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

1 Einleitung

In der Computergrafik werden dreidimensionale Modelle als Polygonen-Netz (Polygonal Mesh) dargestellt, um diese auf einem zweidimensionalen Bildschirm darzustellen. Die Oberfläche eines Modells wird dabei mit Hilfe von Polygonen angenähert. Häufig werden dafür Dreiecks-Netze (Triangle Mesh) verwendet, wobei die Flächen mit Dreiecken nachmodelliert werden, siehe Abbildung 1.
Ein solches Netz besteht aus den Eckpunkten der einzelnen Dreiecke, den Vertices. Diese werden durch Kanten (Edges) verbunden und bilden damit die Polygonalflächen, auch Faces genannt.

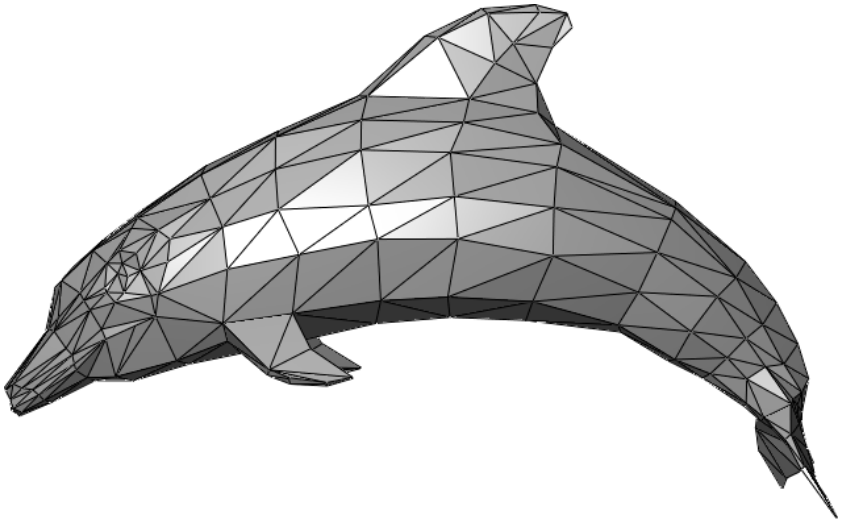

Abbildung 1: Beispiel eines Dreiecks-Polygonen-Netz, von [**?**]

2 Unity

Unity ist eine Spiele-Engine mit eingebauter Entwicklungsumgebung für 2D-, 3D- und VR-Spiele/-Simulationen. Die Engine kommt mit einem eigenen Editor, in welchem diverse Szenarien erstellt und bearbeitet werden können. Des Weiteren unterstützt Unity selbst programmierte Scripte auf der Grundlage von C#.

2.1 Meshes in Unity

Unity bietet die Möglichkeit, mit Hilfe von selbstgeschriebenen Scripten eigene 3D-Modelle zur Laufzeit erstellen zu lassen. Dafür stellt Unity ein eigenes Mesh-System zur Verfügung, basierend auf Dreiecksnetzen, die *UnityEngine.Mesh*-Klasse. Damit diese ein Mesh rendern kann, erwartet das Mesh ein *UnityEngine.Vector3*-Array für die Vertices, wobei ein *Vector3* einen Punkt im dreidimensionalen Raum darstellt und ein

int-Array, das die Reihenfolge der Vertices (über die Indices der Vertices) für die Dreiecke festlegt.
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   var P0 = new Vector3(0, 0, 0);
21   var P1 = new Vector3(0, 1, 0);
22   var P2 = new Vector3(1, 0, 0);
23   var P3 = new Vector3(1, 1, 0);
24
25   var verticies = new List<Vector3> { P0, P1, P2, P3 };
26
27   //--- Triangles
28   var 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:

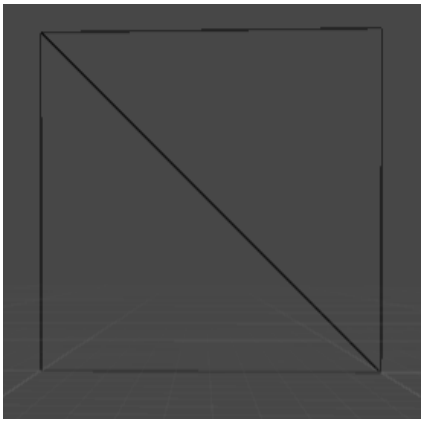

Abbildung 2: Die Wireframeansicht des erstellten Meshes im Unity Editor

2.2 Nachteile von Unity-Meshes

Die Vorteile bei dieser Herangehensweise sind, dass die von Unity verwendeten Netze auch bei größeren Netzen vergleichsweise wenig Speicherplatz benötigen, da dieser Ansatz auf eine direkte Repräsentation von Faces und Edges verzichtet. Daraus ergeben sich einige Nachteile. Durch die fehlende Referenz auf die Polygonenflächen sind Operationen auf Face-Ebene, wie eine Nachbarsuche, Abfragen auf alle Punkte Kanten oder die Unterteilung einzelner Polygonen in kleinere Einheiten (auch ,,Subdivision" genannt), sehr Zeitintensiv, weshalb sich solche Fälle nicht für Echtzeitanwendungen eignen. Des Weiteren stehen die Vertices im Unity-Mesh in keinem topographischen Zusammenhang, wodurch eine Iteration über das gesamte Netz erschwert wird, um beispielsweise zusammenhängende Punkte zu bearbeiten.

3 Half-Edge-Mesh

Um die oben genannten Problemstellungen zu lösen, gibt es andere Ansätze Polygonalnetze zu realisieren. Einer dieser Lösungen ist das Konzept der Half-Edge-Meshes. Ein solches Mesh besteht aus folgenden Komponenten:

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

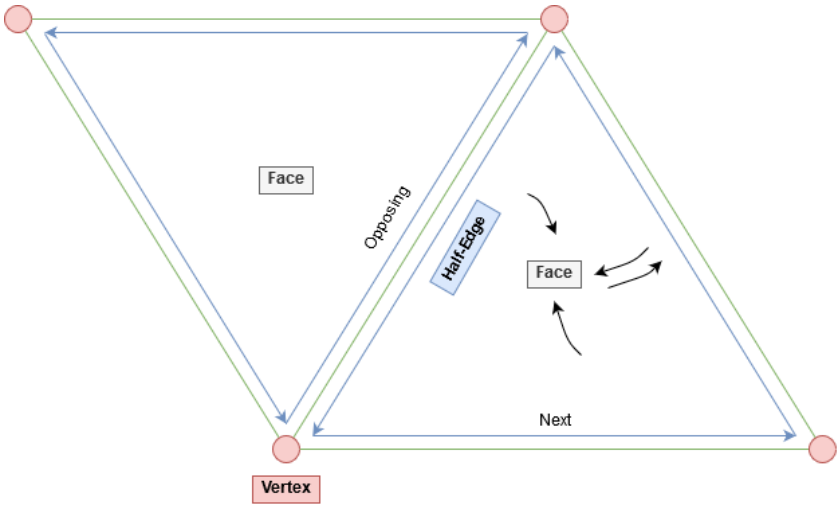

Abbildung 3: Die Elemente eines Half-Edge-Mesh.

Der Ansatz des Half-Edge-Mesh sieht dabei vor, dass jede Kante des Netzes aus zwei Half-Edges besteht, sodass jeweils eine Half-Edge auf einen Vertex der Kante zeigt, wie in Abbildung 3 gezeigt. Jede Half-Edge besitzt eine Referenz auf die ihr gegenüberliegende Half-Edge und auf die ihr Nachfolgende. Zudem referenziert sie die anliegende Face und den Vertex, aus dem sie hervorgeht.

3.1 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 [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.

3.2 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 Half-Edge 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.

3.3 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   var halfEdge = new HalfEdge(vertex, face, next, Count);
5   vertex.HalfEdge = halfEdge;
6   _halfEdges.Add(halfEdge);
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.

4 Implementierung des 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   var a = Vertices.CreateVertex(va);
5   var b = Vertices.CreateVertex(vb);
6   var c = Vertices.CreateVertex(vc);
7
8   var heA = HalfEdges.CreateHalfEdge(a, null, null);
```

```
9     var face = Faces.CreateFace(heA);
10    var heB = HalfEdges.CreateHalfEdge(b, face, heA);
11    var 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.

4.1 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   var vertices = Vertices.Select(p => p.Point).ToList();
4
5   // --- Add triangles
6   foreach (var face in Faces)
7   {
8     var 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 vorhan-

den 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.

Literatur

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

[WP17]

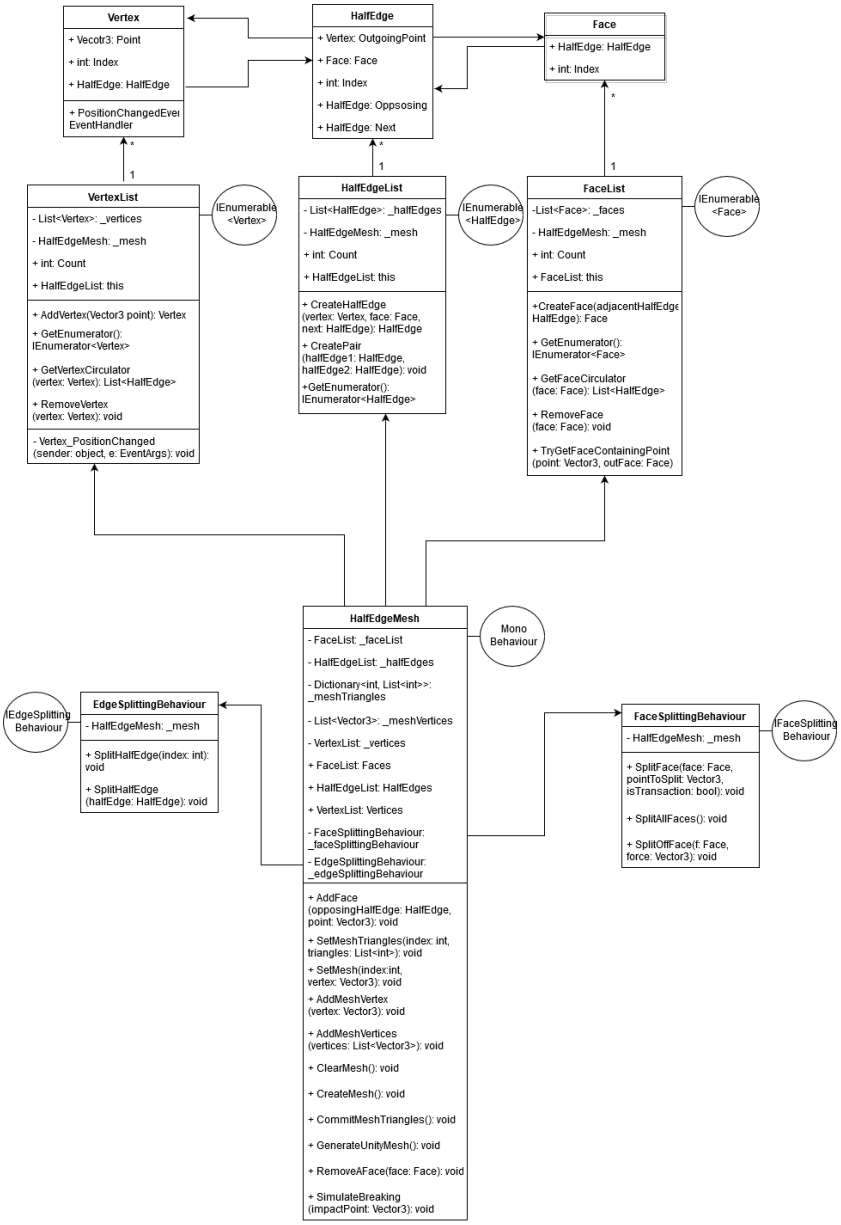

Abbildung 4: UML-Klassendiagramm des Half-Edge-Mesh Projekts