Praxisprojekt

Half-Edge Mesh für Unity3D

Erstellt von:
Yannick Dittmar
Studiengang: Allgemeine Informatik
Matrikelnummer: 11117676

Datum der Abgabe: xx.xx.xxxx

Betreuung: Dennis Buderus

Technische Hochschule Köln
Fakultät für Informatik und Ingenieurswissenschaften

Inhaltsverzeichnis

1 Grundlagen

Dieses Kapitel beschäftigt sich mit den Grundlagen, die für diese Arbeit relevant sind. Dabei geht es sowohl um die Entwicklungsumgebung Unity3D, als auch um das Konzept der Triangluar Meshes, die essenziell in der 3D-Computergrafik sind.

1.1 Unity3D

Unity3D ist eine plattformübergreifende Spiele-Engine mit eingebauter Entwicklungsumgebung für Zwei- und Dreidimensionale, sowie Augmented- und Virtual-Reality Spiele

und Simulationen. Die Engine besitzt einen eigenen Editor, in dem diverse Szenarien erstellt und bearbeitet werden können. Um diese Szenarien zum Leben zu bringen unterstützt Unity selbst programmierte Scripte auf der Grundlage von C#. Insgesamt laufen auf über drei Milliarden Geräten Programme, die mit Unity erstellt wurden, auf über 25 verschiedenen Plattformen, wie Windows oder Linux und den gängigen Spielekonsolen wie die PlayStation 4, XBox One und die Nintendo Switch. Zudem sind 50% aller mobilen Spiele und 60% aller VR/AR Anwendungen mithilfe von Unity entstanden [**?**]. Spiele wie ,,Pokémon Go", ,,Superhot" und das Simulationsspiel ,,Universe Sandbox" sind drei Beispiele für Spiele, die in Unity entstanden sind.

1.2 Triangular Meshes

Um auf einem Computer eine dreidimensionale Szene, zum Beispiel mit der Hilfe von Unity darzustellen, müssen alle dargestellten Modelle angenähert werden. In der Regel werden dafür Dreiecksnetze (Triangular Mesh) verwendet, die die Oberfläche eines Objekts mithilfe von Dreiecken annähert 1. Das Mindeste an Informationen die für ein solches Dreiecksnetz benötigt werden, ist ein Reihe von Eckpunkten, den Vertices, die immer in Dreiermengen kommen und deren Position im dreidimensionalen Raum [**?**, S.262]. Diese werden durch Kanten (Edges) verbunden und bilden damit Dreiecksflächen, auch Faces genannt.

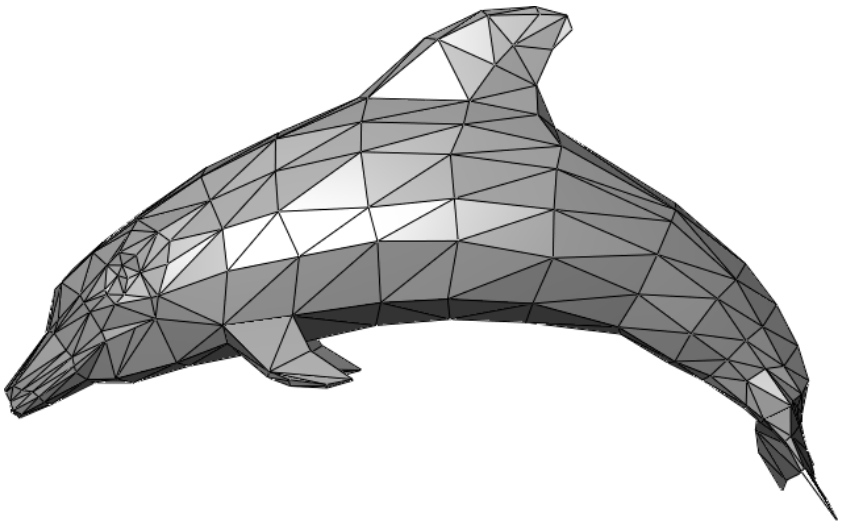

Abbildung 1: Beispiel eines Dreiecks-Polygonen-Netz, von [**?**]

Die einfachste Implementierung eines solchen Meshes sieht vor, für jedes Dreieck drei Punkte mit einer X-, Y- und Z-Koordinate zu speichern. Wichtig ist, dass die Orientierung der Dreiecke immer gleich bleibt, also dass im gesamte Netz die Punkte aller Dreiecke im oder gegen den Uhrzeigersinn angegeben sind. Ein Nachteil von dieser Art, ein Mesh zu speichern ist, dass Punkte die häufig im Netz vorkommen mehrfach gespeichert werden.

Aus diesem Grund kann eine abgewandelte Version davon verwendet werden, ein Indexed Mesh. Ein solches Mesh trennt die Vertices von der Verwendung im Mesh. Dafür werden zwei Listen verwendet, eine mit den Punkten und eine Liste mit den Indices, wie die Dreiecke zu konstruieren sind. Ein Index zeigt auf eine Position in der Vertex-Liste und drei Indices formen jeweils eine Face.

1.3 Triangular Meshes in Unity

Unity bietet die Möglichkeit, mit Hilfe von selbstgeschriebenen Scripten eigene Indexed Meshes zu erstellen. Dafür stellt Unity ein eigenes Mesh-System zur Verfügung, die *UnityEngine.Mesh*-Klasse. Damit diese ein Mesh rendern kann, erwartet das Mesh zum Einen ein *UnityEngine.Vector3*-Array für die Vertices, wobei ein *Vector3* ein Punkt im dreidimensionalen Raum darstellt. Zum Anderen erwartet es ein *int*-Array, welches die mit den Indices der Vertices die Reihenfolge der Dreiecke festlegt. Zu beachten ist, dass die Orientierung eines Unity-Meshs immer im Uhrzeigersinn ist, im Gegensatz zu Anwendungen wie Blender, die gegen den Uhrzeigersinn arbeiten. Der folgende Code zeigt beispielhaft, wie ein Unity-Mesh erzeugt werden kann:

```
1  public void CreateMesh()
2  {
3    //--- Der Vollstaendigkeit halber vorhanden
4    meshFilter = gameObject.GetComponent<MeshFilter>();
5    if (meshFilter == null)
6    meshFilter = gameObject.AddComponent<MeshFilter>();
7
8    //--- vom MeshFilter zum Mesh
9    mesh = meshFilter.sharedMesh;
10   if (mesh == null)
11     mesh = new Mesh { name = "Quad" };
12
13   //--- MeshRenderer holen
14   meshRenderer = this.gameObject.GetComponent<MeshRenderer>();
15   if (meshRenderer == null)
16     meshRenderer = gameObject.AddComponent<MeshRenderer>();
17
18   //--- Mesh zusammenstellen
19   //--- Vertices/Points
20   Vector3 P0 = new Vector3(0, 0, 0);
21   Vector3 P1 = new Vector3(0, 1, 0);
22   Vector3 P2 = new Vector3(1, 0, 0);
23   Vector3 P3 = new Vector3(1, 1, 0);
24
25   List<Vector3> verticies = new List<Vector3> { P0, P1, P2, P3 };
26
27   //--- Triangles
```

```
28      List<int> triangles = new List<int>();
29
30      triangles.Add(0);
31      triangles.Add(1);
32      triangles.Add(2);
33      triangles.Add(2);
34      triangles.Add(1);
35      triangles.Add(3);
36
37      //--- Mesh befuellen
38      mesh.Clear();
39      //--- Vertices zuweisen
40      mesh.vertices = verticies.ToArray();
41      //--- Triangles zuweisen
42      mesh.triangles = triangles.ToArray();
43      //--- Mesh dem MeshFilter zuweisen
44      meshFilter.sharedMesh = mesh;
45  }
```

Und liefert folgendes Ergebnis:

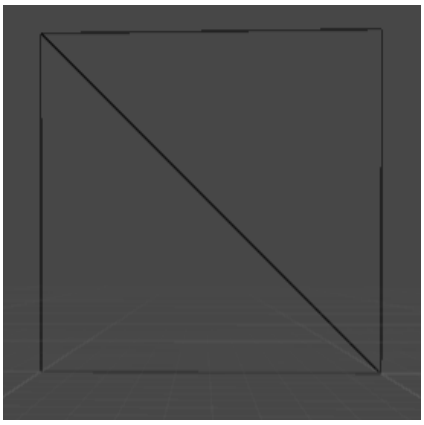

Abbildung 2: Die Wireframeansicht des erstellten Meshes im Unity Editor

1.4 Datenstrukturen für Meshes

Wird zur Laufzeit eine Nachbarschaftsbeziehung abgefragt, zum Beispiel welche Faces und Kanten an einem Punkt anliegen oder welche Endpunkte eine Edge hat, stößt ein Indexed Mesh schnell an seine Grenzen. Diese Informationen lassen sich für ein solches Mesh über eine erschöpfende Suche ermitteln, dessen Laufzeit von der größe des Meshes abhängig ist. Um diese Probleme zu lösen gibt es unterschiedliche Datenstrukturen, die einzelne Komponenten eines Meshes direkt in eine Beziehung zueinander setzen.

1.4.1 Triangle-Neighbor Structure

Ein solcher Ansatzt ist die Triangle-Neighbor Structure (Dreiecks-Nachbar Struktur). Dabei erhält jedes Dreieck eine Referenz auf jeden seiner Nachbarn und jeder Vertex speichert zusätzlich noch einen Pointer auf ein benachbartes Dreieck. Eine beispielhafte Implementierung sieht wie folgt aus [?, S.269]:

```
class Triangle
{
  Triangle[3] Neighbours;
  Vertex[3] Vertices;
}

class Vertex
{
  // --- Vertex specific data
  Triangle Triangle;
}
```

Der Vorteil zu einem Indexed Mesh ist, dass nun nicht mehr jeder Punkt betrachtet werden muss, sondern mithilfe der Faces Aussagen über das Mesh getroffen werden können. Ein Problem, was beim Traversieren des Meshes auftritt ist, dass nach jedem Schritt geprüft werden muss, ob die nächste Face nicht gleichzeitig auch die letzte Face war. Damit dieses Problem nicht auftritt, kann von einem facebasierten Ansatz auf einen Edgebasierten umgestellt werden.

1.4.2 Winged-Edge Mesh

Eine edgebasierte Datenstruktur ist das Winged-Edge Mesh. Diese Datenstruktur besteht aus Edges, Faces und Vertices. Jede Face und jeder Vertex verweist auf eine anliegende Kante. Zudem besitzt jede Kante eine Referenz auf ihren Start- und Endpunkt (Head und Tail), die beiden anliegenden Faces sowie die beiden vorherigen und nachfolgenden Edges. Aus dieser Beschreibung ergibt sich eine solche Implementierung [?, S.273]:

```
class Edge
{
  Edge LeftPrevious, RightPrevious, LeftNext, RightNext;
  Vertex Head, Tail;
  Face Left, Right;
}

class Face
{
  // --- Face specific data
  Edge Edge;
}
```

```
13
14 class Vertex
15 {
16   // ---- Vertex specific data
17   Edge Edge;
18 }
```

Die Vorteile über der Triangle-Neighbor Structure sind, dass nicht nur Zugriffe zwischen Faces und Vertices konstant sind, sondern auch Abrufe von Kanten auf Dreiecke und Punkte vice versa sind in konstanter Zeit möglich. Die Herausforderung beim Arbeiten mit einem Winged-Edge Mesh ist, dass immer drauf geachtet werden, aus welcher Richtung die aktuelle Edge kommt, um in der richtigen Richtung weiterzuarbeiten. Dieses Unannehmlichkeit wird im Half-Edge Mesh gelöst.

2 Half-Edge Mesh

Um ein Traversieren eines Meshes, Abfragen der Nachbarschaftsbeziehungen der einzelnen Meshkomponenten und Operationen wie „Subdivisionen" von Faces (Unterteilung der Faces in kleinere Dreiecke) so einfach wie möglich zu machen, gibt es neben den oben genannten Ansätzen noch den Ansatz der Half-Edge Meshes. Ein solches Mesh besteht aus folgenden Komponenten:

- eine Liste von Vertices
- eine Liste von Half-Edges
- eine Liste von Faces.

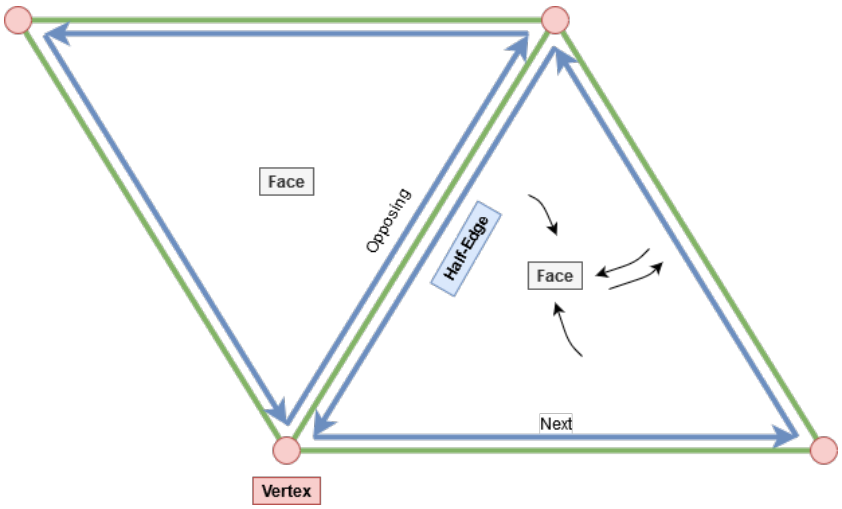

Abbildung 3: Die Elemente eines Half-Edge-Mesh.

Im Gegensatz zu den Winged-Edge Meshes modellieren die Half-Edge Meshes eine Kante nicht explizit, sondern als Kombination aus zwei Half-Edges, die in jeweils entgegengesetzte Richtungen auf einen der Endpunkte der Kante zeigen, die Half-Edges sind gegen den Uhrzeigersinn orientiert. Durch die Aufteilung der Kanten kann jede Half-Edge genau einer Face zugeordnet werden und besitzt somit nicht mehr eine Referenz auf den linken und rechten Nachfolger, sondern einen nur noch einen Pointer auf die nächste Half-Edge der Face, wie in Abbildung 5 gezeigt. Zudem Besitzt jede Half-Edge einen Verweis auf die zugehörige Face sowie auf die Gegenüberliegende Half-Edge. Ein Verweis auf die Vorherige ist nicht nötig, da diese die übernächste Kante ist.

2.1 Vergleich der Datenstrukturen

Jede der vorgestellten Datenstrukturen kommt mit Vor- und Nachteilen. Die Entscheidung, welche von diesen verwendet wird, ist vom Use Case der Anwendung abhängig. Vergleichen kann man die Datenstrukturen in Sachen Speichernutzung und der Laufzeit von Nachbarschaftsabfragen. Bei der Speichernutzung wird ein allgemeines Mesh mit n_v Vertices betrachtet, bei der Laufzeitanalyse wird zusätzlich mit n_t die Anzahl der Dreiecke und mit m_{ev} die Anzahl der Kanten pro Vertex dargestellt.

Tabelle 1: Vergleich der Datenstrukturen

	Indexed Mesh	TNS	WEM	HEM
Relativer Speicherbedarf	$36 \times n_v Byte$	$116 \times n_v Byte$	$228 \times n_v Byte$	$228 \times n_v Byte$
Laufzeitanalyse der möglichen Abfragen:				
Edge → Vertices	N/A	N/A	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Edge → Faces	N/A	N/A	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Edge → angrenzende Edges	N/A	N/A	$\mathcal{O}(n_{ev})$	$\mathcal{O}(n_{ev})$
Vertex → Edges	$\mathcal{O}(n_t)$	$\mathcal{O}(n_{ev})$	$\mathcal{O}(n_{ev})$	$\mathcal{O}(n_{ev})$
Vertex → Faces	$\mathcal{O}(n_t)$	$\mathcal{O}(n_{ev})$	$\mathcal{O}(n_{ev})$	$\mathcal{O}(n_{ev})$
Face → Edges	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Face → Vertices	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Face → angrenzende Faces	$\mathcal{O}(n_t)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$

Auffällig ist, dass der Speicherbedarf mit der Komplexität des Netzes wächst, wodurch einzelne Abfragen beschleunigt oder ermöglicht werden. Zusätzlich fällt auf, dass das Winged-Edge Mesh gleich wie das Half-Edge Mesh abschneidet, allerdings bietet das Half-Edge Mesh in der Handhabung große Erleichterungen, durch eine eindeutige Zuordnung von Half-Edge zu Face.

2.2 Implementierung der Komponenten

Die oben beschriebenen Komponenten sind im Folgenden in C# implementiert, um das Half-Edge Mesh in Unity3D verwenden zu können.

2.2.1 Klassenstruktur

Aus den beschriebenen Elementen einer Half-Edge-Netzstruktur ergibt sich das in Abbildung 6 gezeigte Klassendiagramm. Der grundlegende Aufbau der einzelnen Klassen basiert dabei auf dem Plankton-Mesh [WP17]. Jede Komponente besitzt eine eigene Listenklasse. Allerdings besitzen die einzelnen Komponenten im Plankton-Mesh keine direkte Referenz auf die benachbarten Komponenten, sondern einen Verweis auf den Index der Elemente in der jeweiligen Liste. So hat eine Half-Edge keine weitere Half-Edge als ,,Next", sondern ein den jeweiligen Index der nächsten Kante.

2.2.2 Die Vertex, HalfEdge und Face Klassen

Die Klassen *Vertex, HalfEdge* und *Face* sind die Datenmodelle der oben beschriebenen Komponenten. Die Vertex-Klasse besitzt einen *Vector3* Point, der die Position im Raum darstellt, eine *HalfEdge*, die von diesem Punkt aus geht (Im Gegensatz zu [WP17] wird hier eine Referenz gespeichert.) und der Index des Punktes, um die Arbeit mit dem Unity-Mesh zu erleichtern. Zudem kann ein *PositionChangedEvent* abonniert werden, um Positionsänderungen im Unity-Mesh direkt zu zeigen.
Eine HalfEdge besitzt die oben erwähnten Eingenschaften: den Punkt von dem sie ausgeht, die anliegende Face, die gegenüberliegende und nächste HalfEdge sowie den Index der HalfEdge. Wie auch der Index der Vertices, ist dieser Index für das Unity-Mesh wichtig.
Die Face-Klasse besitzt eine Referenz auf eine anliegende HalfEdge und den Index der Face, diese wieder für die Arbeit mit dem Unity-Mesh.

2.2.3 Listenklassen

Objekte der Klassen *Vertex, HalfEdge, Face* werden jeweils in einer Listenklasse gespeichert. Die Klassen *VertexList, HalfEdgeList, FaceList* implementieren das *IEnumerable*-Interface, um die Iteration über die Elemente zu vereinfachen. Zudem beinhalten diese Klassen die Kernlogiken für die Komponenten. VertexList bietet die Möglichkeit, einen neuen Vertex hinzuzufügen oder einen zu entfernen, die HalfEdgeList verfügt über eine *CreateHalfEdge*-Methode, die wie folgt eine HalfEdge anlegt:

```
1
2 public HalfEdge CreateHalfEdge(Vertex vertex, Face face, HalfEdge next)
3 {
4   HalfEdge halfEdge = new HalfEdge(vertex, face, next, Count);
5   vertex.HalfEdge = halfEdge;
6   _halfEdges.Add(halfEdge); // ---- _halfEdges ist die zugrunde liegende Liste
7   return halfEdge;
8 }
```

Und auch die FaceList besitzt eine Create-Methode, um eine Fläche korrekt anlegen zu können. Dabei wird die Referenz der Face auf die HalfEdge gesetzt und umgekehrt.

Eine Weitere wichtige Methode ist die *GetFaceCirculator*-Methode, die eine Liste aller HalfEdges, die an einer gegebenen Face anliegen, zurück gibt.

2.3 Erstellen eines Half-Edge-Meshes

Die Oben genannten Listenklassen werden von der *HalfEdgeMesh*-Klasse verwendet, um aus der beschriebenen Half-Edge-Datenstruktur ein von Unity renderbares Mesh zu erstellen.
Um ein Dreieck, die einfachste mögliche Netzstruktur zu erzeugen, kann die Methode *CreateMesh* verwendet werden. Diese erstellt die drei Eckpunkte des Dreiecks, verbindet zwei mit einer neuen Half-Edge und erzeugt damit ein Face. Die Fläche wird dann verwendet um die fehlenden Kanten mit Referenzen zu erzeugen und anschließend wird die Referenz der ersten Half-Edge auf die zweite gesetzt. Zum Schluss wird *GenerateUnityMesh* aufgerufen um mit den angelegten Daten das UnityMesh zu generieren.

```
1
2  public void CreateMesh(Vector3 va, Vector3 vb, Vector3 vc)
3  {
4    Vertex a = Vertices.CreateVertex(va);
5    Vertex b = Vertices.CreateVertex(vb);
6    Vertex c = Vertices.CreateVertex(vc);
7
8    HalfEdge heA = HalfEdges.CreateHalfEdge(a, null, null);
9    Face face = Faces.CreateFace(heA);
10   HalfEdge heB = HalfEdges.CreateHalfEdge(b, face, heA);
11   HalfEdge heC = HalfEdges.CreateHalfEdge(c, face, heB);
12   heA.Next = heC;
13
14   GenerateUnityMesh();
15 }
```

Sollen beim erstellen des Netzes weitere Flächen hinzugefügt werden, ist es möglich diese Methode zu erweitern oder weitere Faces mit *AddFace* hinzuzufügen.

2.3.1 GenerateUnityMesh

Um aus den Daten des Half-Edge-Mesh ein für Unity brauchbares Mesh zu generieren, müssen folgende Daten aus dem Half-Edge-Mesh entnommen werden: Die Position jedes Punktes, als Liste und ein Array mit der Reihenfolge, wie diese Punkte zu verbinden sind. Die Zusammenstellung dieser Daten passiert mit Hilfe von Linq. Hier der wichtigste Teil der *GenerateUnityMesh*-Methode.

```
1    ClearMesh();
2    // --- Add vertices
3    List<Vertex> vertices = Vertices.Select(p => p.Point).ToList();
4
5    // --- Add triangles
```

```
6     foreach (Face face in Faces)
7     {
8       HalfEdge adjacentHalfEdges = Faces.GetFaceCirculator(face).ToList();
9       SetMeshTriangles(face.Index, adjacentHalfEdges
10      .Select(p => p.OutgoingPoint.Index).ToList(), true);
11    }
12
13    AddMeshVertices(vertices);
14    CommitMeshTriangles();
```

Da das Netz eines komplexen Modells sehr groß werden kann, ist es für die Laufzeit von Vorteil, wenn bei lokalen Änderungen nicht das gesamte Netz neu generiert werden muss, wobei jedes Mal über alle Punkte, Kanten und Flächen iteriert werden muss. Stattdessen werden alle Punkte zusätzlich in einer Liste gespeichert, die beim bearbeiten des Netzes das UnityMesh aktualisiert. Werden dem Half-Edge-Mesh neue Punkte hinzugefügt, können diese mit den Methoden *AddMeshVertex* und *AddMeshVertices* ergänzt werden. Am Ende der Methode wird die aktualisierte Liste dem UnityMesh übergeben.
Auch die Triangles des UnityMeshes werden gecached. Da Manipulationen des Meshes in der Regel bedeuten, dass sich die Triangels relativ zu den drei Punkte einer Face verändern, werden diese in dreier Tuplen in ein Dictionary geschrieben. Der Schlüssel ist dabei der Index der Face. Die Indexliste lässt sich mit *SetMeshTriangles* bearbeiten. Dabei wird ein Eintrag an der Stelle des Faceindex hinzugefügt, sofern er nicht vorhanden ist oder verändert, falls ein Eintrag existiert. Wichtig ist dies zum Beispiel, wenn eine Face geteilt wird und ein Dreieckseintrag mit zwei von drei Punkten übernommen wird. Als weiteren Parameter kann angegeben werden, ob eine Veränderung Teil einer größeren Transaktion war, um zu vermeiden, dass bei umfangreicheren Operationen, wie der Subdivision aller Faces, für jeden Methodenaufruf das Dictionary in eine Liste umzuwandeln und das Mesh erneut rendern zu müssen. Wird diese Option verwendet, muss nach Abschluss der Transaktion *CommitMeshTriangles* ausgeführt werden, um die Änderungen ins Mesh zu übernehmen.

2.3.2 GenerateHalfEdgeMesh

3 Rechnungen

Um die Werte aus Tabelle 1 zu erhalten, müssen folgende Annahmen getroffen werden: Bei der Speichernutzung wird ein allgemeines Mesh mit n_v Vertices und n_t Triangles betrachtet. Allgemein kann angenommen werden, dass ein Mesh ungefähr doppelt so viele Eckpunkte wie Dreiecke hat, woraus sich $n_t \approx 2 * n_v$ ergibt. Zudem kann die Annahme getätigt werden, dass in C# ein *Vector3*, der aus drei *floats* besteht eine Größe von $3 * 32Bit = 12Byte$ hat, ein *int* $4Byte$ groß ist und eine Referenz auf ein Objekt bei einer 64 Bit-Architektur $8Byte$ benötigt.

3.1 Indexed Mesh

$n_v \times 12Byte$(Vector3) $+ 3 \times n_t \times 4Byte$(Pro Dreieck gibt es 3 int Indices)
$\approx n_v \times 12Byte + 6n_v \times 4Byte$
$\approx \underline{36 \times n_vByte}$

3.2 Triangle-Neighbor Structure

$n_t \times (3 \times 8Byte$(3 Referenzen auf Nachbardreiecke)$+3 \times 8Byte$(3 Referenzen auf Eckpunkte))$+$
$n_v \times (8Byte$(Referenz auf ein anliegendes Dreieck) $+ 12Byte$(Vector3))
$\approx 2 \times n_v \times (24Byte + 24Byte) + n_v \times 20Byte$
$\approx 96 \times n_vByte + 20 \times n_vByte$
$\approx \underline{116 \times n_vByte}$

3.3 Winged-Edge Mesh

$n_v \times *(8Byte$(Referenz von Vertex auf Edge)$+12Byte$(Vector3))$+n_t \times 8Byte$(Referenz von Face auf Edg
$1.5n_t$(Jede Edge wird von zwei Faces verwendet)$\times(4 \times 8Byte$Referenz der Kante auf andere Kanten+
$2 \times 8Byte$(Referenz auf anliegende Faces) $+ 2 \times 8Byte$(Referenz auf Eckpunkte))
$\approx 20 \times n_vByte + 2 \times n_v \times 8Byte + 3 \times n_v \times 64Byte$
$\approx 20 \times n_vByte + 16 \times n_vByte + 192n_vByte$
$\approx \underline{228 \times n_vByte}$

3.4 Half-Edge Mesh

$n_v \times *(8Byte$(Referenz von Vertex auf Edge)$+12Byte$(Vector3))$+n_t \times 8Byte$(Referenz von Face auf Edg
$3 \times n_t(4 \times 8Byte$(Referenzen der HalfEdge auf den ausgehenden Punkt, die nächste und gegenüberliege
$\approx 20 \times n_vByte + 2 \times n_v \times 8Byte + 6 \times n_v \times 32Byte$
$\approx \underline{228 \times n_vByte}$

Literatur

[Wik] Wikipedia the free encyclopedia. Polygon mesh, 2007. https://commons.wikimedia.org/wiki/File:Dolphin$_t riangle_m esh.png/media/File$: $Dolphin_t riangle_m esh.png[Online; zuletztbesuchtam 04.12.2019].David\ StasiukWill\ Pearson$

[WP17]

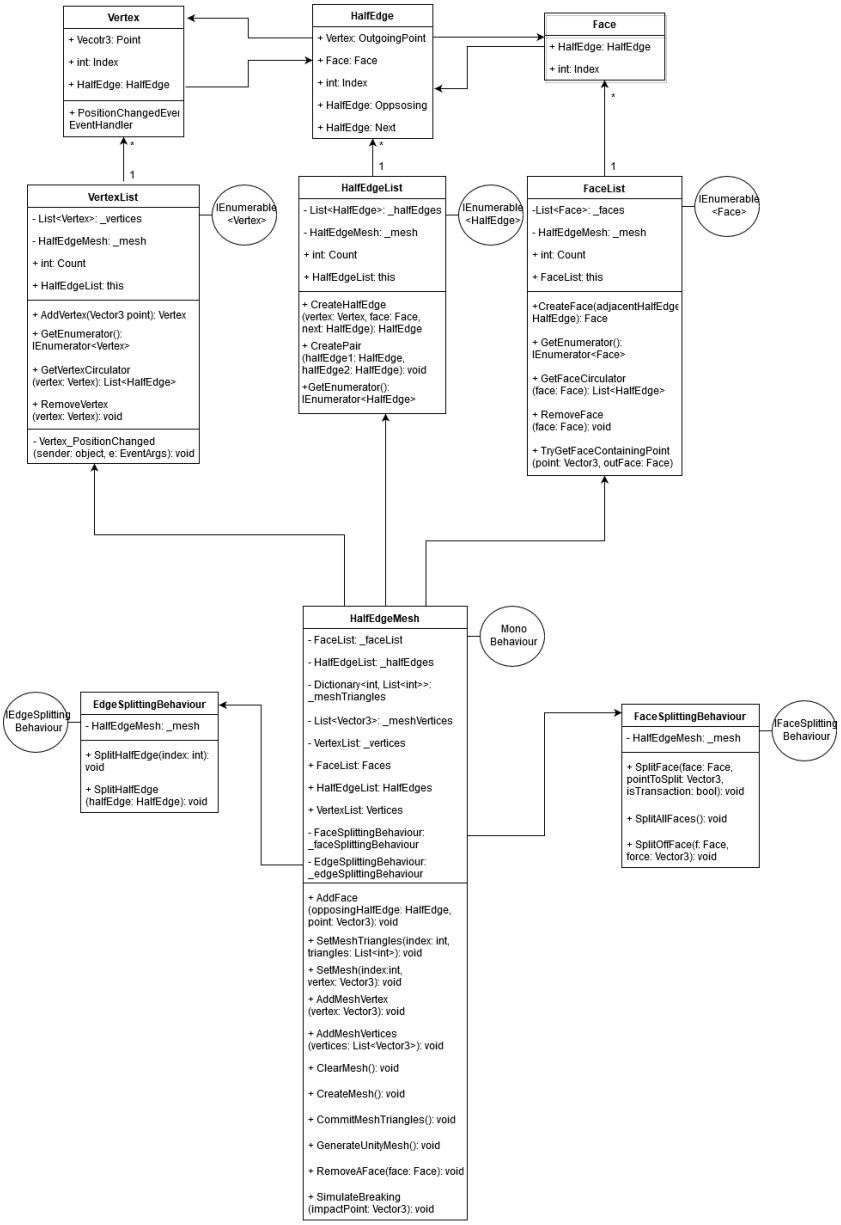

Abbildung 4: UML-Klassendiagramm des Half-Edge-Mesh Projekts