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 Abstract

Das Ziel dieser Arbeit ist es, die Performance von Unity-Meshs mit der eines Half-Edge Mesh zu vergleichen. In dieser Arbeit wird versucht, die folgende Forschungsfrage zu beantworten: Wie unterscheidet sich die Laufzeit von Unity-Meshs und Half-Edge Meshs, bei der Ausführung von Standartmethoden für dynamische Anwendungen und was bedeutet dies für ihre Anwendungsbereiche? Um diese Frage zu beantworten, wird ein Prototyp entwickelt und das Half-Edge Mesh theoretisch betrachtet, um Aussagen über das durchschnittliche Laufzeitverhalten treffen zu können. Diese Analysen haben gezeigt, dass sich Half-Edge Meshs durch ihre deutlich bessere Laufzeit für dynamische Meshs besser eignen, dafür allerdings einen höheren Speicherbedarf aufweisen. Deshalb sollte bei größeren Meshs abgewogen werden, ob der Anwendungsfall ein Half-Edge Mesh benötigt oder ob die native Unity-Lösung ausreichend ist.

2 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.

2.1 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 erwecken 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 der Nintendo Switch. Zudem sind 50% aller mobilen Spiele und 60% aller VR/AR Anwendungen mithilfe von Unity entstanden [3]. Spiele wie ,,Pokémon Go", ,,Superhot" und das Simulationsspiel ,,Universe Sandbox" sind drei Beispiele für Spiele, die in Unity erstellt wurden.

2.2 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, wie in Abbildung 1 anhand eines Delfins gezeigt. Informationen, die für ein solches Dreiecksnetz garantiert benötigt werden, sind eine Reihe von Eckpunkten, den Vertices, die immer in Dreiermengen kommen und deren Position im dreidimensionalen Raum [2, S.262]. Diese werden durch Kanten (Edges) verbunden und bilden damit Dreiecksflächen, auch Faces genannt.

Die einfachste Implementierung eines solchen Meshes sieht vor, für jedes Dreieck drei Punkte mit jeweils 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. Eine konsequente Behandlung der Orientierung der Dreiecke ist für jede Art von Mesh wichtig. 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 [2, S.265].

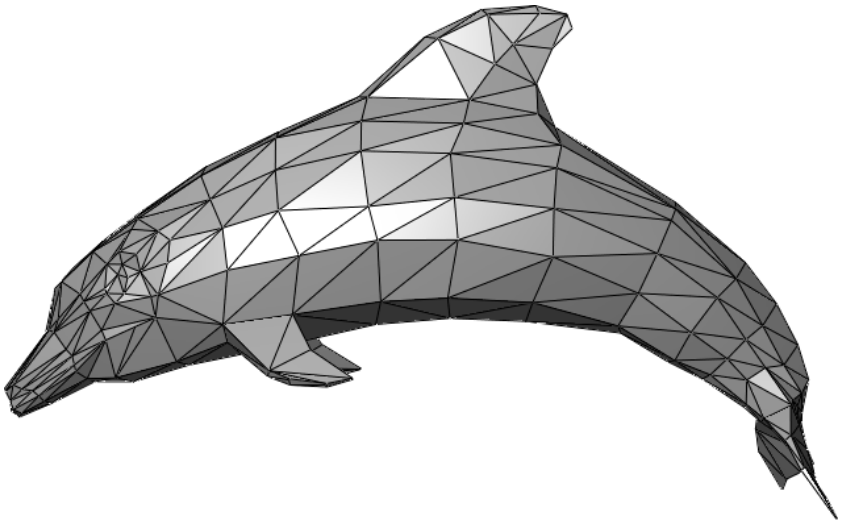

Abbildung 1: Beispiel eines Dreiecks-Polygonen-Netz, von [4]

2.3 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 Reihenfolge der Eckpunkte festlegt, indem die Indizes der zu verwendenden Vertices angegeben werden. Zu beachten ist, dass die Vertices eines Unity-Meshs immer im Uhrzeigersinn orientiert sind, 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        {0, 1, 2,   // —— Dreieck 1
30         2, 1, 3};  // —— Dreieck 2
31
32      //—— Mesh bef\"ullen
33      mesh.Clear();
34      //—— Vertices zuweisen
35      mesh.vertices = verticies.ToArray();
36      //—— Triangles zuweisen
37      mesh.triangles = triangles.ToArray();
38      //—— Mesh dem MeshFilter zuweisen
39      meshFilter.sharedMesh = mesh;
40  }
```

Und liefert folgendes Ergebnis:

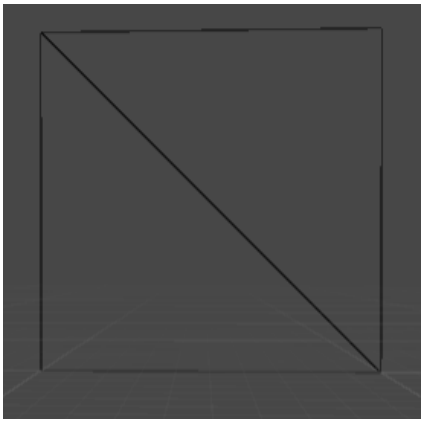

Abbildung 2: Die Wireframeansicht des erstellten Meshes im Unity Editor

2.4 Datenstrukturen für Meshes

Wird zur Laufzeit eine Nachbarschaftsbeziehung abgefragt, zum Beispiel welche Faces oder 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.

2.4.1 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 [2, S.269]:

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

class Vertex
{
  // --- Vertex specific data
  Vector3 Point;
  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.

2.4.2 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 [2, S.273]:

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

```
7
8  class Face
9  {
10   // --- Face specific data
11   Edge Edge;
12 }
13
14 class Vertex
15 {
16   // --- Vertex specific data
17   Vector3 Point;
18   Edge Edge;
19 }
```

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. Diese Unannehmlichkeit wird im Half-Edge Mesh gelöst.

3 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.

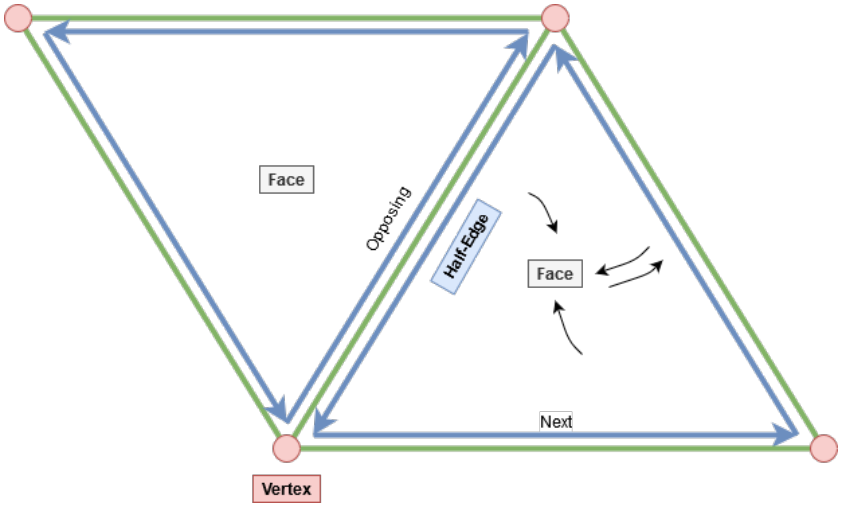

Abbildung 3: Die Elemente eines Half-Edge-Mesh.

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 3 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.

3.1 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.

3.2 Implementierung der Komponenten

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

3.2.1 Klassenstruktur

Aus den beschriebenen Elementen einer Half-Edge-Netzstruktur ergibt sich das in Abbildung 4 gezeigte Klassendiagramm. Der grundlegende Aufbau der einzelnen Klassen basiert dabei auf dem Plankton-Mesh [5]. 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.

3.2.2 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 [5]

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.

3.2.3 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,
3    HalfEdge next)
4  {
5    HalfEdge halfEdge = new HalfEdge(vertex, face, next, Count);
6    vertex.HalfEdge = halfEdge;
7    // ---- _halfEdges ist die zugrunde liegende Liste
8    _halfEdges.Add(halfEdge);
9    return halfEdge;
10 }
```

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:

```
1  public List<HalfEdge> GetFaceCirculator(Face f)
2  {
3    List<HalfEdge> result = new List<HalfEdge>();
4    result.Add(f.HalfEdge);
5    result.Add(f.HalfEdge.Previous);
6    result.Add(f.HalfEdge.Previous.Previous);
7    return result;
8  }
```

3.3 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.

3.3.1 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)
```

```
9           .ToList();
10        SetMeshTriangles(face.Index, adjacentHalfEdges
11          .Select(p => p.OutgoingPoint.Index).ToList(), true);
12    }
13
14    AddMeshVertices(vertices);
15    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.

3.3.2 GenerateHalfEdgeMesh

Andersherum kann es genauso sinnvoll sein, ein bestehendes UnityMesh in ein Half-Edge Mesh umzuwandeln, um die Vorteile dieser nutzen zu können. Die Logik dahinter befindet sich in der *HalfEdgeMeshBuilder*-Klasse. Der Builder kann dem HalfEdge-Mesh hinzugefügt werden und besitzt eine Referenz auf das HalfEdgeMesh, um dieses bearbeiten zu können. Um ein UnityMesh in ein HalfEdgeMesh zu überführen, kann die *BuildHalfEdgeMeshFromUnityMesh* verwendet werden. Diese Methode füllt die drei Komponentenlisten anhand der Daten aus dem UnityMesh. Um dies zu erreichen gelten drei Grundsätze:

1. Für jeden Punkte gibt es einen Vertex
2. Für jeden Index gibt es eine HalfEdge, vom Punkt mit dem Index ausgehend
3. Jeweils drei Indices bilden eine Face.

```
public void BuildHalfEdgeFromUnityMesh(Mesh mesh)
{
  List<Vector3> vertices = mesh.vertices;
  List<int> triangles = mesh.triangles;

  // --- Add vertices to HalfEdgeMesh
  int[] indexChanges = new int[vertices.Length];
  for (int i = 0; i < vertices.Length; i++)
  {
    Vector3 point = vertices[i];
    // --- CreateVertex checks if position is already saved
    // --- and returns the first matching Vertex
    Vertex vertex = _mesh.Vertices.CreateVertex(point);

    // --- Saving the index change
    indexChanges[i] = vertex.Index;
  }

  // --- Change Vertex Index to first occurence
  if (indexChanges.Any())
  {
    for (int i = 0; i < triangles.Length; i++)
    {
      // --- change here!
      triangles[i] = indexChanges[triangles[i]];
    }
  }

  // --- For every three indices in triangles (= a face) add a face
  // --- and three halfEdges
  for (int i = 0; i < triangles.Length - 3; i += 3)
  {
    Vertex point1 = _mesh.Vertices[triangles[i]];
    Vertex point2 = _mesh.Vertices[triangles[i + 1]];
    Vertex point3 = _mesh.Vertices[triangles[i + 2]];
    int[] indices = new[] {
      point1.Index,
      point2.Index,
      point3.Index
    };
    List<HalfEdges> pairs = _mesh.HalfEdges
      .Where(p => indices.Contains(p.OutgoingPoint.Index)
      && indices.Contains(p.EndPoint.Index));

    HalfEdge halfEdge1 = _mesh.HalfEdges
      .CreateHalfEdge(point1, null, null);
```

```
47
48      Face face = _mesh.Faces.CreateFace(halfEdge1);
49
50      HalfEdge halfEdge2 = _mesh.HalfEdges
51        .CreateHalfEdge(point2, face, halfEdge1);
52      HalfEdge halfEdge3 = _mesh.HalfEdges
53        .CreateHalfEdge(point3, face, halfEdge2);
54      halfEdge1.Next = halfEdge3;
55
56      foreach (HalfEdge pair in pairs)
57      {
58        if (pair.OutgoingPoint.Index == halfEdge1.EndPoint.Index
59        && pair.EndPoint.Index == halfEdge1.OutgoingPoint.Index)
60          _mesh.HalfEdges.CreatePair(pair, halfEdge1);
61        else if (pair.OutgoingPoint.Index == halfEdge2.EndPoint.Index
62        && pair.EndPoint.Index == halfEdge2.OutgoingPoint.Index)
63          _mesh.HalfEdges.CreatePair(pair, halfEdge2);
64        else if (pair.OutgoingPoint.Index == halfEdge3.EndPoint.Index
65        && pair.EndPoint.Index == halfEdge3.OutgoingPoint.Index)
66          _mesh.HalfEdges.CreatePair(pair, halfEdge3);
67      }
68    }
69  }
```

Beim Erstellen des HalfEdgeMeshes muss beachtet werden, dass beim erzeugen eines Vertex mit *_mesh.Vertices.CreateVertex(point)* geprüft wird, ob dieser Punkt bereits vorhanden ist, um redundante Daten zu vermeiden und ein zusammenhängendes Netz zu garantieren. Wenn in einem UnityMesh allerdings einen Punkt im *vertices*-Array mehrfach vorkommt, verschieben sich nach dieser Stelle alle weiteren Indices. Somit muss zu beginn das UnityMesh ,,bereinigt" werden. Beim Erstellen der Vertices werden etwaige Indexänderungen gespeichert und für alle Indices geprüft. Anschließend werden, wie beim erstellen des UnityMeshes auch, jeweils drei Indices gleichzeitig betrachtet.

Zu den Punkten hinter den Indices wird je eine HalfEdge erstellt und zu jeder Triplette wird eine Face erstellt, mit Referenz auf eine der HalfEdges. Zudem müssen alle HalfEdges, die ein Paar mit einer aktuell betrachteten HalfEdge bilden, ermittelt werden. Dafür werden alle HalfEdges gesucht, dessen eingehender und ausgehender Punkt in der Liste der aktuell betrachteten Punkte liegt:

```
1  List<HalfEdge> pairs = _mesh.HalfEdges
2    .Where(p => indices.Contains(p.OutgoingPoint.Index)
3    && indices.Contains(p.EndPoint.Index));
```

4 Operationen

Nachdem ein Half-Edge Mesh prozedural erstellt wurde oder ein Unity Mesh umgewandelt wurde, können auf ein Half-Edge Mesh einige Standardoperationen angewendet werden. Folgende Operationen werden im folgenden Kapitel erklärt und die Implementierung beschrieben:

- das Teilen einer HalfEdge (Split HalfEdge),
- das Kollabieren einer Kante (Edge Collapse),
- das Aufteilen einer Face (Subdivision) und
- eine einfache 2D-Zerstörungssimulation (Simulate Breaking).

4.1 Split Half-Edge

Ziel der *SplitHalfEdge*-Methode ist es, eine HalfEdge so zu teilen, dass das Ergebnis der Operation ein konformes Half-Edge Mesh ist, also dass jede Face immer noch aus drei Half-Edges besteht und dass jede Half-Edge maximal einen Partner besitzt. Um das Teilen durchzuführen, müssen folgende Schritte durchgeführt werde, die in der Abbildung 5 schematisch dargestellt sind:

1. Bestimme die zu splittende HalfEdge.
2. Erzeuge einen neuen Punkt, dort wo sich die HalfEdge aufteilt.
3. Erzeuge drei neue HalfEdges und eine neue Face:
   - Eine HalfEdge zum neuen Punkt, vom Ursprung der gesplitteten HalfEdge
   - Eine HalfEdge als Nachfolger der ersten HalfEdge, mit dem neuen Punkt als Ursprung
   - Eine HalfEdge als Vorgänger der gesplitteten HalfEdge
4. Setze die Referenzen der HalfEdges neus
5. Führe Schritte 1-4 für die der gesplitteten HalfEdge gegenüberliegende HalfEdge ebenfalls aus
6. Setze die Paare neu

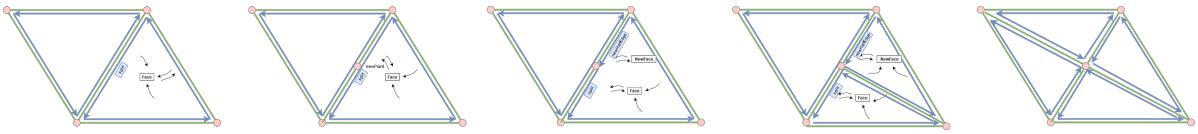

Abbildung 5: Schematische Darstellung eines Edgesplits. In Bild 1 ist die Ausgangssituation dargestellt, Bild 2 zeigt Schritt 2, die Bilder 3 und 4 zeigen die Schritte 3 und 4 und Bild 5 zeigt das Ergebnis, nach den Schritten 5 und 6.

Der beschriebene Algorithmus ist im Folgenden implementiert:

```
1  private bool _lock = false;
2  private HalfEdge _newHalfEdge, _split;
3
4  public void SplitHalfEdge(HalfEdge split, Vector3 splitPoint)
5  {
6    // --- Schritt 1:
7    split.Face.HalfEdge = split; // --- Set Reference of
8    // --- Face to split to know where the new face goes
9    _split = split;
10
11   // --- Schritt 2:
12   Vertex newPoint = _mesh.Vertices.CreateVertex(splitPoint);
13
14   // --- Schritt 3:
15   HalfEdge newHalfEdge = _mesh.HalfEdges
16     .CreateHalfEdge(split.OutgoingPoint, null, split);
17   _newHalfEdge = newHalfEdge;
18   Face newFace = _mesh.Faces.CreateFace(newHalfEdge);
19   split.OutgoingPoint = newPoint;
20
21   HalfEdge newHalfEdgeToSplit = _mesh.HalfEdges
22     .CreateHalfEdge(split.Next.EndPoint, split.Face, split);
23   HalfEdge newHalfEdgeFromNewHalfEdge = _mesh.HalfEdges
24     .CreateHalfEdge(newPoint, newFace, split.Next.Next);
25   _mesh.HalfEdges
26     .CreatePair(newHalfEdgeToSplit, newHalfEdgeFromNewHalfEdge);
27
28   // --- Schritt 4:
29   newHalfEdge.Next = newHalfEdgeFromNewHalfEdge;
30   newHalfEdgeFromNewHalfEdge.Next.Next = newHalfEdge;
31   split.Next.Next = newHalfEdgeToSplit;
32
33   // --- Aktualisiere das Unity Mesh
34   _mesh.AddMeshVertex(newPoint.Point);
35
36   _mesh.SetMeshTriangles(split.Face.Index,
37   _mesh.Faces.GetFaceCirculator(split.Face)
38     .Select(p => p.OutgoingPoint.Index)
39     .ToList(), true);
40
41   _mesh.SetMeshTriangles(newFace.Index,
42   _mesh.Faces.GetFaceCirculator(newFace)
43     .Select(p => p.OutgoingPoint.Index)
44     .ToList(), true);
45
```

```
46    if (!_lock) // ---- Blockiere nach dem ersten Aufruf,
47          // ---- um eine Endlosschleife zu vermeiden
48    {
49      // ---- Schritt 5:
50      if (split.Opposing != null)
51      {
52        _lock = true;
53        SplitHalfEdge(split.Opposing, splitPoint);
54        // ---- Schritt 6:
55        _mesh.HalfEdges.CreatePair(_split, newHalfEdge);
56        _mesh.HalfEdges.CreatePair(split, _newHalfEdge);
57        _lock = false;
58      }
59      _mesh.CommitMeshTriangles();
60    }
61  }
```

Wird die *SplitHalfEdge*-Methode auf das erste Mesh in Abbildung 6 angewendet, so ist das Mesh im zweite Bild das Ergebnis, wenn die Diagonale gesplittet wird. Wird stattdessen die linke Kante unterhalb der Mitte geteilt wird, entsteht das Mesh im dritten Bild. Das letzte Bild zeigt, dass der Teilungspunkt nicht zwingend auf der Kante liegen muss, da sich die Kanten entsprechend anpassen.

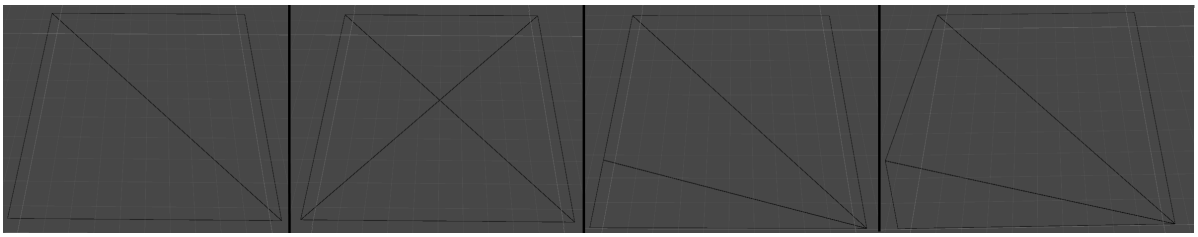

Abbildung 6: Bild 1 ist das Ausgangsmesh, welches in Bild 2 in der Mitte der Diagonale gesplittet wurde, in Bild 3 ein viertel auf dem Weg nach oben, auf der linken Seite und im 4. Bild eine Einheit links von der Kante

Der beschriebene Algorithmus besitzt eine konstante Laufzeit, da alle Änderungen lokal an den betroffenen HalfEdges vorgenommen werden können und die einzigen ,,komplexen" Operationen auf der *GetFaceCirculator*-Methode basieren, die immer die drei anliegenden HalfEdges einer Face zurückgibt. Wird im Gegensatz dazu die selbe Operation nur mithilfe vom UnityMesh versucht, so müsste mittels Brute-Force zuerst jeder Index der Eckpunkte der gesuchten Kante ermittelt werden, da diese nicht explizit dargestellt wird, um anschließend in den Indices die Dreiecke zu finden, die diese Eckpunkte verwenden, damit diese durch die neu bestimmten Dreiecke ersetzt werden können. Daraus resultiert eine Laufzeit in Abhängigkeit von der Anzahl der Vertices im Netz, wodurch es zu Performanceproblemen bei großen Netzen kommen kann.

4.2 Edge Collapse

Der Edge-Collapse ist eine Operation, die dazu dient, die Komplexität eines Models zu reduzieren. Dabei werden zwei Punkte zusammengeführt, indem eine Kante zusammenfällt, wodurch bei einer Operation zwei Dreiecksflächen entfernt werden, ohne die gesamte Struktur zu verändern [1]. In diesem Ansatz kann bestimmt werden, zu welchem Punkt die Kante zusammenfällt, je nachdem, welche der beiden HalfEdges als Ursprung gewählt wird, da dafür der Endpunkt der HalfEdge verwendet wird. Um einen Edge-Collapse durchzuführen, müssen die Schritte aus Abbildung 7 befolgt werden:

1. Bestimme die an der HalfEdge anliegenden Vertices, HalfEdges und Faces.
2. Setze die Referenzen der Nachbarschaftsbeziehung neu, die Nachbarn der HalfEdges die an der selben HalfEdge liegen und nicht teil der kollabierenden Kante sind, werden Nachbarn.
3. Entferne die betroffene Faces und HalfEdges.
4. Setze alle Referenzen von einem Vertex auf den andern, entferne den Unbenutzten.

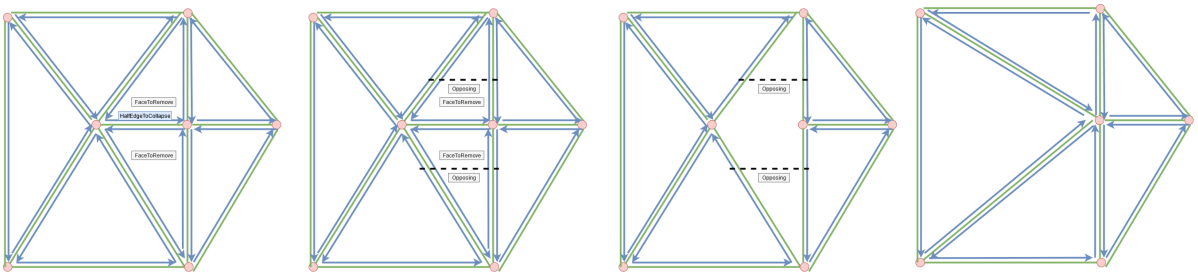

Abbildung 7: Schematische Darstellung des Edge-Collapses. Bild 1 zeigt die Ausgangssituation, Bild 2 stellt die neuen Nachbarschaftsreferenzen dar, in Bild 3 wurden die Faces entfernt und in Bild 4 ist das Ergebnis zu sehen.

Die Implementierung des beschriebenen Algorithmus sieht folgendermaßen aus:

```
1  public void EdgeCollapse(HalfEdge halfEdge)
2  {
3    // ~~ the point the halfEdge is collapsing to
4    Vertex collapsingPoint = halfEdge.EndPoint;
5    Vertex pointToRemove = halfEdge.OutgoingPoint;
6
7    // ~~ determine the halfEdges to remove
8    // ~~ those are the HalfEdges around the halfEdge Face
9    // ~~ and it's pairs face
10   HalfEdge pair = halfEdge.Opposing;
11
12   List<HalfEdeg> halfEdgesToRemove =
13   _mesh.Faces.GetFaceCirculator(halfEdge.Face);
```

```
14    halfEdgesToRemove
15    .AddRange(_mesh.Faces.GetFaceCirculator(pair.Face));
16
17    // --- set the neighbours right
18    foreach(HalfEdge he in halfEdgesToRemove)
19    {
20      if (he.Index == halfEdge.Index
21      || he.Index == pair.Index
22      || he.Next.Index == halfEdge.Index
23      || he.Next.Index == pair.Index
24      || he.Opposing == null
25      || he.Next.Opposing == null)
26        continue;
27
28      _mesh.HalfEdges.CreatePair(he.Opposing, he.Next.Opposing);
29    }
30
31    _mesh.Faces.RemoveFace(halfEdge.Face);
32    _mesh.Faces.RemoveFace(pair.Face);
33
34    foreach (HalfEdge he
35    in _mesh.Vertices.GetVertexCirculator(pointToRemove))
36    {
37      if(he.OutgoingPoint.Index == pointToRemove.Index)
38      {
39        he.OutgoingPoint = collapsingPoint;
40      }
41    }
42
43    _mesh.Vertices.RemoveVertex(pointToRemove);
44    _mesh.GenerateUnityMesh();
45  }
```

Die Methode *RemoveFace* ist eine Methode aus der *FaceList* und regelt auch das Löschen von allen weiteren Referenzen:

```
1   public void RemoveFace(Face f)
2   {
3     List<HalfEdge> halfEdges = GetFaceCirculator(f);
4     foreach (var halfEdge in halfEdges)
5     {
6       if (halfEdge.Opposing != null)
7       halfEdge.Opposing.Opposing = null;
8
9       if(_mesh.Vertices.GetVertexCirculator(halfEdge.OutgoingPoint)
10        .Count <= 0)
11      {
```

```
12            _mesh.Vertices.RemoveVertex(halfEdge.OutgoingPoint);
13        }
14    }
15    _mesh.HalfEdges.RemoveAll(p => halfEdges.Contains(p));
16
17    _faces.Remove(f);
18 }
```

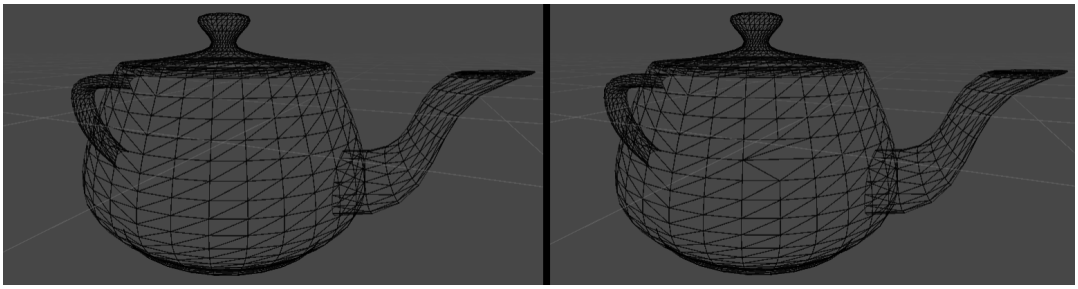

Abbildung 8: Edge-Collapse am Beispiel des Utah-Teapot

4.3 Subdivision

Im Gegensatz zum Edge-Collapse kann es beim Arbeiten mit 3D-Modellen auch nötig sein, eine Face in mehrere kleinere Faces zu unterteilen, um den Detailgrad des Modells zu erhöhen oder Zerstörung realistischer zu modellieren. Um dies zu erreichen, bietet die *FaceSplittingBehaviour*-Klasse die *SplitFace*-Methode, die eine gegebene Face an einem gegebenen Punkt in drei kleinere Faces Aufteilt. Um dies zu erreichen muss:

1. Der neue Punkt auf der Face erzeugt werden.
2. Für jede HalfEdge der Face:
   - Eine HalfEdge vom Endpunkt der HalfEdge zum neuen Punkt erzeugt werden.
   - Eine HalfEdge vom neuen Punkt zum Startpunkt der HalfEdge erzeugt werden.
   - Eine Face (wenn nötig) erzeugt werden.
   - Die ,,Next"-Referenz der HalfEdge auf die HalfEdge zum neuen Punkt gesetzt werden.
3. Für jede neue HalfEdge der Partner gesetzt werden.

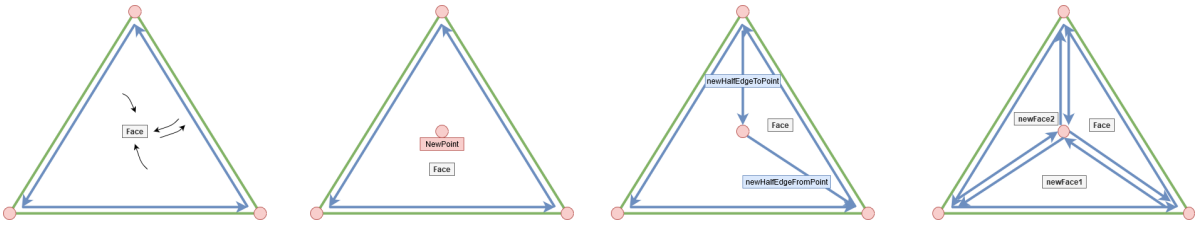

Abbildung 9: Schematische Darstellung des SplitFace Algorithmus. Bild 1 zeigt die Ausgangsposition, Bild 2 zeigt Schritt 2, Bild drei einen Teilschritt von Schritt 3 und Bild 4 zeigt das Ergebnis

In C# ergibt sich daraus der folgende Quellcode, welcher durch den Übergabeparameter *isTransaction* für den Gebrauch im gesamten Mesh optimiert ist.

```
1  public void SplitFace(Face face, Vector3 pointToSplit,
2    bool isTransaction = false)
3  {
4    // --- Schritt 1:
5    Vertex center = _mesh.Vertices.CreateVertex(pointToSplit);
6    _mesh.AddMeshVertex(center.Point);
7
8    // --- Schritt 2:
9    List<HalfEdge> halfEdges = _mesh.Faces.GetFaceCirculator(face);
10   List<HalfEdge> newHalfEdges = new List<HalfEdge>();
11   // --- Keeping a reference to all new HalfEdges to pair them later
12
13   // --- Schritt 3:
14   for (int i = 0; i < halfEdges.Count; i++)
15   {
16     HalfEdge halfEdge = halfEdges[i];
17     Face newFace = face;
18     // --- reuse old face
19     if (i == 0)
20     {
21       newFace.HalfEdge = halfEdge;
22     }
23     else
24     {
25       newFace = _mesh.Faces.CreateFace(halfEdge);
26     }
27
28     HalfEdge newHalfEdgeFromCenter =
29       _mesh.HalfEdges.CreateHalfEdge(center, newFace, halfEdge);
30     HalfEdge newHalfEdgeToCenter =
31       _mesh.HalfEdges.CreateHalfEdge(halfEdge.EndPoint,
32       newFace, newHalfEdgeFromCenter);
33     halfEdge.Next = newHalfEdgeToCenter;
```

```
34
35      _mesh.SetMeshTriangles(newFace.Index,
36      _mesh.Faces.GetFaceCirculator(newFace)
37        .Select(p => p.OutgoingPoint.Index)
38        .ToList(), true);
39      newHalfEdges.AddRange(new List<HalfEdge>{
40        newHalfEdgeFromCenter,
41        newHalfEdgeToCenter,
42        halfEdge
43      });
44    }
45
46    // --- Schritt 4:
47    _mesh.HalfEdges.CreatePair(newHalfEdges[0], newHalfEdges[4]);
48    _mesh.HalfEdges.CreatePair(newHalfEdges[3], newHalfEdges[7]);
49    _mesh.HalfEdges.CreatePair(newHalfEdges[6], newHalfEdges[1]);
50
51    // --- Optimisation for SplitAllFaces
52    if(isTransaction)
53      _mesh.CommitMeshTriangles();
54  }
```

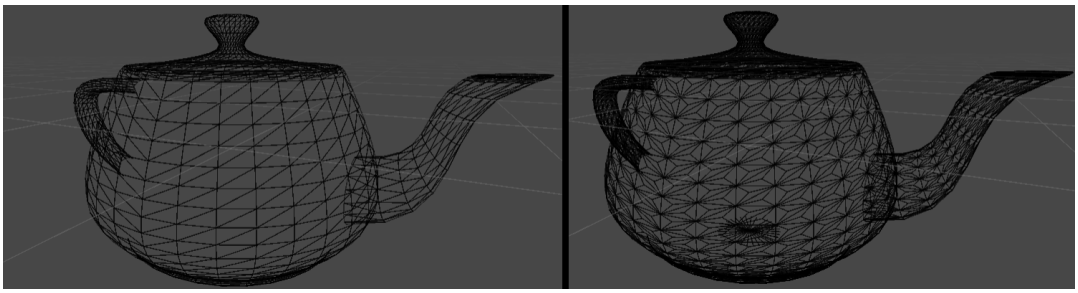

Abbildung 10: Subdivision aller Faces am Beispiel des Utah-Teapot

4.4 2D-Zerstörungssimulation

Mit Hilfe der Subdivision und dem Edgesplit kann eine einfache, dynamische Zerstörungssimulation für zweidimensionale Oberflächen erstellt werden, welche zum Beispiel im Kontext eines Videospiels eine interaktive Welt bietet.

Die Simulation basiert auf einem Punkt des Meshs, der das Zentrum der Zerstörung darstellt, vergleichbar mit dem Aufschlagpunkt eines Hammers. Um diesen Punkt herum werden zufällig weitere Punkte platziert, je nach Material der getroffenen Face unterschiedlich viele, in größer werden Abständen. Von der direkt getroffenen Face wird jede HalfEdge gesplittet, um so große Splitter, die sich durch das Material ziehen zu simulieren, da auch die gegenüberliegenden HalfEdges geteilt werden. Die verteilten Punkte werden Anschließend als neuer Punkt für die Subdivision verwendet, um so den Bruch

konzentrisch vom Epizentrum ausgehen zu lassen. Zu guter Letzt werden die neuen Faces aus dem Mesh ausgetrennt und in ein eigenes Mesh umgewandelt, die einzelnen Bruchstücke bewegen zu können.

In Unity ist es wie unten programmiert:

```
1  public void SimulateBreaking(Vector3 impactPoint)
2  {
3
4    if (Faces.TryGetFaceContainingPoint(impactPoint,
5    out Face impactedFace))
6    {
7      var bbd = new BreakingBehaviourDeterminer();
8
9      List<Vector3> breakpoints = bbd.GetBreakingPoints(impactedFace,
10      impactPoint);
11
12     List<HalfEdge> halfEdges =
13       Faces.GetFaceCirculator(impactedFace).ToList();
14     _faceSplittingBehaviour.SplitFace(impactedFace, impactPoint);
15
16     // --- Aufteilen der Urspruenglichen HalfEdges
17     foreach (HalfEdge oldHe in halfEdges)
18     {
19       _edgeSplittingBehaviour.SplitHalfEdge(oldHe,
20         VectorMath.GetIntermediateVector(
21         oldHe.OutgoingPoint.Point, oldHe.EndPoint.Point));
22     }
23
24     // --- Platzieren der neuen Punkte, splitten der Faces
25     foreach (Vector3 point in breakpoints)
26     {
27       if (Faces.TryGetFaceContainingPoint(point, out Face face))
28       {
29         List<HalfEdge> oldHalfEdgesOfFace =
30           Faces.GetFaceCirculator(face).Select(p => p.Next);
31         _faceSplittingBehaviour.SplitFace(face, point);
32       }
33     }
34
35     // --- Liste aller erzeugten Faces
36     List<Face> faces = _edgeSplittingBehaviour.GetCreatedFaces();
37     faces.AddRange(_faceSplittingBehaviour.GetCreatedFaces());
38
39     // --- Herausloesen der Faces
40     foreach (Face face in faces)
41     {
42       // --- Unity GameObject
```

```
43          GameObject go = new GameObject("Shard");
44
45          HalfEdgeMesh mesh = go.AddComponent<HalfEdgeMesh>();
46          go.AddComponent<Rigidbody>().useGravity = true;
47
48
49          List<HalfEdge> createdHalfEdges =
50            Faces.GetFaceCirculator(face).ToList();
51            List<Vertex> vertices = createdHalfEdges
52            .Select(p => p.OutgoingPoint).ToList();
53
54          mesh.CreateMesh(vertices[0].Point,
55            vertices[1].Point, vertices[2].Point);
56          Faces.RemoveFace(face);
57        }
58
59        // ---- entfernen der Erzeugten Faces
60        _edgeSplittingBehaviour.ClearCreatedFaces();
61        _faceSplittingBehaviour.ClearCreatedFaces();
62      }
63  }
```

Bei der Bestimmung der Bruchpunkte wird der *BreakingBehaviourDeterminer* verwendet, welcher zufällige Punkte in größer werdenden Kreisen platziert, relativ zu dem gewählten Punkt. Die Methode *GetMaterialBreakingBehaviour* gibt eine Liste mit Integern zurück, die die Anzahl an Punkten in jedem Kreis für ein Material vordefiniert.

Der Code der *GetBreakingPoints*-Methode sieht wie Folgt aus:

```
1   public List<Vector3> GetBreakingPoints(Face face,
2     Vector3 impactPoint)
3   {
4     var result = new List<Vector3>();
5
6     Vertex v1 = face.HalfEdge.OutgoingPoint.Point;
7     Vertex v2 = face.HalfEdge.Next.OutgoingPoint.Point;
8     Vertex v3 = face.HalfEdge.Next.Next.OutgoingPoint.Point;
9
10    List<int> materialBehaviour =
11      GetMaterialBreakingBehaviour(face.materialType);
12
13    for (int i = 0; i < materialBehaviour.Count; i++)
14    {
15      for (int j = 0; j < materialBehaviour[i]; j++)
16      {
17        Vector3 newPoint = UnityEngine.Random.insideUnitCircle *
18          (i + .5f) + new Vector2(impactPoint.x, impactPoint.z);
19        Vector3 addPoint = new Vector3(newPoint.x,
```

```
20            impactPoint.y, newPoint.y);
21          result.Add(addPoint);
22        }
23      }
24      return result;
25  }
```

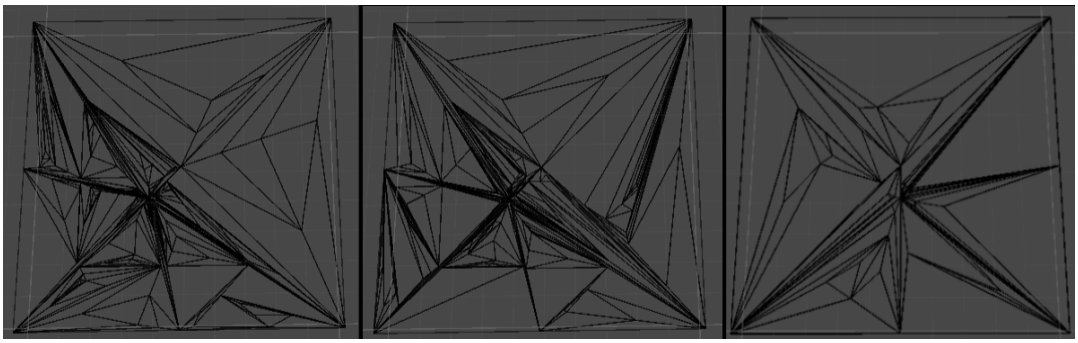

Abbildung 11: Die Ergebnisse von der Zerstörungssimulation mit unterschiedlichen Startpunkten

5 Vergleich zwischen Half-Edge Mesh und UnityMesh

Nachdem die Funktionsweise der wichtigsten Operationen für ein Half-Edge Mesh im vorherigen Kapitel beschrieben wurden, werden diese im Folgenden mit vergleichbaren Operationen eines UnityMeshs verglichen.

Bei den Methoden *SplitHalfEdge*, *EdgeCollapse* und *SplitFace* handelt es sich um Operationen auf dem Mesh, die lokale Änderungen verursachen. Da es einen direkten Zugriff auf die betroffenen Stellen gibt, mithilfe der Face- und Half-Edge-Listen muss nicht das gesamte Mesh nach den betroffenen stellen abgesucht werden. Zudem kann mittels der *GetFaceCirculator*-Methode auf alle betroffenen Half-Edges in konstanter Zeit zugegriffen werden, da diese Methode immer die drei Half-Edges über die Referenz der Face bestimmen kann. Insgesamt haben die *SplitHalfEdge*- und *SplitFace*-Methode also eine konstante Zeitkomplexität, unabhängig von der größe des Meshs. Um allerdings das gleiche Resultat mit einem UnityMesh zu erhalten, muss zuerst mittels Brute-Force jeder Index der Eckpunkte der gesuchten Kante oder Face ermittelt werden, da diese nicht explizit dargestellt werden. Diese Suche nach allen Kanten mit den selben Eckpunkten könnte wie folgt aussehen:

```
1  List<int> findEdges(int a, int b)
2  {
3    // --- All triangle indices
4    int[] tris = _mesh.triangles;
5
6    // --- The result contains every starting index
7    // --- of a matching triangle
8    List<int> result = new List<int>();
9
10   // --- for each pair of three
11   for(int i = 0; i < tris.Length; i+=3)
12   {
13     int matches = 0; // --- count the matching indices
14     if(tris[i] == a || tris[i] == b)
15       matches++;
16     if(tris[i+1] == a || tris[i+1] == b)
17       matches++;
18     if(tris[i+2] == a || tris[i+2] == b)
19       matches++;
20
21     // --- if there are two, consider this triangle index
22     if(matches == 2)
23       result.Add(i);
24   }
25   return result;
26 }
```

Dieser beispielhafte Implementierungsvorschlag kann benutzt werden, um die betroffe-

nen Faces für einen EdgeSplit zu finden, um die entsprechende Stelle für eine Subdivision zu finden, muss der dritte betroffene Punkt mitgegeben werden. Auffällig bei dieser Implementierung ist, dass es, im Gegensatz zum Half-Edge-Mesh abhängig von der größe des Meshs ist, wodurch es bei dieser Implementierung mit großen Netzen zu Laufzeitproblemen kommen kann.

Auch die *EdgeCollapse*-Methode verwendet die *GetFaceCirculator*-Methode mit konstanter Laufzeit. Zusätzlich wird die *GetVertexCirculator*-Methode verwendet, um die Half-Edges an den betroffenen Vertices zu finden. Theoretisch besitzt diese Methode eine konstante Laufzeit in Abhängigkeit von der Anzahl der Half-Edges, die den gefragten Vertex verwenden. Im Plankton-Mesh [5] wird folgende Implementierung verwendet:

```
public IEnumerable<int> GetVertexCirculator(int halfedgeIndex)
{
  if (halfedgeIndex < 0 || halfedgeIndex > this.Count) {
    yield break;
  }

  int h = halfedgeIndex;
  int count = 0;
  do
  {
    yield return h;
    h = this[this.GetPairHalfedge(h)].NextHalfedge;
    if (h < 0) {
    throw new InvalidOperationException
      ("Unset index, cannot continue.");
    }
    if (count++ > 999) {
      throw new InvalidOperationException
        ("Runaway vertex circulator");
    }
  }
  while (h != halfedgeIndex);
}
```

Allerdings funktioniert dieser Ansatz nur auf der zugrunde liegenden Annahme, dass das abgebildete Mesh vollständig ist, es also keine Half-Edge gibt, die keinen Nachbarn hat. Aus diesem Grund wird eine andere Implementierung verwendet, da diese Eigenschaft für ein solches Netz nicht gegeben sein muss. Diese sieht wie folgt aus:

```
public List<HalfEdge> GetVertexCirculator(Vertex v)
{
  return _mesh.HalfEdges
  .Where(p => p.OutgoingPoint.Index == v.Index).ToList();
}
```

Durch diese Generalisierung wird allerdings die konstante Laufzeit durch eine Lineare ersetzt, wodurch die Laufzeiten des Half-Edge-Meshs und des UnityMeshs jeweils in

lineare Abhängigkeit von der größe des Netzes stehen. Allerdings kann, wie das Plankton-Mesh zeigt, die *GetVertexCirculator*-Methode noch weiter optimiert werden.

Alles in allem kann gesagt werden, dass ein Half-Edge-Mesh mit konstanten Laufzeiten für die wichtigsten Operationen besser geeignet ist für Echtzeitsimulationen mit dynamischen Meshs, die sich zur Laufzeit verändern. Auch eignet sich ein Half-Edge-Mesh zur Bearbeitung eines Meshs, um den Detailgrad mithilfe der Subdivision zu erhöhen oder durch EdgeCollapse zu reduzieren. Allerdings kommen diese Möglichkeiten mit dem Nachteil, dass die Grundstruktur des Half-Edge-Mesh allein mehr als sechs mal so groß ist, wie ein Indexed Mesh der selben Anzahl an Vertices. Für statische Meshs eignet sich daher eher das UnityMesh und allgemein muss auf den einzelnen Anwendungsfall geachtet werden, um zu entscheiden, ob ein Half-Edge-Mesh oder ein Indexed Mesh verwendet werden soll.

6 Ausblick

Im Anschluss an diese Arbeit gibt es noch einige Features, um die diese Implementierung eines Half-Edge-Mesh erweitert werden kann. Zum Einen muss die *GetVertexCirculator*-Methode optimiert werden, um eine konstante Zeitkomplexität bei allen Grundoperationen zu erhalten, damit es in Echtzeitanwendungen verwendet werden kann.

Des Weiteren ist ein Ziel, die 2D-Zerstörungssimulation zu erweitern, um diese in dreidimensionalen Simulationen verwenden zu können. Dies kann geschehen, indem die Bruchpunkte nicht nur, wie bei der 2D-Simulation, auf der Oberfläche, sondern auch innerhalb des Modells verteilt werden. Die erzeugten ,,Scherben" werden um einen vierten Punkt erweitert, wodurch diese ein Tetraeder bilden und zu einem dreidimensionalen Splitter werden.

In diesem Zuge sollte die Simulation auch die Kraft (force), die auf das Modell wirkt, betrachtet werden, um in unterschiedlichen Situationen realistischere Resultate zu erzielen.

Sollte der gewählte Ansatz nicht die gewünschten Effekte erzielen, kann eine andere Methode verwendet werden, um die Brucheffekte zu berechnen, zum Beispiel mithilfe von Voronoi-Diagrammen.

Zum Schluss soll alles in einer Simulation zusammengefügt werden, die als Demo dient, in der die Effekte gezeigt werden, um sie anschließend in Anwendungen, wie zum Beispiel Computerspielen verwenden zu können.

7 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.

7.1 Indexed Mesh

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

7.2 Triangle-Neighbor Structure

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

7.3 Winged-Edge Mesh

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

7.4 Half-Edge Mesh

$n_v \times *(8Byte$(Referenz von Vertex auf Edge)+12$Byte$(Vector3))+n_t×8$Byte$(Referenz von Face auf Edg
3×n_t(4×8$Byte$(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

[1] P. Castello u. a. „Viewpoint entropy-driven simplification“. In: *WSCG '2007: Full Papers Proceedings*. Jan. 2007, S. 249–256.

[2] Peter Shirley und Steve Marschner. *Fundamentals of Computer Graphics*. Third Edition. CRC Press, 2010. Kap. 12 Data Structures for Graphics.

[3] *Unity. Public Relations*. [Zuletzt besucht: 15.01.2020]. URL: https://web.archive.org/web/20200114191426/https://unity3d.com/public-relations.

[4] the free encyclopedia Wikipedia. *Polygon mesh*. [Online; zuletzt besucht am 04.12.2019]. 2007. URL: https://commons.wikimedia.org/wiki/File:Dolphin_triangle_mesh.png/media/File:Dolphin_triangle_mesh.png.

[5] David Stasiuk Will Pearson Daniel Piker. *Plankton*. [Online; zuletzt besucht am 13.12.2019]. 2017. URL: https://github.com/meshmash/Plankton.

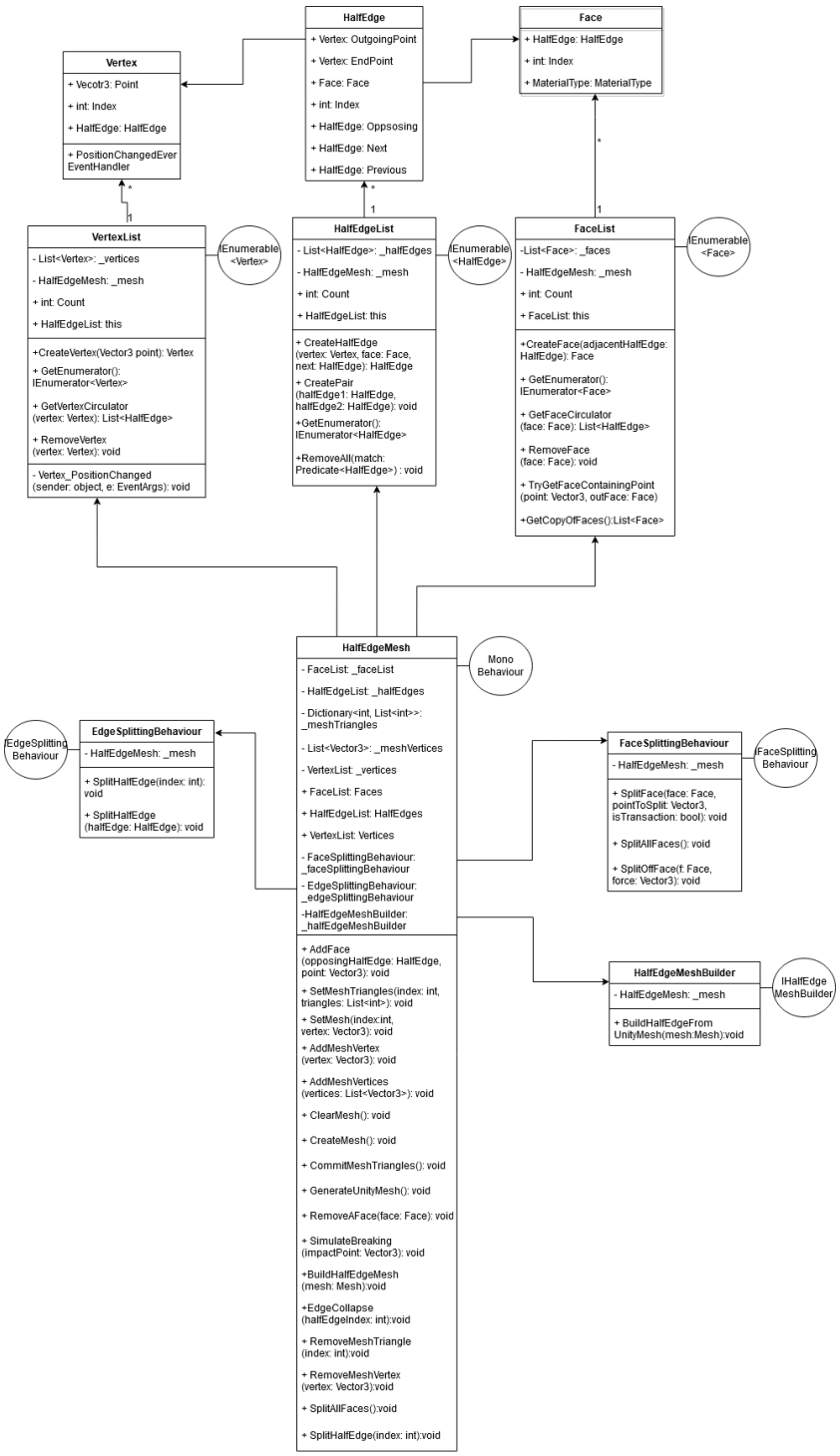

Abbildung 4: UML-Klassendiagramm des Half-Edge-Mesh Projekts