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

16. Juli 2013

Inhaltsverzeichnis

1 Einleitung

1.1 Fragestellung

Ist es möglich die „Half-Edge Data Structure“ (kurz HES) in einer gemanagten Programmiersprache wie C# unter der Berücksichtigung von LINQ und Lambda Ausdrücken so zu implementueren, dass damit grundlegende Geometriemanipulation in der Computergrafik erfolgen kann?

1.2 Anforderungen und Ziele

1.3 Die Half-Edge Data Structure (kurz HES) als Algorithmus

1.3.1 Die Basis der HES

Verbindungen und Beziehungen in der HES

Doubly Connected Edge List

Die doubly connected edge list (kurz DECL) ist der verwendeten HES nicht unähnlich. Beide Datenstrukturen ähneln sich immens und haben doch ein paar wenige entscheidende Unterschiede. TODO

1.3.2 Speicherverbrauch im Gegensatz zu Face basierten Lösungen

1.3.3 Vorteile der HES

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 Kor-

rektheit 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

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

```
3
4  // This is the statement
5  var numQuery =
6      from num in numbers
7      where (num % 2) == 0
8      select num;
9
10 // Here the query gets executed.
11 foreach (int num in numQuery)
12 {
13     Console.Write(num);
14 }
```

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 [1]

```
1  List<int> numQuery2 =
2      (from num in numbers
3       where (num % 2) == 0
4       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) [1] 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 [1]

```
//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);
```

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

```
// 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 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 Warum die Programmiersprache C# ?

2.4.2 Furtwangen University Simulation and Entertainment Engine (FUSEE)

2.4.3 Grober Ablauf einer HES Initialisierung

UML Diagramme zum Initialisierungslauf

2.5 Der Import von Geometriedaten im „Wavefront Object“ Format

2.5.1 Warum das Wavefront Object Format

Face basierter Import - Edge basiertes Handling

2.5.2 Importer für das Wavefront Format

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

2.6.1 Beispiel eines Handler Konstruktes und seiner Implementierung

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 Konstrukturen. 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." [1]

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"

2.7 Pointer Container und ihre Rolle in LINQ For Geometry

2.7.1 Half-Edges Pointer-Container

Vertex Normal Handler

Vertex UV Handler

2.7.2 Edges Pointer-Container

2.7.3 Vertices Pointer-Container

2.7.4 Faces Pointer-Container

2.8 Der Geometry Teil in LINQ For Geometry

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

2.8.1 Benchmarks zu Laufzeiten des Programms

2.9 Anwendungsfälle von LINQ For Geometry

2.9.1 LINQ For Geometry als Editor Datenstruktur in FUSEE

2.10 LINQ und Lambda Ausdrücke und ihre Stärken und Schwächen bei der Selektierung großer Datenmengen

2.11 Stern- und Umlaufenumeratoren (Iteratoren)

2.11.1 Die Geschwindigkeit der Half-Edge Datenstruktur in den Enumeratoren

2.11.2 Verwendete „Design-Patterns“ und Softwarelösungen

2.11.3 LINQ und Lambda Ausdrücke in den Enumeratoren

2.11.4 Unterschiedliche Iteratoren kurz dargestellt

2.12 Manipulation von Mesh Daten in der Datenstruktur

2.12.1 Manipulation von Vertices

2.12.2 Manipulation von Edges und Half-Edges

2.12.3 Manipulation von Faces

2.12.4 Beispielhafte Implementierung von Standard Algorithmen zur Geometriemanipulation als Modul

2.12.5 Implementierung von Catmull Clark als Modul für LINQ For Geometry

Was ist der Catmull Clark Algorithmus?

Vorteile der Implementierung in der HES

3 Schluss

3.1 Ergebnis der Arbeit

3.1.1 Wie weit ist LINQ For Geometry fortgeschritten

3.1.2 Welche Möglichkeiten zur Erweiterung durch eigenen Code bietet LINQ For Geometry

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

Listings

Literaturverzeichnis

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