügt werden.

Quellcode Beispiele 2.8: HandleHalf-Edge.cs - Is Valid?

```
        public bool isValid
        {
            get { return _DataIndex >= 0; }
        }
```

Eine Besonderheit der Handler ist die mit „internal“ gekennzeichnete Deklaration der Indizes. Siehe hierzu Listing 2.9. Durch das „internal“ C# Schlüsselwort können die Handler nur aus der jeweiligen gleichen Assembly angesprochen und verändert werden. Dies verhindert einen unbefugten oder unabsichtlichen Fremdzugriff von Außen. Während der Laufzeit wird also die Konsistenz der Datenstruktur in sich geschützt, um so das Programm vor Abstürzen durch Zeiger Fehler zu schützen.

Quellcode Beispiele 2.9: HandleHalf-Edge.cs - Deklarationen als internal

```
        internal int _DataIndex;
```

„Der interne Zugriff wird häufig in komponentenbasierter Entwicklung verwendet, da er einer Gruppe von Komponenten ermöglicht, in einer nicht öffentlichen Weise zusammenzuwirken, ohne dem Rest des Anwendungscodes zugänglich zu sein.“ [Mic13]

Die vollständige Implementierung dieses Structs und aller sieben weiteren kann im Visual Studio 2010 Softwareprojekt unter folgender Verzeichnisstruktur betrachtet werden. „LinqForGeometry/LinqForGeometry.Core/src/Handles“

2.10 Pointer Container und ihre Aufgabe in LINQ For Geometry

Pointer Container in LINQ For Geometry sind C# Structures und existieren für jeden in der Datenstruktur essentiellen Datentypen und gehören zum Kern Projekt (LinqForGeometry.Core) des LINQ For Geometry Projekts. Zu den Containern zählen Half-Edges Container, Edge Container, Face Container und Vertex Container. Ein Pointer Container wird für jeden Datensatz, der in die Datenstruktur eingefügt wird, initialisiert und in einer Liste gespeichert. Jeder Container enthält, den Spezifikationen der Half-Edge Datenstruktur folgend, Handles auf andere eingepflegte oder in kurzer Zeit einzupflegende Datensätze z.B. durch eine mathematische Operation (welche den voraussichtlichen Index eines einzufügenden Datensatzes berechnet). Pointer Container sind sozusagen die Gebilde, welche die Half-Edge Datenstruktur aufspannen und ihre Eigenschaften im Speicher vorhalten. Möchte man ein gespeichertes Mesh transformieren, so sind alle Operationen die vorgenommen werden Änderungen an den Pointer Containern bzw. den in ihnen enthaltenen Daten. Pointer Container sind also die Basis der ganzen Datenstruktur.

Durch die Gegebenheiten der Implementierung von List<T> (Generic Lists) in C# ist es leider nicht möglich, die Container bei Änderungen direkt in einer Liste zu manipulieren. Jeder Container muss erst aus der Liste kopiert werden um ihn dann zu ändern und anschliessen an der gleichen Stelle wieder einzufügen. Nötig hierzu ist, dass der Index des Containers in der jeweiligen Liste bekannt ist. Hierzu ein kleines Beispiel welches die Erklärung etwas erläutern soll.

Quellcode Beispiele 2.10: PtrContExample1.cs - Manipulation eines Pointer Containers

```
// This list is a list full of Half-Edge Pointer Containers.
List<HalfEdgePtrCont> _LhePointerContainers = new List<HalfEdgePtrCont>();

...
// We are building up a Half-Edge Structure and do interesting things ...
```

```
...

// Now we want to manipulate a specific Pointer Container.
// So we first need a Handle to a Pointer Container.
// We retrieve "hep" by selecting the third last element in the Pointer
    Container List.
// (Does not make any special sense it's just for the example)
HandleHalfEdge heH = new HandleHalfEdge(_LhePointerContainers.Count() - 3);

// Now we want to manipulate this data set.
_LhePtrCont[heH] = new HalfEdgePtrCont()
{
    ...
    _f = _LhePointerContainers[heH]._f,
    ...
};

// Note: This is much faster than to allocate a new HalfEdgePtrCont in Memory
    and then using RemoveAt() and Insert() on the List<t>.
```

Dieses Verfahren ist aber um einiges effizienter als die von C# bereit gestellten Insert() und RemoveAt() Operationen auf List<T>. Aus diesem Grund sollte es wann immer es nötig ist auch den Vorzug erhalten.

2.10.1 Half-Edges Pointer-Container

Der Half-Edge Pointer Container, in LINQ For Geometry class HEdgePtrCont, speichert alle Informationen die laut Half-Edge Datenstruktur an einer Half-Edge anliegen müssen. Die zugehörige Datei im LINQ For Geometry Projekt ist „HEdgePtrCont.cs“.

- Ein HandleHalfEdge _he; (auf die Twin Half-Edge)
- Ein HandleHalfEdge _nhe; (auf die nächste Half-Edge im Uhrzeigersinn)
- Ein HandleVertex _v; (auf den Vertex auf den die Half-Edge zeigt)
- Ein HandleFace _f; (auf das zugehörige Face)

In der Implementierung von LINQ For Geometry werden an einer Half-Edge noch zusätzliche Handler Informationen für die Verwendung und Konvertierung des Meshes in und nach Fusee gespeichert.

- HandleVertexNormal _vn; (auf die Vertex Normale eines Vertex)
- HandleVertexUV _vuv; (auf die UV Texture Koordinaten eines Vertex)

Da jede Half-Edge auch eine Verbindung zum Vertex hat auf den sie zeigt, ist es sehr praktisch, an ihr auch die zusätzlichen Informationen zu speichern, die der Vertex für die Fusee Integration benötigt. Es wird also ein Index auf die Vertex Normale im Speicher zum angesprochenen Vertex gespeichert. Zusätzlich noch

ein Index auf die UV Texture Koordinaten die für den Vertex gelten. So kann die Iteration bei einer späteren Konvertierung der Datenstruktur in ein für Fusee verständliches Mesh Objekt überhaupt erst möglich gemacht werden. Es wäre mit erheblichem Mehraufwand und wachsendem Speicherverbrauch auch möglich gewesen, die Informationen an den einzelnen Vertices durch Verknüpfungen mit Listen, welche jeweils die die Daten für einen Vertex speichern zu realisieren, allerdings hätte das eine regelrechte Aufweichung der Idee der Datenstruktur bedeutet (alle wichtigen Informationen werden an den Half-Edges gespeichert). Zusätzlich wäre es so wesentlich aufwendiger geworden, eine korrekte Kanten und Winkel basierte Vertex Normalen Berechnung pro Vertex pro Face für die in Fusee vorhandene Beleuchtungsshader zu implementieren wie sie durch diese kleine Erweiterung der Half-Edge Datenstruktur jetzt bereits in LINQ For Geometry vorhanden ist.

2.10.2 Edges Pointer-Container

Ein Edge Pointer Container, in LINQ For Geometry zu finden im Struct „EdgePtrcont.cs“, ist eine Repräsentation von Kanten in der Half-Edge Datenstruktur . Jeder Edge Pointer Container beinhaltet zwei Handles auf Half-Edges.

- HandleHalfEdge _he1; (Handle auf die Erste Half-Edge eines Half-Edge Paares)
- HandleHalfEdge _he2; (Handle auf die Zweite Half-Edge eines Half-Edge Paares)

Diese beiden Handles sind ein jeweils zusammengehöriges Half-Edge Paar. Durch die Bereitstellung dieser Edge Pointer Container ist zum Beispiel eine Selektierung von echten Kanten in einem 3D Editor denkbar. Sie dienen ebenfalls als Einstiegspunkt, um Manipulationen auf das gespeicherte Mesh anzuwenden.

2.10.3 Vertices Pointer-Container

Dadurch, dass in der Half-Edge Datenstruktur alle wichtigen Informationen bereits an jeder Half-Edge gespeichert werden, speichern Vertex Pointer Container lediglich einen Handle auf eine von ihm ausgehende Half-Edge. Da ein Vertex aber meistens mehrere ausgehende Half-Edges besitzt, wird der Platz nach dem „first come first serve“ Prinzip verteilt. Das bedeutet, dass nur die erste Half-Edge, die während des Initialisierungslaufs an den Vertex angelegt wird an dieser Stelle auch referenziert wird. Durch diesen Handle können Iteratoren sehr einfach realisiert werden, trotz dass sie in anderen Datenstrukturen

wie der Face basierten Datenstruktur einiges an Mehraufwand bedeuten würden. Als Beispiel soll hier der Sternumlauf Iterator genannt werden. Dieser kann alle eingehenden oder ausgehenden Half-Edges zu einem gegebenen Vertex bestimmen. Hier dargestellt kurz der Ablauf des Iterators für eingehende Half-Edges:

1. Es wird ein Vertex als Startpunkt erwartet.
2. Es wird die ausgehende Half-Edge des Vertex angesprochen (und den Index merkt sich der Algorithmus als Determinante).
3. Es wird die Twin Half-Edge der eben geholten Half-Edge angesprochen.
4. Die Twin Half-Edge wird als erste eingehende Half-Edge im Ergebnisset zwischengespeichert.
5. Zu dieser Half-Edge wird jetzt die „Next“ Half-Edge angesprochen.
6. Ihre Twin Half-Edge ist nun wieder eine eingehende Half-Edge.
7. Ab jetzt wird der Algorithmus ab Schritt 3 wiederholt bis man als Index einer nächsten Half-Edge die zuvor gespeicherte Determinante erreicht.

So ist es mit ein paar Pointer Sprüngen möglich, diesen Algorithmus schnell und effizient durchzuführen. In einer Face basierten Datenstruktur wäre eine solche Operation wegen der fehlenden Beziehungen und Vernetzungen ohne massive Änderungen der Strukur kaum in dieser Geschwindigkeit möglich.

2.10.4 Faces Pointer-Container

Laut Definition der Half-Edge Datenstruktur speichert ein Face jeweils einen Zeiger auf eine ihm angehörige Half-Edge. In LINQ For Geometry wurde genau das in der Implementierung umgesetzt. Das erleichtert den Einstieg in geometrische Operationen wenn der Startpunkt für diese als Face gegeben ist. So ist es einfach die Grenzen (Boundaries) eines Faces und z.B. die das Face umspannenenden Half-Edges, Edges, Vertices oder sogar die benachbarten Faces herauszufinden. LINQ For Geometry stellt in seiner Implementierung all diese Iteratoren zur Verfügung um damit weitere Algorithmen zu entwerfen.

- HandleHalfEdge _h;

Zusätzlich zu den Standards der Half-Edge Datenstruktur bietet LINQ For Geometry die Möglichkeit an jedem Face ein Handle auf eine dem Face zugehörige Face Normale zu speichern. Diese Face Normale gibt die Richtung des Faces an sollte die Datenstruktur in eine 3D Engine implementiert werden in

welcher Backface Culling unterstützt und dabei Normalen als Ausrichtungsmerkmal beachtet werden. In LINQ For Geometry wird die Face Normale aktuell nur zur Berechnung der Vertex Normalen herangezogen. Fusee benötigt für die Unterstützung von Backface Culling lediglich eine Traversierung der Polygone entgegen des Uhrzeigersinns. Diese Anforderung ergibt sich in Engines welche OpenGL als Grafikschnittstelle verwenden. Zu diesen gehört Fusee auf Grund der Verwendung der OpenTK Schnittstelle. OpenTK basiert auf OpenGL.

- HandleFaceNormal _fn;

2.11 Die Geometrie in LINQ For Geometry

Das Geometry Objekt ist der Kern des ganzen LINQ For Geometry Projekts. Von der Umwandlung eines mit dem Importer eingelesenen Meshes bis zur Speicherung während der Laufzeit managed das Geometry Objekt ein komplettes Mesh. Die gespeicherten Daten im Objekt werden in Collections vorgehalten um einen schnellen Zugriff durch Iteratoren zu gewährleisten. Es werden Listen anstelle von Arrays oder ähnlichem verwendet, um die Handhabung von Zugriffen auf die Daten zu erleichtern.
Eine Übersicht über die gespeicherten und zur Verfügung stehenden Collections inklusiver ihrer Zugriffsmodifizierer und Datentypen.

- Handles auf Pointer Container
  - public List<HandleVertex> _LverticeHndl;
  - public List<HandleEdge> _LedgeHndl;
  - public List<HandleFace> _LfaceHndl;
- Pointer Container
  - private List<VertexPtrCont> _LvertexPtrCont;
  - private List<HEdgePtrCont> _LhedgePtrCont;
  - private List<EdgePtrCont> _LedgePtrCont;
  - private List<FacePtrCont> _LfacePtrCont;
- Reale Daten
  - public List<float3> _LvertexVal;
  - public List<float3> _LfaceNormals;
  - public List<float3> _LVertexNormals;
  - private List<float2> _LuvCoordinates;
  - private List<float3> _LvertexValDefault;

Diese Listen enthalten alle wichtigen Informationen über das in der Half-Edge Datenstruktur vorhandene Mesh. Durch Manipulation der darin gespeicherten Daten kann das Mesh verändert werden. Alle von LINQ For Geometry implementierten Iteratoren und Geometrie Manipulatoren benutzen diese Listen.
Das Geometrie Objekt kann aus dem LINQ For Geometry.ExternalModul Projekt heraus angesprochen werden um weitere Manipulationsalgorithmen zu erstellen.

Quellcode Beispiele 2.11: ExternalModule.cs - Wie wird ein Geometry Objekt im externen Modul verwendet

```
/*
  The following method header would apply
  a transformation matrix to a geometry object.
*/
public static bool Transform(float4x4 transformationMatrix, ref Geometry geo)
    {...}
```

Das Geometry Object wird hier durch Call by Reference (dargestellt durch das Keyword „ref“ in C#) übergeben. Diese Methode spart aufgrund der Größe eines Geometry Objekts und der vielen darin enthaltenen Daten einiges an Zeit und Speicherplatz im Arbeitsspeicher (RAM).

2.12 Anwendungsszenarien von LINQ For Geometry

LINQ For Geometry kann mit der Half-Edge Datenstruktur als Basis für viele Anwendungesgebiete interessant sein. Aktuell ist das System als Ersatz für das in Fusee integrierte Mesh Konzept eingebettet. Hier lädt es Meshes aus Wavefront Object Dateien während der Laufzeit eines Fusee Programms in den Speicher und hält die Daten bereit. Das Mesh kann dann während der Laufzeit durch verschiedene Methoden manipuliert werden. Allerdings sind weitere Einsatzgebiete denkbar und teilweise sogar angedacht. Das System könnte es zum Beispiel möglich machen, Meshes zur Laufzeit eines Spiels zu verändern. Dabei wäre es denkbar, Konstruktion oder Destruktion in einer Echtzeit Spielwelt mit LFG zu realisieren. LFG ist ein sehr gut geeignetes System für alle Algorithmen welche eine kantenbasierte Präsentation der Daten erwarten. Dazu zählen z.B. verschiedene Subdivision Surfacing Algorithmen wie der „Loop“ und der „Catmull-Clark“ Algorithmus. Diese Algorithmen beschreiben eine Möglichkeit, Meshes in immer feinere Strukturen zu unterteilen um ihnen so eine gleichmässigere Oberfläche zu verleihen oder Details herauszuarbeiten oder aber diese zu verringern. Der interessanteste Fall für LINQ For Geometry ist sicherlich die möglicherweise zukünftige Verwendung des Projekts für einen 3D Editor auf Basis der Fusee Engine. Dabei wäre es dann möglich, Meshes in Echtzeit zu bearbeiten und sie in der Fusee Engine anzuzeigen und zu benutzen ganz wie es bereits in 3D Modellierungssoftware wie Cinema 4D aktuell der Fall ist. Die erstellten Meshes könnten dann abgespeichert oder direkt als Assets in einem Fusee Projekt verwendet werden. Ein Editor mit dieser Funktion könnte durch das LinqForGeometry.ExternalModules Projekt recht einfach um neue Funktionen erweitert werden und kann so an die Wünsche und Anforderungen der verschiedenen Nutzer angepasst werden.

2.13 LINQ Abfragen und ihr Verhalten bei der Selektierung großer Datenmengen in LINQ For Geometry

Trotz dass LINQ einen einfachen Einstieg in die Formulierung der Algorithmen ermöglicht, gibt es einige wichtige Gegebenheiten zu beachten. Die Geschwindigkeit von LINQ Abfragen skaliert sehr stark mit der größe der Collection aus welcher eine Abfrage ein Ergebnis filtern soll. LINQ hat bei sehr großen Collections wie der Half-Edge Collection eines hochpolygonalen Meshes im LFG Projekt eine Schwäche im Performance Sektor gezeigt. Um z.B. die Half-Edges eines Faces heraus zu finden, müsste der Algorithmus mit LINQ über das komplette Set der vorhandenen Half-Edges laufen. Bei einem Mesh mit m Polygonen beträgt die Anzahl der Half-Edges $n = m * c$ (c =Anzahl der Vertices pro Face). Und über diese Anzahl an Half-Edges müsste für jedes Face iteriert werden. Der aktuell implementierte Algorithmus welcher die Topologie der Datenstruktur ausnutzt benötigt nur c Pointer Sprünge um alle Kanten eines Faces zu finden und ist damit um ein Vielfaches schneller.

LINQ funktioniert allerdings sehr gut für Operationen auf kleinen Collections an Daten oder aber wenn sich die Größe des Ergebnis Sets nicht zu stark von der Größe der Quellen Collection unterscheidet. So werden LINQ Abfragen aktuell in den meisten Enumeratoren verwendet welche durch einen anderen Enumerator bereits ein gefiltertes Daten Set als Parameter empfangen. Eingehende Half-Edges eines Vertices werden aufgrund der großen Collection an Quell Daten durch einen Algorithmus auf der Datenstruktur bestimmt, aber ausgehende Half-Edges werden durch eine LINQ/Lambda Abfrage auf das Ergebnis des ersten Algorithmus abgefragt. Dieses Vorgehen hat sich während der Entwicklung des Projektes bewährt. So können „teure“ Algorithmen mit „nativer“ Implementierung wiederverwendet werden um günstigere, schnelle, neue LINQ Abfragen aus Kombinationen zu realisieren. Aktuell sind zwei der zehn Enumeratoren nativ realisiert während sich alle weiteren aus Kombinationen mit LINQ Abfragen zusammen setzen.

LINQ eignet sich also trotz der Limitierung auf kleinere Daten Sets sehr gut um Traversierungs und Manipulations Algorithmen auf der Half-Edge Datenstruktur zu formulieren, solange immer betrachtet wird wie sinnvoll eine Abfrage in der Relation des Quell Sets zur Größse des zu erwartenden Ergebnis Sets ist. Für diese Arbeit enstand hierdurch kein Nachteil. Diese Limitierung hat sogar dafür gesorgt, dass LINQ Statements kürzer formuliert werden konnten und Enumeratoren durch Kombination frei von redundantem Code geblieben sind.

2.14 Stern- und Umlaufenumeratoren (Iteratoren/Enumeratoren)

Iteratoren sind die wichtigste Basis an Funktionen, die LINQ For Geometry einem Nutzer des Projekts bereit stellt. Iteratoren sind hier als „Zeiger“ auf Elemente aus den Collections der Datenstruktur zu bezeichnen. Ein Iterator hat ein festgelegtes „Bewegungsmuster“ und bewegt sich über bestimmte definierte Abschnitte der Collections der Datenstruktur. Die Iteratoren in LINQ For Geometry sind jeweils auf einen Start Datentyp und auf einen Return type festgelegt. Durch LINQ Ausdrücke in der Methodensyntax oder durch die Realisierung von kleinen Traversierungsalgorithmen liefern diese Iteratoren ein Ergebnisset, meistens eine Liste, eines bereits vorher bekannten Datentyps an den Aufrufer zuück. Durch die geschickte Kombination dieser Basisiteratoren ist es möglich, viele Problemszenarien zu bewältigen.
In LINQ For Geometry ist der Rückgabetyp jedes Iterators im Code des Projekts, auch „Enumerator“ genannt, immer vom Typ IEnumerable<T>. Dieser macht die Iteration über eine Collection erst möglich. Enumeratoren können in C# dann in einer „foreach“ Anweisung oder LINQ Abfrage verarbeitet werden. Methoden, welche einen IEnumerator zurückliefern, können auch direkt als Argument in einer „foreach“ Anweisung eingesetzt werden. Im ersten Beispiel wird der einfache Aufruf eines LINQ For Geometrybasis Enumerators gezeigt. Das Ergebnisset kann in der foreach Anweisung weiterverarbeitet werden.

Quellcode Beispiele 2.12: EnumeratorForeach1.cs - Aufrufen von Enumeratoren Teil 1

```
// Use this enumerator to retrieve all faces that directly connect to the
    specific vertex.
foreach (HandleFace hFace in EnVertexAdjacentFaces(vertexHandle))
{
  ...
}
```

Hier im zweiten Beispiel 2.13 wird das Ergebnisset erneut durch eine LINQ Abfrage gefiltert. Dieses Beispiel zeigt, wie einfach man einen bestehenden Iterator/Enumerator erweitern oder verändern kann, ohne dass man dessen Implementierung direkt manipuliert. Die LINQ Abfrage „verändert“ durch nur wenig zusätzlichen Code sogar den Datentypen des Ergebnissets für die „foreach“ Anweisung.

Quellcode Beispiele 2.13: EnumeratorForeach1.cs - Aufrufen von Enumeratoren Teil 2

```
// This enumerator uses a LINQ statement to filter the result again.
// Get all half-edges that are adjacent to a specific face.
```

```
// And then just retrieve the half-edge pointer container for each already found half-edge.
foreach (HEdgePtrCont currentHedge in EnFaceAdjacentHalfEdges(faceHandle).Select(handleHalfEdge => _LhedgePtrCont[handleHalfEdge]))
{
  ...
}
```

Hier sind natürlich noch wesentlich komplexere LINQ Statements möglich um das Ergebnis an die eigenen Wünsche anzupassen. Es wäre zum Beispiel machbar, aus den herausgesuchten Half-Edges die Vertices zu extrahieren auf die sie zeigen. Somit hätte man durch einfache Erweiterung einen Enumerator geschaffen welcher die Vertices zu einem Faces findet. Diesen Lösungsweg setzt auch LINQ For Geometry im Quellcode ein um die Vertices eines Faces zu finden.

Aktuell existieren zehn verschiedene Enumeratoren in LINQ For Geometry. Es folgt eine Auflistung der Namen und der Funktionen der Iteratoren. Ebenfalls wird kurz darauf hingewiesen ob ein Iterator LINQ Ausdrücke einsetzt oder direkt auf der Datenstruktur operiert.

- EnAllVertices()
  - Dieser Iterator bietet lediglich die Handler Vertex Collection als IEnumerable an.
- EnAllEdges()
  - Dieser Iterator bietet lediglich die Handler Edge Collection als IEnumerable an.
- EnAllFaces()
  - Dieser Iterator bietet lediglich die Handler Face Collection als IEnumerable an.
- EnStarVertexVertex()
  - Der EnStarVertexVertex() Enumerator ist in der Lage alle direkt durch eine Kante zum Ausgangsvertex verbundenen Vertices zu finden.
  - Er wird durch eine LINQ Anweisung realisiert, die wiederum aus anderen von LFG bereitgestellten Enumeratoren besteht. Besser gesagt ist dieser Enumerator eine Kombination aus LINQ Abfragen und aus einem Algorithmus zur Kantenfindung welcher direkt, also „nativ“ auf der Datenstruktur arbeitet. Dabei handelt es sich um den EnVertexIncomingHalfEdge() Enumerator.

- EnVertexIncomingHalfEdge()
  - Dieser Enumerator ist einer der wichtigsten Bestandteile der Vertex Normalen Berechnung. Er liefert durch geschickte Nutzung der Datenstruktur alle Half-Edges des Geometry Objektes zurück, welche auf einen bestimmten Vertex zeigen. Er findet Kanten durch die Zirkulation auf Half-Edges um den Vertex herum.
  - Dieser Enumerator wird zusätzlich in anderen Enumeratoren benutzt.
- EnStarVertexOutgoingHalfEdge()
  - Im Gegensatz zum EnVertexIncomingHalfEdge() wird dieser Enumerator alle Half-Edges die von einem bestimmten Vertex ausgehen finden. Dazu setzt er ein LINQ Statement ein, das auf dem Ergebnisset von EnVertexIncomingHalfEdge() arbeitet.
- EnVertexAdjacentFaces()
  - Mit Hilfe dieses Enumerators ist es möglich alle Faces zu finden welche an einen gegebenen Vertex angrenzen.
  - Dieser setzt ebenfalls den EnVertexIncomingHalfEdge() Enumerator ein, um dann mit einem LINQ Statement dessen Ergebnis weiter zu verarbeiten.
- EnFaceAdjacentHalfEdges()
  - Es handelt sich hierbei um einen Enumerator der durch Ausnutzung der Topologie der Datenstruktur alle Half-Edges zu einem bestimmten Face ausfindig macht. Dazu iteriert die Methode so lange über die Next Pointer der Half-Edges, bis wieder die Half-Edge erreicht wird bei welcher der Algorithmus gestartet ist. Im Normalfall handelt es sich hierbei dann bei einem Mesh aus Quads um 4 Pointer Sprünge.
- EnFaceAdjacentVertices()
  - Mit diesem Enumerator können alle zu einem Face gehörigen Vertices gefunden werden.
  - Auch diese Methode setzt den EnFaceAdjacentHalfEdges() Enumerator ein und verwendet dessen Ergebnisset um durch eine LINQ Abfrage die gewünschten Informationen zu erhalten.
- EnFaceAdjacentFaces()
  - Hiermit kann der Benutzer alle direkt an die Kanten eines gegebenen Faces angrenzenden Faces ausfindig machen.

– Der Enumerator setzt dazu eine LINQ Abfrage auf das Ergebnisset des EnFaceAdjacentHalfEdges() Enumerators ein.

2.14.1 Die Verwendung von LINQ und Lambda in den Enumeratoren

LINQ Ausdrücke werden in den Enumeratoren recht häufig verwendet. In allen Fällen handelt es sich bisher um Select() Operationen auf einer Collection aus Daten. Natürlich kann sich dies in Zukunft durch die Erweiterungen und Wünsche der Benutzer ändern. Lambda ist auch in den Enumeratoren ein wichtiger Bestandteil der LINQ Methodensyntax. Allerdings werden bis jetzt nur sehr einfache Lambda Ausdrücke eingesetzt. Der folgende Ausdruck beschreibt ein LINQ Select Statement auf einer Collection aus Half-Edge Handles.

```
EnFaceAdjacentHalfEdges(faceHandle).Select(handleHalfEdge => _LhedgePtrCont[
    handleHalfEdge]._v)
```

Hier wird das Statement noch einmal in seine Bestandteile aufgeschlüsselt, um das Zusammenspiel des Lambda Ausdrucks auf der Collection des ersten Enumerators aufzuzeigen.

```
// Erster Enumerator Aufruf. Hierbei handelt es sich bei dem return Datensatz
    um eine Collection aus Half-Edge Handles.
EnFaceAdjacentHalfEdges(faceHandle);

// Die Collection des ersten Enumerators wird durch ein Select Statement
    weiter spezifiziert.
EnFaceAdjacentHalfEdges(faceHandle).Select(...);

//Der Lambda Ausdruck (oben durch "`..."' ersetzt) im Select Bereich der LINQ
    Abfrage sucht zu jeder Half-Edge Handle das zu ihr passende Vertex Handle.
handleHalfEdge => _LhedgePtrCont[handleHalfEdge]._v
```

2.14.2 Unterschiedliche Implementierungen von Iteratoren kurz dargestellt

Um das ganze System ein wenig zu verdeutlichen möchte diese Arbeit hier kurz die zwei wichtigsten Enumerator Typen vorstellen. Zum einen gibt es in LINQ For Geometry Enumeratoren, die ein Ergebnisset durch einige Operationen direkt auf den Verbindungen der Datenstruktur zusammen stellen. Enumeratoren welche diesem Lösungweg einschlagen sind meist sehr Performance kritisch. Sie werden in vielen anderen Enumeratoren wiederverwendet und sind zur Laufzeit sehr häufig aktiv. LINQ For Geometry implementiert die-

se Iteratoren deswegen nicht als reine LINQ Statements. Es wäre schlichtweg unnötig, tausenden von Half-Edges eines Meshes zu durchlaufen um herauszufinden welche Half-Edges zu einem bestimmten Face gehört. Bei dem ersten Beispiel handelt es sich um den EnFaceAdjacentHalfEdges() Enumerator. Er stellt eine wichtige Methode für fast alle Berechnungen auf der Datenstruktur, von den Face Normalen bis zur Triangulation, dar. Wie zu sehen, iteriert diese Methode mit Hilfe einer doWhile() Schleife über die begrenzenden Half-Edges eines Faces. Dieser Enumerator ist durch die wenigen Pointer Sprünge die er durchführen muss sehr schnell. Genau das macht ihn zu einer guten Basis für weitere Iteratoren.

Quellcode Beispiele 2.14: Geometry.cs - EnFaceAdjacentHalfEdges()

```
1        public IEnumerable<HandleHalfEdge> EnFaceAdjacentHalfEdges(HandleFace
    faceHandle)
2        {
3            int startHedgeIndex = _LfacePtrCont[faceHandle]._h;
4            int currentIndex = startHedgeIndex;
5            List<HandleHalfEdge> LHedgeHandles = new List<HandleHalfEdge>();
6            do
7            {
8                LHedgeHandles.Add(new HandleHalfEdge(currentIndex));
9                if (_LhedgePtrCont[_LhedgePtrCont[currentIndex]._nhe]._f !=
    faceHandle)
10                   break;
11
12               currentIndex = _LhedgePtrCont[currentIndex]._nhe;
13           } while (currentIndex != startHedgeIndex);
14           return LHedgeHandles.AsEnumerable();
15       }
```

Weiterhin gibt es aber auch Iteratoren, die tatsächlich aus einem einzeiligen LINQ Statement bestehen. Fast immer verarbeiten diese Iteratoren die Ergebnisse eines der „nativ“ auf der Datenstruktur ausgeführten Enumeratoren. Sie erweitern oder verändern dessen Ergebnisse. Diese in LINQ formulierten Methoden müssen grundsätzlich wesentlich kleinere Collections an Daten verarbeiten. Aus diesem Grund eignen sich LINQ Abfragen hierfür wunderbar. Sie sind somit zugänglich und gerade noch hinreichend komplex gestaltet um auch in Zukunft eine Basis für neue Traversierungsmethoden zu bieten. Das Ergebnis eines jeden Enumerators könnte durch neue Operationen also noch weiter modifiziert und noch umfangreicher gestaltet werden.

Quellcode Beispiele 2.15: Geometry.cs - EnFaceAdjacentVertices()

```
1        public IEnumerable<HandleVertex> EnFaceAdjacentVertices(HandleFace
    faceHandle)
2        {
3            return EnFaceAdjacentHalfEdges(faceHandle).Select(handleHalfEdge
    => _LhedgePtrCont[handleHalfEdge]._v).AsEnumerable();
4        }
```

2.14.3 Die Geschwindigkeit der Half-Edge Datenstruktur in den Enumeratoren

Wie in der oberen Übersicht über die Iteratoren 2.14 erklärt wurde, gibt es einige Iteratoren die für die Traversierung der Half-Edge Datenstruktur sehr wichtig sind. Diese Iteratoren wurden dann so konstruiert, dass ihre Operationen direkt auf der Datenstruktur ausgeführt werden. Um aus den bestehenden weitere Iteratoren zu schaffen, können die bereits angebotenen Enumeratoren wieder verwendet werden. Der Enumerator EnFaceAdjacentHalfEdges() ist durch seine massive Wiederverwendung in mehreren anderen Enumeratoren einer der wichtigsten Enumeratoren im Projekt. Es war daher wichtig, diese Methode auf Geschwindigkeit zu optimieren. Die folgenden Schritte zeigen den Unterschied zwischen dem LINQ basierten Enumerator und dem „nativ“ auf der Datenstruktur entworfenen Iterator auf. Ziel ist es also die hier als Basis Enumeratoren bezeichneten Methoden zu untersuchen und die besten Möglichkeiten zukünftig in jeder Erweiterung der Funktionalität von LINQ For Geometry zu verwenden. Diese Basis bietet einen schnellen Einstieg in den Entwurf weiterer Traversierungsalgorithmen durch seine vielen Kombinationsmöglichkeiten.
Es ist unbedingt zu beachten, dass die Größe der Quelldaten Collection eine erhebliche Auswirkung auf die Geschwindigkeit eines (LINQ)Iterators hat. So ist es also essentiell, dass Enumeratoren wie EnFaceAdjacentHalfEdges() im Hinblick auf Geschwindigkeit entworfen werden. Ein Beispiel kann am Iterator EnVertexIncomingHalfEdge(HandleVertex vertexHandle) festgemacht werden. So ist seine Implementierung in Form des folgendes Codes die schnellste im Moment denkbare Version dieses Iterators im Projekt. Allerdings könnte der Iterator zu Testzwecken auf zwei verschiedene Arten im Projekt integriert werden.

Der erste Versuch gestaltete den Enumerator als direkt auf der Topologie der Half-Edge Datenstruktur arbeitenden Algorithmus.

Quellcode Beispiele 2.16: Geometry.cs - EnVertexIncomingHalfEdge()

```
1        public IEnumerable<HandleHalfEdge> EnVertexIncomingHalfEdge(
    HandleVertex vertexHandle)
2        {
3
4            List<HandleHalfEdge> LTmpIncomingHedges = new List<HandleHalfEdge
    >();
5
6            //Get the one outgoing half-edge for the vertex.
7            HandleHalfEdge currentHedge = _LvertexPtrCont[vertexHandle]._h;
8            //Remember the index of the first half-edge
9            int startHedgeIndex = currentHedge;
10           do
11           {
12               if (currentHedge == -1)
13                   break;
14
15               HEdgePtrCont currentHedgeContainer = _LhedgePtrCont[
    currentHedge];
16               if (vertexHandle == _LhedgePtrCont[currentHedgeContainer._he].
    _v)
17               {
18                   LTmpIncomingHedges.Add(currentHedgeContainer._he);
19               }
20               currentHedge = _LhedgePtrCont[currentHedgeContainer._he]._nhe;
21           } while (currentHedge != startHedgeIndex);
22
23           return LTmpIncomingHedges.AsEnumerable();
```

Hier die gleiche Funktionalitat formuliert als LINQ Abfrage in Methodensyntax.

```
1 return _LhedgePtrCont.Where(e => e._v == vertexHandle).AsParallel().Select(e
    => _LhedgePtrCont[e._he._DataIndex]._he).AsParallel().ToList();
```

Formuliert als Abfragesyntax:

```
1 return (from e in _LhedgePtrCont where e._v == vertexHandle select
    _LhedgePtrCont[e._he]._he).AsParallel().ToList();
```

Betrachtet man die Methode CalcVertexNormal() welche den EnFaceAdjacent-HalfEdges() Enumerator massiv einsetzt zeigt sich folgender Sachverhalt. Aktuell (Stand August 2013) fallen die FPS von durchschnittlich 99FPS laut FRAPS[16] , mit Vertex Normalen Berechnung und nativem Algorithmus auf

[16]Demo unter http://www.fraps.com/

der Datenstruktur (alg1), auf durchschnittlich 3FPS mit den LINQ Abfragen in Abfragesyntax (alg2) und Methodensyntax (alg3). Dieses Beispiel ist auf das Utah Teapot Model mit aktivierter Beleuchtungsberechnung und „Chessboard“ Textur bezogen. Die folgenden Messwerte entstammen dem Visual Studio 2010 Profiler Tool und weiterer Messungen mit Fraps. Ein Überblick über die Messwerte gibt das Diagramm 2.14.5.

Um die Messungen[17] [18] zu vergleichen wurden die 3D Modelle durch eine im Rahmen dieser Arbeit erstellten automatisch ablaufenden Demo Anwendung transformiert. Jedes Modell wurde dreissig Sekunden lang in verschiedene Richtungen rotiert. Bei jeder Messung wurde eine Textur verwendet. Die Messung erfolgte einmal mit und einmal ohne Vertex Normalen Berechnung und Beleuchtung in Fusee.

Die Kennzahlen der folgenden VS2010 Profiler Messung sind als Inklusive und Exklusive Samples angegeben. Inklusive Samples beschreiben die Zeit die in einer Funktion inklusive ihrer Kinder Methoden für eine Berechnung benötigt wird. Exklusive Samples beschreiben die Zeit welche im Body der Methode selbst ohne Kinder Methoden verbracht wird.

> „Die Inklusiv-Werte werden anhand der Gesamtzahl der für eine Funktion in der Aufrufliste erfassten Samplings berechnet. Dies gilt auch für die beim Ausführen untergeordneter Funktionen erfassten Samplings. Wenn eine erfasste Aufrufliste analysiert wird, werden die Werte für die inklusiven Samplings aller Funktionen der Aufrufliste erhöht.“ [Mic13, Analyse von Leistungsdaten]

> „Die Exklusiv-Werte werden anhand der Gesamtzahl an der für eine Funktion erfassten Samplings berechnet, wenn diese ihren eigenen Code ausführt. Exklusive Werte enthalten kein Samplings, die beim Ausführen untergeordneter Funktionen erfasst werden. Wenn eine erfasste Aufrufliste analysiert wird, wird die exklusive Samplinganzahl nur für die Funktion erhöht, die Code ihres Funktionstexts ausführt.“ [Mic13, Analyse von Leistungsdaten]

[17] Messungen durchgeführt auf folgender PC Konfiguration: Intel Core i5 2500k, 16GB 1333er Ram, HDD Festplatte 7200RPM, ATI Radeon HD 7950 3GB VRam, Windows 7 64Bit auf Samsung 840 SSD. Messwerte berechnet durch den Visual Studio 2010 Profiler

[18] FPS Werte durch Fraps Benchmark Funktion ermittelt, höhere FPS = besser.

Messergebnis für den nativen EnVertexIncomingHalfEdge() Algorithmus auf der Datenstruktur (alg1) aus 2.14.3

Abbildung 2.3: Messung der CalcVertexNormal() Methode

Laut Profiler ist also der in der CalcVertexNormal() Methode benutze Enumerator EnVertexIncomingHalfEdge() [19] mit 4.4% Inklusiven Samplings in seiner Ausführung sehr sparsam. Der Algorithmus, welcher die Topologie der Half-Edge Datenstruktur ausnutzt um keine riesigen Datenmengen bearbeiten zu müssen, ist in dieser Form also sehr effizient.

Hier ist zu sehen, dass die Ausführung des Enumerators im gesamt Kontext einer Messung einen sehr kleinen Anteil sowohl an Inklusiven Samplings (4,38%) und Exklusiven Samplings (1.74%) hält.

	Inklusive Samplings	Exklusive Samplings	Inklusive Sampling...	Exklusive Samplin...
EnFaceAdjacentVertices(valuetype Linq	675	38	4,71	0,27
EnVertexIncomingHalfEdge(valuetype	628	250	4,38	1,74
LoadAsset(string)	148	0	1,03	0,00
ReuseExistingConnection(valuetype Lir	1	1	0,01	0,01

Abbildung 2.4: Messung des Enumerators EnVertexIncomingHalfEdge() im Gesamtkontext einer Demo Anwendung ohne LINQ. Die relevante Zeile ist blau markiert.

[19] Beschriebener Wert blau markiert

Messergebnis für die Methodensyntax (alg2) aus 2.14.3

Abbildung 2.5: Messung der CalcVertexNormal() Methode.

Bei dieser Variante sind die Inklusiv Samples für den Enumerator mit 52.8% sehr viel höher als im vorigen Vergleich 2.3. Der hohe Wert ergibt sich hier durch die LINQ Abfrage innerhalb des Iterators und deren Aufrufe an das System. Zu sehen in Abbildung 2.6.

Abbildung 2.6: Messung des Enumerators EnVertexIncomingHalfEdge() in Methodensyntax.

Die Inklusiven Samplings dieser Methode sind sehr gering. Dieser Zustand ergibt sich daraus, dass ausser dem LINQ Query Aufruf und dem Return Wert nichts weiter in dieser Funktion geschieht. Die große Auslastung dieser Methode schlägt sich im hohen Wert der Inklusiv Samples nieder. Zur Methode in 2.3 besteht ein Unterschied von 48,42%. Die Auslastung des benutzen CPU KernsTest Systems lag mit dieser Methode nahe an 100% und nie unter 80%.

	Inklusive Samplings	Exklusive Samplings	Inklusive Sampling...	Exklusive Samplin...
EnFaceAdjacentVertices(valuetype LinqForG	15	1	0,16	0,01
EnVertexIncomingHalfEdge(valuetype Linq	5.104	3	52,80	0,03
LoadAsset(string)	197	0	2,04	0,00
ToMesh(bool)	5.435	12	56,22	0,12

Abbildung 2.7: Messung des Enumerators EnVertexIncomingHalfEdge() in Methodensyntax im Kontext eines Demo Ablaufs.

Die hohen Messergebnisse des Iterators resultieren aus der sehr großen Collection an Half-Edges über welche die LINQ Abfrage iterieren muss, um alle eingehenden Half-Edges eines Vertex zu finden. Der Native Algorithmus benötigt hier im Gegenzug nur ein paar Pointer Sprünge über Nachbar (Twin)Half-Edges wie in 2.14.3 dargestellt.

Messergebnis für die Abfragesyntax (alg3) aus 2.14.3

Abbildung 2.8: Messung der CalcVertexNormal() Methode.

In der Abfragesyntax steigt der Wert der Inklusive Samples des Enumerators im Gegensatz zur Methodensyntax. Er liegt hier bei 94.3%. Dieser Zustand ergibt sich wie in Abbildung 2.9 ersichtlich aus dem höheren Aufwand des ToList() Konverters. Eine weitere Parallelisierung im Enumerator durch Angabe von AsParallel(), wie in 2.14.3, kann hier wegen einer Einschränkung der Abfragesyntax nicht erfolgen.

Abbildung 2.9: Messung des Enumerators EnVertexIncomingHalfEdge() in Abfragesyntax.

Im Kontext einer kompletten Demo Anwendung steht der Enumerator mit 14846 Inklusiv Samples im Vergleich mit dem ersten vorgestellten Algorithmus in Abbildung 2.4 nicht sehr gut da.

	Inklusive Samplings	Exklusive Samplings	Inklusive Sampling...	Exklusive Samplin...
EnFaceAdjacentVertices(valuetype LinqForG	32	0	0,20	0,00
EnVertexIncomingHalfEdge(valuetype LinqF	14.846	4	94,30	0,03
LoadAsset(string)	192	0	1,22	0,00

Abbildung 2.10: Messung des Enumerators EnVertexIncomingHalfEdge() in Abfragesyntax im Kontext eines Demo Ablaufs.

2.14.4 Zusammenfassung der Erkenntnisse aus den Messwerten

Aus den genannten Werten ist also ersichtlich, dass LINQ Abfragen auf großen Datenmengen in LFG nach Möglichkeit vermieden werden sollten. Im Falle der Messwerte handelt es sich bei dem Utah Teapot um eine Liste aus 1400+ (Half-Edge) Elementen. Diese Elemente müssen für jeden Vertice, um seine Normale zu berechnen, einmal durchsucht werden. Somit würde sich mit den LINQ Abfragen mindestens ein Algorithmus der Komplexität n^2 ergeben. Wann immer es möglich ist, sollten die Vorteile der Datenstruktur im Entwurf von Enumeratoren/Iteratoren auch benutzt werden um die Performance so gut es geht zu erhalten.

2.14.5 Benchmarking mit verschiedenen 3d Modellen

Das folgende Diagramm 2.14.5 beschreibt Benchmark Ergebnisse welche mit Hilfe einer Fusee Demo Anwendung[20] ermittelt wurden. Gemessen wurde die Transformation von verschiedenen beispielhaften 3d Modellen. Zu diesen Modellen zählen einfachste geometrische Gebilde aber auch komplexere Modelle aus der Computergrafik und ein Modell aus dem 3D Computerspiel „Doom 3“. Insgesamt wurden zwölf Messungen durchgeführt. Die ermittelte Größe Frames per Second (FPS) kann hier an der x-Achse des Diagramms abgelesen werden. Alle Modelle wurden mit und ohne Vertex Normalen Berechnung gemessen. Die Fusee seitige Beleuchtung war nur im Falle der Messung mit der Vertex Normalen Berechnung aktiv. Für jedes Modell wurde die gleiche Textur verwendet.

Diese Messung zeigt den Einfluss der Vertex und Polygon Anzahl auf das System zur Laufzeit. Bei den komplexeren Modellen mit einer Polygonanzahl $>$ 200 führt das Aktivieren der Vertex Normalen Berechnung zu einer reduktion der FPS auf 1/3 der FPS der Basis Messung ohne Vertex Normalen Berechnung. Diese Tatsache zeigt, dass eine Berechnung solch großer Datenmengen und der darin benutzen Iterationen einiges an Leistung kostet. Die verwendete Methode zur Berechnung der Vertex Normalen ist Kantenbasiert und kann daher nur in einer Datenstruktur wie der Half-Edge Datenstruktur eingesetzt werden. In einer Face basierten Datenstruktur wäre diese Berechnung nur noch weiteren erheblichen Mehraufwand zu schaffen.

[20] Die für die Messung benutze Demo Anwendung befindet sich im Fusee Beispielprojekt von LINQ For Geometry. Sie kann durch verschiedene Settings aktiviert werden. Der Quellcode dokumentiert die benötigten Variablen.

Benchmark LFG/Fusee
VSync=disabled
FPS
0
1000
2000
3000
4000
5000
6000
7000
Models
5979
5451
2213
Cube (Triangles)
verts=8, polys=12
5393
5184
2067
Cube (Triangles) w. Normal calc.
verts=8, polys=12
5948
5446
2185
Cube (Quads)
verts=8, polys=6
5257
4942
2146
Cube (Quads) w. Normal calc.
verts=8, polys=6
1249
1219
933
Sphere (Quads)
verts=266, polys=288
436
432
394
Sphere (Quads) w. Normal calc.
verts=266, polys=288
569
557
492
Cylinder (Quads)
verts=758, polys=720
180
177
169
Cylinder (Quads) w. Normal calc.
verts=758, polys=720
302
297
279
Teapot(Mixed)
verts=1297, polys=1296
102
99
90
Teapot(Mixed) w. Normal calc.
verts=1297, polys=1296
280
276
256
Doom3 Hellknight (Triangles)
verts=1737, polys=2626
100
98
96
Doom3 Hellknight (Triangles) w. Normal calc.
verts=1737, polys=2626
Max FPS
Average FPS/Time
Min FPS

2.15 Manipulation von Mesh Daten in der Datenstruktur

Um die Mesh Daten eines Geometry Objektes zu manipulieren gibt es verschiedene Möglichkeiten. Durch Manipulationen kann das Mesh geringfügig oder gar komplett verändert werden. Es ist allerdings wichtig, keine Verbindungen der Datenstruktur im Geometry Objekt zu beschädigen. Ein Geometry Objekt hält zwei verschiedene Arten von Daten vor. Zum einen sind dies „reale Daten“. Dabei handelt es sich um tatsächliche Werte, also in etwa Vertex Koordinaten, Vertex Normalen, Face Normalen und UV Koordinaten. Zum anderen existieren virtuelle Beziehungen der einzelnen Elemente der Datenstruktur untereinander, die hier als Half-Edges, Edges, Vertices und Faces bezeichnet werden. Um also die Form eines Meshes zu verändern müssen also in aller Regel die Beziehungen der Elemente verändert werden. Ein einfaches Beispiel dazu wäre die Triangulation eines Meshs. Es werden hierbei keine neuen realen Daten erzeugt sondern lediglich Half-Edges, Edges und Faces manipuliert.

Beide Manipulationsarten können aber auch kombiniert werden. Ein klassischer Fall wäre ein Subdivision Surfacing Algorithmus welcher durch Erweiterung des Geometry Objektes das Mesh in seiner Form verändern würde. Hierbei verändern sich alle Teile eines Geometry Objektes. Es entstehen neue Kanten und Faces und es werden neue Vertices aus bereits vorhandenen berechnet und eingefügt.

2.15.1 Manipulation von Edges und Half-Edges

Half-Edges sind das wichtigste Konstrukt in der Datenstruktur. Daher sind sie auch der interessanteste Ansatz zur Manipulation von Meshes. Die Triangulation eines Quadrilateral Meshes in LINQ For Geometry ist ein gutes Beispiel für eine einschlägige Veränderung des Mesh Objekts durch Manipulation und Einfügen neuer Half-Edges. Durch wenige Änderungen an den Pointern der vorhandenen Half-Edges und das einfügen von zweier neuer Half-Edges kann ein Quad trianguliert werden.

2.15.2 Manipulation von Vertices

Vertices können in LINQ For Geometry durch Veränderung ihrer Half-Edge Pointer oder durch direkte Änderung ihrer tatsächlichen Werte manipuliert werden. Diese Tatsache wird von LINQ For Geometry im Algorithmus für die Mesh Skalierung eingesetzt. Durch Matrixmultiplikation jedes Vertex mit der gleichen Skalierungsmatrix kann das Objekt entlang aller drei Achsen (X,

Y, Z) gleichzeitig oder getrennt skaliert werden. Um z.B. eine Translation zu erreichen werden alle Vertices mit einer anderen Matrix multipliziert. Das gleiche gilt für eine Rotation. Vertex Manipulation ist also unersetzlich für die Veränderung der Daten im dreidimensionalen Raum.

2.15.3 Manipulation von Faces

Faces bieten am wenigsten Manipulationsmöglichkeiten in LINQ For Geometry. Nur der Index auf die Normale und der Pointer auf eine begrenzenden Half-Edge des Face kann verändert werden. Faces besitzen keine physischen Daten (bis auf die Face Normale). Sie sind darum aber nicht weniger wichtig. Denn viele Manipulationsalgorithmen wie die Triangulation eines Meshes oder die Vertex Normalen Berechnung befassen sich mit den Faces als Einstiegspunkt in die geometrische Operation.

2.16 Beispielhafte Implementierung von Standard Algorithmen zur Geometriemanipulation als Modul

LINQ For Geometry bietet zur beispielhaften Manipulation eines Geometry Objekts verschiedene Methoden an. Alle diese Methoden sind im Modul Projekt LinqForGeometry.ExternalModules in der Static Klasse Transformations zu finden.
Zu ihnen zählen:

- Die Transformation des Objekts durch die Übergabe von Faktoren pro Achse.
- Die Skalierung durch die Angabe von Faktoren pro Achse.
- Eine Translation durch Übergabe von Faktoren pro Achse.
- Eine Rotation um die Globale X, Y oder Z Achse.

2.16.1 Transformation mit einer Transformationsmatrix

Um die Vertices eines Meshes zu transformieren wird eine zuvor in der Methode zusammengestellte Transformationsmatrix auf jeden Vertex des Meshes angewendet. Ohne LINQ würde man in einer Schleife über jeden Vertex laufen und das Ergebnis wieder in eine neue Liste schreiben mussen. Diese Liste müsste dann die aktuelle Vertex Liste im Mesh überschreiben um die neuen Werte zu übernehmen.
Durch die Nutzung von LINQ entwickelt sich der oben genannte Algorithmus zu einem Einzeiler und zeigt hier sehr gut warum LINQ die Entwicklung solcher

Algorithmen sehr viel lesbarer und einfacher gestaltet. Die hier gezeigte LINQ Abfrage ist in der Methodensyntax geschrieben.

Quellcode Beispiele 2.17: Transformation.cs - Manipulation der Vertices

```
1        public static bool Translate(float tX, float tY, float tZ, ref
    Geometry geo)
2        {
3            try
4            {
5                float4x4 transfMatrix = new float4x4(
6                        new float4(1f, 0f, 0f, tX),
7                        new float4(0f, 1f, 0f, tY),
8                        new float4(0f, 0f, 1f, tZ),
9                        new float4(0f, 0f, 0f, 1f)
10                   );
11
12               geo._LvertexVal = geo._LvertexVal.Select(tmpVertValue =>
    transfMatrix * tmpVertValue).ToList();
13
14               return true;
```

Das Geometry Objekt wird dieser Methode lediglich Call by Reference übergeben um die benötigte Rechenleistung zu minimieren und die Bearbeitung mit der einzeiligen LINQ Funktion möglich zu machen. So werden die Daten direkt ohne zwischenspeichern ressourcenschonend auf der Quelle verändert, ohne dass das Ergebnisset dann nochmal explizit gespeichert oder übergeben werden muss.

2.17 Berechnungen für die Darstellung eines Meshes in der Fusee Engine

Um ein in der LINQ For Geometry Half-Edge Datenstruktur gespeichertes Mesh in der Fusee Engine zu rendern bedarf es einiger Operationen. Die wichtigsten Operationen für die Vorbereitung der Darstellung sind alle in der LFG Methode ToMesh() zu finden. Diese Methode kann als Konverter bezeichnet werden. Um LFG an eine andere Engine anzupassen müsste diese Operation falls nötig geändert werden. ToMesh() prüft zuerst ob der Benutzer eine Triangulation des Meshes wünscht. Falls ja wird das Mesh beim ersten Rendern nach ToMesh trianguliert. Diese Änderung ist zur Laufzeit nicht Rückgängig zu machen. Es folgen jetzt einige Schritte die für eine spätere Beleuchtungsberechnung in Fusee nötig sind.
Die Face Normalen werden jetzt durch CalcFaceNormal(HandleFace face) berechnet. Diese Methode sucht alle Vertices eines Faces heraus und berechnet aus den ersten drei Vertices des Faces durch das Kreuzprodukt eine Normale für das aktuelle Face. Diese Normale wird jetzt in eine Liste eingetragen und ein Handle auf sie wird am Face gespeichert. Die Face Normalen sind ein wichtiges Element um im nächsten Schritt die Vertex Normalen berechnen zu können welche in Fusee für eine korrekte Beleuchtung des Modells verantwortlich sind. Vertex Normalen für Fusee werden in LINQ For Geometry durch die Methode CalcVertexNormal(HandleVertex vertex) berechnet. Diese Berechnung ist eine der aufwändigsten Kalkulationen im Projekt. Für jede Eingehende Half-Edge auf einen Vertex wird eine Vertex Normale berechnet. Daraus ergibt sich, dass jedes Face beim Konvertieren nach Fusee.Mesh für den Vertex jeden Vertex eine Normale erhält um Kanten und Abstufungen korrekt auszuleuchten. Die Methode gliedert sich wie folgt:

- Es werden alle eingehenden Half-Edges eines Vertex gesucht.
- Für jede Half-Edge gilt Folgendes:
  - Prüfe ob die Half-Edge zu einem Face gehört um keine Löcher im Mesh anzusprechen und dadurch Nullpointer zu provozieren.
  - Frage die Normale des zur Half-Edge gehörenden Faces ab.
  - Lege eine Ergebnisvariable an welche später alle nötigen aufsummierten Normalen enthält.
    - Iteriere über jede andere eingehende Half-Edge.
    - Entspricht diese Half-Edge jener für welche gerade die Normale berechnet wird, so addiere einmal ihre Normale zum Ergebnis.

    - Ist es eine andere Half-Edge, dann vergleiche den Winkel zwischen beiden Face Normalen und addiere die Half-Edge nur wenn der Winkel kleiner als der in _constSmoothingAngle gewählte Threshold ist.

  - Sind alle nötigen Normalen addiert muss der erhaltene Ergebniswert an der Half-Edge gespeichert und der Algorithmus bearbeitet die nächste Half-Edge im gesamt Set.

Diese Berechnung der Vertex Normalen kann durch Setzen der Boolean Variable _VertexNormalActive in Geometry ein oder ausgeschaltet werden. Diese Funktion wurde zu Test Zwecken während dieser Arbeit implementiert und kann zum Debugging benutzt werden.

Sind jetzt alle Normalen berechnet, iteriert ToMesh() noch einmal über alle Faces und Half-Edges und sammelt die relevanten Informationen in Collections, die für Fusee später in Arrays konvertiert werden. Hierbei werden, da Fusee aktuell nur mit Triangles arbeitet (Stand Juli 2013), auch die Triangles neu berechnet. Dazu lauft ein Iterator über die Half-Edges des Meshes und speichert die Indizes der Vertices in der richtigen Reihenfolge in einer Liste ab. Die meisten OpenGL Echtzeit 3D Engines benutzen aktuell nur noch Triangles für die repräsentation der Meshes. OpenGL hat die Funktion GL_QUADS in OpenGL 3.1 und darüber entfernt und in 3.0 gilt diese Funktion als "deprecated".

> „GL_QUADS [...] have been removed from core OpenGL 3.1 and above (they are only deprecated in OpenGL 3.0).[21] “

Man hat sich für Triangles entschieden, da sie zur Repräsentation von Meshes sehr gut geeignet sind. Ein Quad besteht genauer gesagt nur aus zwei Triangles und deswegen sollte die OpenGL API nicht mit zusätzlichen Konvertierungen belastet werden. Die Khronos Group, welche für die Entwicklung und Wartung von OpenGL zuständig ist beschrieb im März 2009 in der Spezifikation zu OpenGL 3.1, dass die OpenGL API in Zukunft „ausgedünnt“ oder „schlanker“ gemacht werden soll.[22]

[21] `http://www.opengl.org/wiki/Primitive` (geprüft 30.07.2013 Abschnitt „Quads“)
[22] `https://www.khronos.org/news/press/khronos-releases-streamlined-opengl-3.1-specification`

> „Many algorithms use triangle meshes for representing geometric surfaces. The simple triangle can serve as a basic surface primitive to adaptively approximate smooth freeform geometry. A triangle mesh naturally allows to adapt the required number of primitives to the geometric shape instead of the basic topology of an arbitrary object. Triangle meshes are efficient, since the triangle is currently the only surface patch that is directly supported by computer graphics hardware and even software rendering methods like ray tracing or radiosity prefer to use piecewise linear primitives to represent geometric objects due to gains in performance.“ [CKS98, S. 1]

Um die Konvertierung in ein Fusee Mesh Objekt abzuschliessen, muss die ToMesh() Methode ein Objekt von dem Fusee bekannten Datentypen „Fusee.Mesh“ erzeugen und es an den Aufrufer, in Fall dieses Projektes dem Fusee Beispiel Projekt „LinqForGeometry“, zurückgeben. Das so erzeugte Objekt kann dann in der eigenen Fusee App oder eben dem Beispielprojekt mit einem Aufruf auf die im RenderContext enthaltene Methode „Render()“, RC.Render(Mesh meshObjekt), angezeigt werden.

3 Schluss

3.1 Fazit

Zum Ende dieser Arbeit hat sich LINQ For Geometry von einer simplen Datenstruktur in eine in die Fusee Engine integrierte Umgebung zum Handling und zur Manipulation von 3D Meshes entwickelt. Das Projekt hat einen Zustand erreicht, in welchem es möglich ist, Meshes zu laden, sie anzuzeigen und verschiedene Elemente des Meshes gezielt zur Laufzeit zu verändern. Texturen und Vertex Normalen Beleuchtungsberechnung werden in der Datenstruktur durch die Integration in Fusee unterstützt. Das Projekt kann um eigenen Code erweitert werden und bietet eine grundlegende Auswahl an Enumeratoren/Iteratoren für spätere Entwicklungen. Es wäre interessant mit diesem Projekt einen 3D Echtzeit Editor in der Fusee Engine zu implementieren. LINQ Abfragen können nach Wunsch verwendet werden um eigene Enumeratoren/Iteratoren zu entwicklen oder bestehende zu kombinieren. Einige LINQ Abfragen werden für Transformationen auf den Meshes genutzt. Hierzu werden Transformationsmatrizen mit Vertices multipliziert.
Trotz der vielen Möglichkeiten in Iteratoren und Transformationen muss bei der Verwendung der Half-Edge Datenstruktur in Verbindung mit LINQ auf die Größe der Quelldatensätze in Relation mit dem erwarteten Ergebnisset geachtet werden. Auch hier gelten die allgemeinen Regeln zur Entwicklung von Algorithmen. Über eine große Liste mit hunderten von Datensätzen zu iterieren um einen oder wenige Werte abzurufen lässt sich auch durch LINQ kaum performant gestalten. Vor allem bei zeitkritischen Operationen kann dies ein Problem darstellen. Algorithmen sollten in einem solchen Fall die Topologie der Datenstruktur nutzen.
Die grössten Stärken zeigen LINQ und Lambda dann, wenn relativ kleine oder bereits gefilterte Ergebnis Datensätze bearbeitet, weiter gefiltert oder auf andere Datentypen gemapped werden müssen. Diese LINQ Statements sind oft als sehr effektive Einzeiler zu schreiben und leicht zu verstehen und zu verändern. Im Projekt wurden also komplexe, zeitkritische Algorithmen als „native“ Operationen direkt auf der Topologie Datenstruktur entworfen. Es hat sich gezeigt, dass aus wenigen dieser Algorithmen, eine ganze Reihe an grundlegenden

LINQ basierten Iteratoren kombiniert werden konnte. Die Basis für fünf der sieben bereit gestellten Iteratoren bilden die zwei wichtigsten Iteratoren des Programms, welche wiederum als direkte Operationen auf der Datenstruktur entworfen wurden.
Der Support für LINQ und Lambda in einer Echtzeit 3D Datenstruktur ist also durchaus möglich. Er unterliegt durch die genannten Punkte einigen wenigen Einschränkungen, ist aber sicherlich als nützliche Erweiterung der Datenstruktur und dem gesamten LINQ For Geometry System als Datenstruktur zur Bearbeitung von Meshes zu verstehen.

3.2 Ausblick auf die zukünftige Verwendung von LINQ For Geometry

LINQ For Geometry ist an sich nur eine Basis für weitere Entwicklungen. Das System kann an manchen Stellen noch optimiert werden. Den Anspruch an eine vollständig integrierte Version einer Echtzeit 3D Datenstruktur hatte LINQ For Geometry ohnehin nie. In Zukunft ist es angedacht, das Projekt möglicherweise als Datenstruktur für einen Fusee internen 3D Echtzeit Mesh Editor zu verwenden und weitere Algorithmen zur Geometriemanipulation zu formulieren. Das System kann aber auch auf andere 3D Editoren oder Viewer angepasst werden und darf selbstverständlich unter Beachtung der bestehenden Lizenzen in diese integriert werden. So ist also zu hoffen, dass diese Arbeit einen Grundstein für weitere interessante Projekte, nicht zuletzt in der Echtzeit 3D Engine Fusee von Herrn Prof. Müller und seinem Team der Hochschule Furtwangen, legen konnte.

3.3 Wo ist der Code und die Dokumentation zu LINQ For Geometry zu beziehen und einzusehen?

Sollte am Code von LINQ For Geometry Interesse bestehen, ist dieser für alle Interessenten, auch hochschulextern, unter den folgenden GitHub Links abzurufen.

- Im Branch feat_dsteffen_LFG unter `https://github.com/FUSEEProjectTeam/Fusee` (geprüft 01.08.2013)
- Oder im LFG eigenen GitHub Repository ohne Fusee Integration. Hier ist der Code allerdings nicht ohne Anpassungen in den Methoden zur Meshkonvertierung lauffähig. `https://github.com/DomS23/LinqForGeometry` (geprüft 01.08.2013)

Der gesamte Code dieser Arbeit ist „Open Source“ und steht unter der MIT Lizenz, abrufbar auf der Webseite der “Open Source Initative„ unter `http://opensource.org/licenses/MIT` (geprueft: 07.08.2013). Weitere Lizenz Informationen zum Code finden sich in der MIT-License.txt im Root Verzeichnis des Softwareprojekts.

Appendix

Verzeichnis der Quellcode Beispiele

2.1 LINQeasyQuery.cs - Einfaches LINQ Abfrage Beispiel 16
2.2 LINQdirectQueryList.cs - Einfaches direktes LINQ Abfrage Beispiel [Mic13] 16
2.3 LINQzweiSynt.cs - Zwei syntaktische Möglichkeiten [Mic13] . . . 17
2.4 LambdaEasy.cs - Einfacher Lambda Ausdruck 19
2.5 GeoFace.cs - Ein temporärer Face basierter Datentyp. 26
2.6 HandleHalf-Edge.cs - Variablen Deklaration des „Zeigers“ 29
2.7 HandleHalf-Edge.cs - Impliziter cast nach Integer 29
2.8 HandleHalf-Edge.cs - Is Valid? 30
2.9 HandleHalf-Edge.cs - Deklarationen als internal 30
2.10 PtrContExample1.cs - Manipulation eines Pointer Containers . . 31
2.11 ExternalModule.cs - Wie wird ein Geometry Objekt im externen Modul verwendet 37
2.12 EnumeratorForeach1.cs - Aufrufen von Enumeratoren Teil 1 . . 39
2.13 EnumeratorForeach1.cs - Aufrufen von Enumeratoren Teil 2 . . 39
2.14 Geometry.cs - EnFaceAdjacentHalfEdges() 43
2.15 Geometry.cs - EnFaceAdjacentVertices() 43
2.16 Geometry.cs - EnVertexIncomingHalfEdge() 45
2.17 Transformation.cs - Manipulation der Vertices 55

Tabellenverzeichnis

Standardoperatoren in LINQ 18

Abbildungsverzeichnis

2.1 Diese Illustration zeigt die Verbindung der data records an einem beliebigen Polygon durch Pfeilverbindungen. 9
2.2 Diese Illustration zeigt die verschiedenen Layer von LFG und deren Verbindungen. 20
2.3 Messung der CalcVertexNormal() Methode 47
2.4 Messung des Enumerators EnVertexIncomingHalfEdge() im Gesamtkontext einer Demo Anwendung ohne LINQ. Die relevante Zeile ist blau markiert. 47
2.5 Messung der CalcVertexNormal() Methode. 48
2.6 Messung des Enumerators EnVertexIncomingHalfEdge() in Methodensyntax. 48
2.7 Messung des Enumerators EnVertexIncomingHalfEdge() in Methodensyntax im Kontext eines Demo Ablaufs. 49
2.8 Messung der CalcVertexNormal() Methode. 49
2.9 Messung des Enumerators EnVertexIncomingHalfEdge() in Abfragesyntax. 50
2.10 Messung des Enumerators EnVertexIncomingHalfEdge() in Abfragesyntax im Kontext eines Demo Ablaufs. 50

4.1 Klassendiagramm: Dieses UML Diagramm beschreibt die Implementierung des Projektes. 70
4.2 Aktivitätsdiagramm: Der Iterator zum Finden von eingehenden Half-Edges an einem Vertex. 71
4.3 Aktivitätsdiagramm: Die Methode zum Berechnen von Vertex Normalen 72

Literaturverzeichnis

[14t98] 14th ACM Symposium on Computational Geometry (Hrsg.): *Proceedings of the fourteenth annual symposium on Computational geometry: Proceedings of the fourteenth annual symposium on Computational geometry.* New York and NY and USA : ACM, 1998 . – ISBN 0–89791–973–4

[Amr07] Amro Khasawneh ; Code Project (www.codeproject.com) (Hrsg.): *Understanding LINQ: C#.* `http://www.codeproject.com/Articles/19154/Understanding-LINQ-C`. Version: 2007

[BCKO08] Berg, Mark de ; Cheong, Otfried ; Kreveld, Marc van ; Overmars, Mark: *Computational Geometry: Algorithms and Applications: Third Edition.* Bd. 2008. 3. Berlin Heidelberg : Springer-Verlag, 2008. – ISBN 978–3–540–77973–5

[BH07] Box, Don ; Hejlsberg, Anders: *LINQ: .NET Language Integrated Query.* `http://msdn.microsoft.com/en-us/library/bb308959.aspx`. Version: 1, 2007, Abruf: 29.07.2013

[BSBK02] Botsch, M. ; Steinberg, S. ; Bischoff, S. ; Kobbelt, L.: OpenMesh: A generic and efficient polygon mesh data structure. In: *Lehrstuhl für Informatik VIII, Computergraphik und Multimedia, RWTH Aachen* (2002)

[CGA13] CGAL: *CGAL Open Source Project: Computational Geometry Algorithms Library.* `http://www.cgal.org/`. Version: 2013, Abruf: 26.07.2013

[CKS98] Campagna, Swen ; Kobbelt, Leif ; Seidel, Hans-Peter: Directed edges: A scalable representation for triangle meshes. In: *J. Graph. Tools* 3 (1998), Nr. 4, 1–11. `http://dx.doi.org/10.1080/10867651.1998.10487494`. – DOI 10.1080/10867651.1998.10487494. – ISSN 1086–7651

[Dan11] Dan Englesson: The Half-Edge Data Structure. (2011)

[Fol01] Foley, James D.: *Computer graphics: Principles and practice.* 2. Reading and Mass. [u.a.] : Addison-Wesley, 2001. – ISBN 0–201–84840–6

[Hol12] Holmes, Ryan ; Holmes, Ryan (Hrsg.): *Graphics at Holes3D.net.* http://www.holmes3d.net/graphics/. Version: 2012, Abruf: 30.07.2013

[Kir09] Kirtan Patel ; C#Corner (Hrsg.): *Measure Execution Time of Code in C#.* http://www.c-sharpcorner.com/uploadfile/kirtan007/measure-execution-time-of-code-in-C-Sharp/. Version: 2009, Abruf: 18.01.2013

[Kob13] Kobbelt, Leif Prof. D.: *OpenMesh: A generic and efficient data structure.* http://openmesh.org/Documentation/OpenMesh-Doc-Latest/index.html,www.openmesh.org. Version: 2013, Abruf: 29.07.2013

[Kol05] Kolliopoulos, Alexander ; Kolliopoulos, Alexander (Hrsg.): *A generic Half-Edge mesh data structure for .Net.* http://www.dgp.toronto.edu/~alexk/halfedge.pdf. Version: 2005, Abruf: 30.07.2013

[Kru12] Krumke, Sven O.: *Graphentheoretische Konzepte und Algorithmen.* Wiesbaden : Vieweg+Teubner Verlag, 2012. http://dx.doi.org/10.1007/978-3-8348-2264-2. http://dx.doi.org/10.1007/978-3-8348-2264-2. – ISBN 978–3–8348–2264–2

[Küh] Kühnel, Andreas: *Visual C# 2012: Das umfassende Handbuch.* 6. Galileo Computing http://openbook.galileocomputing.de/visual_csharp_2012/. – ISBN 978–3–8362–1997–6

[McG00] McGuire, Max: *The Half-Edge Data Structure.* http://www.flipcode.com/archives/The_Half-Edge_Data_Structure.shtml. Version: 2000, Abruf: 30.07.2013

[MEW08] Marguerie, Fabrice ; Eichert, Steve ; Wooley, Jim: *LINQ in action.* 1. Greenwich and Conn : Manning, 2008 http://swbplus.bsz-bw.de/bsz278367194cov.htm. – ISBN 978–1–933988–16–0

[MHH08] Möller, Tomas ; Haines, Eric ; Hoffman, Naty: *Real-time rendering.* 3. Wellesley and Mass : A.K. Peters, 2008 http://www.gbv.de/dms/weimar/toc/562883495_toc.pdf. – ISBN 978–1–56881–424–7

[Mic13] Microsoft C#-Referenz: *http://msdn.microsoft.com*. http://msdn.microsoft.com/de-de/library. Version: 2013, Abruf: 15.07.2013

[Mül13] Müller, Christoph P.: *FUSEE: Furtwangen University Simulation and Entertainment Engine*. http://fusee3d.org/. Version: 2012/2013, Abruf: 30.07.2013

[Tit11] Tittmann, Peter: *Graphentheorie: Eine anwendungsorientierte Einführung*. München : Hanser Verlag, 2011 http://www.hanser-elibrary.com/action/showBook?doi=10.3139/9783446428539. – ISBN 978–3–446–42789–1

4 UML Diagramme

4.0.1 UML Diagramme zur Implementierung von LINQ For Geometry

Die hier angehängten UML Diagramme wurden während dieser Arbeit entwickelt und stellen die Basis der Implementierung dar. Diese Diagramme wurden vor der tatsächlichen Entwicklung der betreffenden Funktionen geschrieben und können kleine Unterschiede zum tatsächlichen Code aufweisen.

Abbildung 4.1: Klassendiagramm: Dieses UML Diagramm beschreibt die Implementierung des Projektes.

Abbildung 4.2: Aktivitätsdiagramm: Der Iterator zum Finden von eingehenden Half-Edges an einem Vertex.

Abbildung 4.3: Aktivitätsdiagramm: Die Methode zum Berechnen von Vertex Normalen