LINQ for Geometry

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

Dominik Steffen

Erstbetreuer: Prof. Christoph Müller, Fakultät DM

Zweitbetreuer: Prof. Wilhelm Walter, Fakultät DM

18. Juli 2013

Inhaltsverzeichnis

1 Einleitung

1.1 Fragestellung

Ist eine Implementierung der Half-Edge Datenstruktur in C# unter Berücksichtigung von LINQ und Lambda Support möglich?

1.2 Anforderungen und Ziele

1.3 Allgemeines zur Half-Edge Datenstruktur

Die Half-Edge Datenstruktur, oft auch als Doubly-connected edge list bezeichnet, ist eine Datenstruktur welche hauptsächlich in der Computergrafik eingesetzt wird. Sie ist sehr flexibel und kann z.B. benutzt werden um topologische und strukutrelle Informationen in Landkarten zu speichern. Diese 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 zur Speicherung die keinen großen Spielraum für Optimierungen und erweiterungen lassen. Bei Iterationen und geometrischen Manipulationen sind Face basierte Strukturen wesentlich langsamer (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 herausragenste 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. In der Implementierung dieses Projektes kommt es nicht vor, dass eine Half-Edge für sich alleine steht.

1.3.1 Die Basis der Half-Edge Datenstruktur und Unterschiede zur Doubly-connected edge list

Im vorigen Abschnitt wurde erwähnt, dass die Half-Edge Datenstruktur oft auch als Doubly-connected edge list bezeichnet wird. Im Hinblick auf die Implementierung dieser Arbeit ist das so nicht ganz richtig. Dieser Text bezieht sich bei Betrachtung der Doubly-connected edge list immer auf die Darstellung von [1]. Es gibt einige wenige feine Unterschiede der beiden Datenstrukturen die sich in der Implementierung dieses Projekts zu einem späteren Zeitpunkt, nicht übermässig, bemerkbar machen.
Die Half-Edge Datenstruktur besteht grundsätzlich aus vier verschiedenen Datensätzen (data records). Dazu zählen:

- Vertex, Punkt im dreidimensionalen Raum der um zusätzliche Informationen erweitert wurde.
- Face, zu betrachten als Polygon (Geometrische Figur mit mehr als 3 Eckpunkten) mit zusätzlichen gespeicherten 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 benutzen 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 Datens"atzen zu speichern. Diese zusätzlichen Informationen werden im folgenden als Attributs-Informationen bezeichnet. [1, S. 31]

Zu beachten ist hierbei, dass das wichtigste Element der Datenstruktur, die Half-Edge, als gerichtete Kante betrachtet werden sollte.

> In der Half-Edge Datenstruktur sind die Half-Edges Kanten mit einer Orientierung. Der Vertex dem sie entspringen ist dabei als ihr Ursprung zu bezeichnen, während der Vertex auf den sie zeigen und auf den sie einen Zeiger enthalten, als Ziel betrachtet wird. [1, S. 31]

> „Because half-edges are oriented we can speak of the *origin* and the *destination* of a half-edge.“ [1, S. 31]

Doubly Connected Edge List

Die doubly connected edge list (kurz DECL) wie in [1] 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 Datens"atze. [1, 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
- Zeiger auf die nächste Half-Edge im Uhrzeigersinn (Ist variabel implementierbar. LINQ For Geometry implementiert die Zeiger im Uhrzeigersinn)
- Zeiger auf den Vertex auf den die Half-Edge zeigt
- Zeiger auf das Face zu welchem die Half-Edge gehört

speichert die Doubly-connected edge list 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 auf den die Half-Edge zeigt
- Zeiger auf den Vertex von dem die Half-Edge ausgeht
- Zeiger auf das Face zu welchem die Half-Edge gehört

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 Namensgebung der Doubly-connected edge list und der Half-Edge Datenstruktur wird oft synonym 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 [1, 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.

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 Zeiger jeder der data record enthält und eine kurze Skizze zu einem Polygon 1.1 und den Beziehungen der darin enthaltenen data records wie sie eine Implementierung der Half-Edge Datenstruktur darstellen würde.
Hier noch einmal eine Auflistung der einzelnen data records die in der Half-Edge Datenstruktur existieren abnehmend sortiert nach ihrem Informationsgehalt.

- 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. Die folgenden 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 Face zu dem sie gehört
- 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 noch eine Information

- Ein Zeiger auf eine Half-Edge die es begrenzt

Der Vertex ebenso eine Information

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

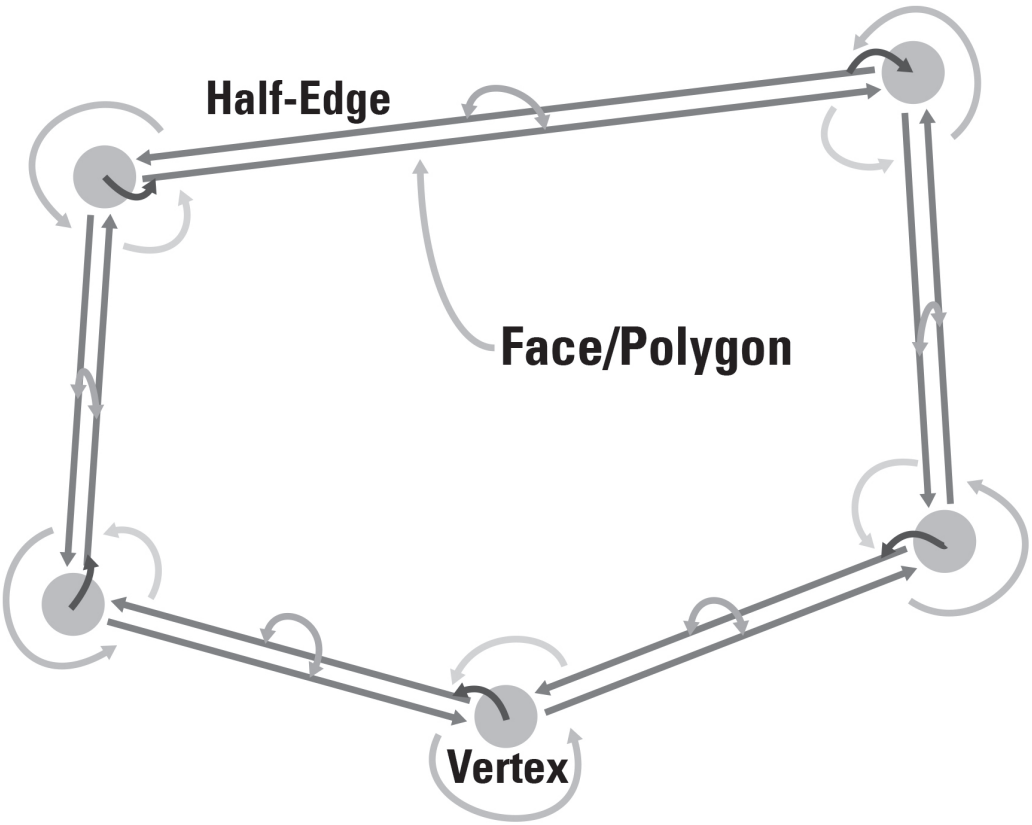

Abbildung 1.1: Diese Illustration zeigt die Verbindung der data records an einem beliebigen Polygon durch Pfeilverbindungen.

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.

1.3.2 Vorteile der Half-Edge Datenstruktur

1.3.3 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 das System für eine Implementierung so zu konstrurieren dass es alle Anforderungen an eine 3D Echtzeitdatenstruktur erfüllt. Vor einer Implementierung der Half-Edge Da-

tenstruktur sollte also gut überlegt sein ob sie für das aufkommende Probleme eine vernünftige Lösung darstellt. In LINQ For Geometry ist die Datenstruktur eine sehr eleganze Lösung. Sie ist hinreichend flexibel und gerade noch maximal komplex um eine mögliche Lösung für die Problemstellung „(Editierbare) Echtzeit 3D Datenstruktur“ darzustellen. Es sollte also darauf geachtet werden die Half-Edge Datenstruktur nicht als Universallösung zu betrachten. Wenn ich nur einige wenige Algorithmen oder 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 möchte diese Arbeit ein kleines Beispiel vorweisen und exemplarisch den Speicherverbrauch der Half-Edge Datenstruktur bestimmen.

Speicherverbrauch im Gegensatz zu Face basierten Lösungen

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 oder einige wenige Operationen nur einmal ausführen wollen so wäre es wohl ratsam eine andere Datenstruktur dafür zu suchen. Allerdings ist es heute in der Computergrafik meistens so, dass wir genügend Speicherplatz im Primärspeicher unserer Rechner und Grafikkarten zur verfügung haben um diesen Umstanz zu verschmerzen. Hier eine Beispielhafte Berechung des Datenverbrauchs.
Man gehe davon aus jedes gespeicherte Element (Daten in Form von Vertices, Faces, Edges, Half-Edges) wird mit 32bit repräsentiert und jeder Zeiger (Pointer) wird mit 8bit repräsentiert dann könnte man den Speicherverbrauch eines einfachen sechs Seitigen Würfels wie folgt beschreiben.
So stellt dieHalf-Edge Datenstruktur einen Würfel dar:

- 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 um das ca. 4 fache höher liegt als der Verbrauch einer Face basierten Lösung. Wie die Half-Edge Datenstruktur das aber wieder ausgleicht ist dem Abschnitt „Vorteile der HES“ zu entnehmen.

1.4 Aktueller Forschungsstatus

1.4.1 Probleme der aktuellen Forschung

1.5 Einführung zu LINQ in C#

Language Integrated Query, kurz LINQ und zu deutsch Sprachintegrierte Abfrage, funktioniert ab C# 3.0 und ist ein Feature von Microsofts .NET Paket. LINQ ist also ein Produkt von Microsoft und erinnert auf den ersten Blick in seinem syntaktischen und semantischem Aufbau stark an das Open Source Projekt MySQL welches aktuell (Stand Juli 2013) von der Oracle Corporation betreut wird. Beide Projekte unterscheiden sich jedoch wesentlich. LINQ ist in erster Linie ein Feature, dass direkt auf Objekten arbeitet wohingegen MySQL auf relationale Daten angewendet wird. Allerdings kann LINQ viel mehr als das.

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 auszulesen. Es kann aber auch dazu verwendet werden mit XML Datensätzen bzw. Dokumenten zu arbeiten ist aber auch 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 Daten. Weitere sind LINQ to DataSet und LINQ to SharePoint auf die hier mangels Relevanz zum Projekt nicht weiter eingegangen wird.
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. Ein weiterer nicht zu verachtender Vorteil von LINQ im Gegensatz zu den meisten Implementationen von SQL Statements ist, dass die Statemtents vom C# compiler zum Zeitpunkt der Kompilierung auf syntaktische Korrektheit geprüft werden und der Benutzer vom IntelliSense System von Visual Studio wertvolle Hinweise während der Generierung von LINQ Statements erhält.

1.5.1 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 die 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 nch 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 variable und nicht die Collection der Daten.
Hier folgt ein Beispiel welches eine einfache LINQ Abfrage auf einen einfachen Array Datensatz der IEnumerable<T> implementiert darstellt.

Listing 1.1: 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 Statement kann jedoch auch erzwungen direkt ausgeführt werden. Durch einen direkten Aufruf der Methoden toList<T> oder toArray<T> auf dem Statement.

Listing 1.2: LINQdirectQueryList.cs - Einfaches direktes LINQ Abfrage Beispiel [2]

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

1.5.2 Abfragesyntax in LINQ

Was im vorigen Absatz als LINQ Statement bezeichnet wurde ist ein LINQ Statement geschrieben in der Abfragesyntax. Diese Syntax zeichnet sich durch einfache Lesbarkeit und einfache Zugänglichkeit aus. Zur Kompilierzeit wird diese Syntax vom Compiler in die so genannte Methodensyntax übersetzt. Beide Methoden sind Semantisch identisch auch wenn sie sich auf den ersten Blick Syntaktisch sehr unterscheiden.

1.5.3 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) [2] 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.

Listing 1.3: LINQzweiSynt.cs - Zwei syntaktische Möglichkeiten [2]

```
1 //Query syntax:
2 IEnumerable<int> numQuery1 =
3     from num in numbers
4     where num % 2 == 0
5     orderby num
6         select num;
7
8 //Method syntax:
9 IEnumerable<int> numQuery2 = numbers.Where(num => num % 2 == 0).OrderBy(n => n
      );
```

1.6 Einführung zu Lambda in C#

Lambda Ausdrücke in C# sind Anonymen Funktionen sehr ähnlich. In dieser Arbeit wurden Lambda Ausdrücke hauptsächlich in LINQ Statemens 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.

1.6.1 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() wenden 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 1.4

Listing 1.4: LambdaEasy.cs - Einfacher Lambda Ausdruck

```
1 // Creating a collection.
2 int[] numbers = new int[5] {0, 1, 2, 3, 4};
3
4 // Executing a LINQ Statement in method form with a lambda expression.
5 // This expression will return every number in numbers multiplied by itself.
6 int[] newNumbers = numbers.Select(x => x * x);
```

2 Hauptteil

2.1 Gegenüberstellung nativer (OpenMesh.org) und gemanagter Implementierungen der Half-Edge Data Structure

2.2 Geschwindigkeitsunterschiede von nativem und C# Code

2.3 Die Vorteile in der Entwicklung von managed Code

2.4 Das Software Projekt LINQ For Geometry (LFG)

2.4.1 Konzept zu LINQ For Geometry

Der Link von LINQ zur Half-Edge Datenstruktur

2.4.2 Warum die Programmiersprache C# ?

2.4.3 Furtwangen University Simulation and Entertainment Engine (FUSEE)

2.5 Der Import von Geometriedaten im „Wavefront Object“ Format

2.5.1 Warum das Wavefront Object Format

2.5.2 Face basierter Import - Edge basiertes Handling

2.5.3 Importer für das Wavefront Format

2.6 Initialisierungsablauf

2.6.1 UML Diagramme zum Initialisierungslauf

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

2.7.1 Beispiel eines Handler Konstruktes und seiner Implementierung

Die Handler Konstrukte „Structs“ in dieser Arbeit, welcher Art auch immer

tet werden.

Listing 2.1: HandleHalf-Edge.cs - Variablen Deklaration des „Zeigers“

```
1        internal int _DataIndex;
```

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 Konstruktuen. Ein Handler Struct stellt eine implizite Konvertierung des Handlers, siehe Listing 2.2, in den Datentypen Integer (int) zur Verfügung.

Listing 2.2: 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 2.3 betrachten. 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.

Listing 2.3: HandleHalf-Edge.cs - Is Valid?

```
1        public bool isValid
2        {
3            get { return _DataIndex >= 0; }
4        }
```

Eine Besonderheit der Handler ist die mit „internal“ gekennzeichnete Deklaration der Indizes. Siehe hierzu Listing 2.4. 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.

Listing 2.4: HandleHalf-Edge.cs - Deklarationen als internal

```
1        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." [2]

Die vollständige Implementierung dieses Structs und aller sieben weiteren kann im Visual Studio 2010 Projekt auf dem diese Arbeit aufbaut unter folgender Verzeichnisstruktur betrachtet werden. „LinqForGeometry/LinqForGeometry.Core/src/Handles“

2.8 Pointer Container und ihre Rolle 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 des ganzen Programms. 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.

Listing 2.5: PtrContExample1.cs - Manipulation eines Pointer Containers

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

```
3
4  ...
5  // We are building up a Half-Edge Structure and do interesting things ...
6  ...
7
8  // Now we want to manipulate a specific Pointer Container.
9  // So we first need a Handle to a Pointer Container.
10 // We retrieve "hep" by selecting the last element in the Pointer Container
       List. (Does not make any special sense it's just for the example)
11 HandleHalfEdge heH = new HandleHalfEdge(_LhePointerContainers.Count() - 1);
12 HalfEdgePtrCont hep = _LhePointerContainers[heH];
13
14 // We now can savely remove it from the List
15 _LhedgePtrCont.RemoveAt(heH);
16
17 // Let us do whatever we want for example manipulating the face pointer
18 // We are setting the face pointer to invalid.
19 hep._f = new HandleFace(-1);
20
21 // Now we can savely insert the changed Container in the List again
22 _LhedgePtrCont.Insert(heH, hep);
```

2.8.1 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 in FUSEE 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 speichernzu 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.

2.8.2 Edges Pointer-Container

Ein Edge Pointer Container, in LINQ For Geometryzu finden im Struct „EdgePtrcont.cs“, ist ein 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 Erste 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 manipulations Algorithmen auf das gespeicherte Mesh anzuwenden.

2.8.3 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 servePrinzip verteilt. Das bedeutet, dass nur die erste Half-Edge die während des Initialisierungslauf an den Vertex angelegt wird an dieser Stelle auch referenziert wird. Durch diesen Handle können Iteratoren sehr einfach realisiert werden die 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 eingehen-

den 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 Ergebnis Set 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 möglich.

2.8.4 Faces Pointer-Container

Laut Definition der Half-Edge Datenstruktur speichert ein Face jeweils einen Zeiger auf eine ihm angehörige Half-Edge. In LINQ For Geometrywurde genau das in der Implementierung umgesetzt. Das erleichtert den Einstieg in manipulations Algorithmen 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 Geometryund seiner Implementierung in der FUSEE Engine wird die Face Normale aktuell nur zur Berechnung der

Vertex Normalen herangezogen. FUSEE benötigt für das Backface Culling lediglich eine Traversierung der Polygone entgegen des Uhrzeigersinns. Diese Besonderheit tritt in Engines auf welche OpenGL als Grafikschnittstelle verwenden.

- HandleFaceNormal _fn;

2.9 Die Geometrie in LINQ For Geometry

Hier Information über das Geometry Objekt und die darin enthaltenen Daten etc.

3 Schluss

3.1 Ergebnis der Arbeit

3.1.1 Der aktuelle Status von LINQ For Geometry

Welche Möglichkeiten zur Erweiterung durch eigenen Code bietet LINQ For Geometry

3.2 Zukünftige Entwicklungen und Ausblick auf die Verwendung von LINQ For Geometry

Appendix

Listings

Literaturverzeichnis

[1] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications: Third Edition*, volume 2008. Springer-Verlag, Berlin Heidelberg, 3 edition, 2008.

[2] Microsoft C#-Referenz. http://msdn.microsoft.com, 2013.