

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 Ingenieurwissenschaften

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.

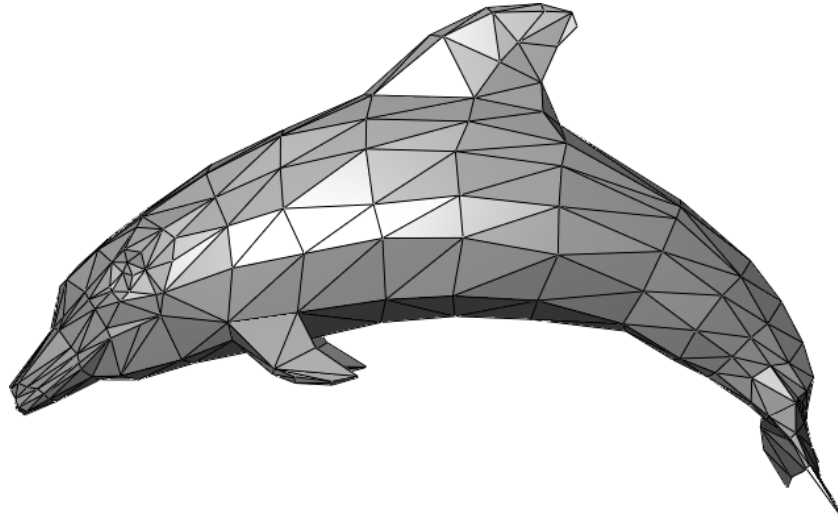


Abbildung 1: Beispiel eines Dreiecks-Polygonen-Netz, von [?]

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 Skripte auf der Grundlage von C#.

2.1 Meshes in Unity

Unity bietet die Möglichkeit, mit Hilfe von selbstgeschriebenen Skripten 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 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:

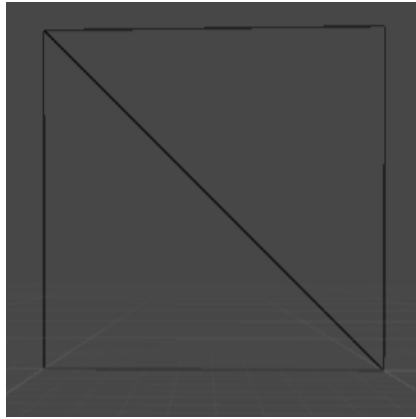


Abbildung 2: Die Wireframeansicht des erstellten Meshes im Unity Editor

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 Polygone 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.

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 Half-Edge-Mesh

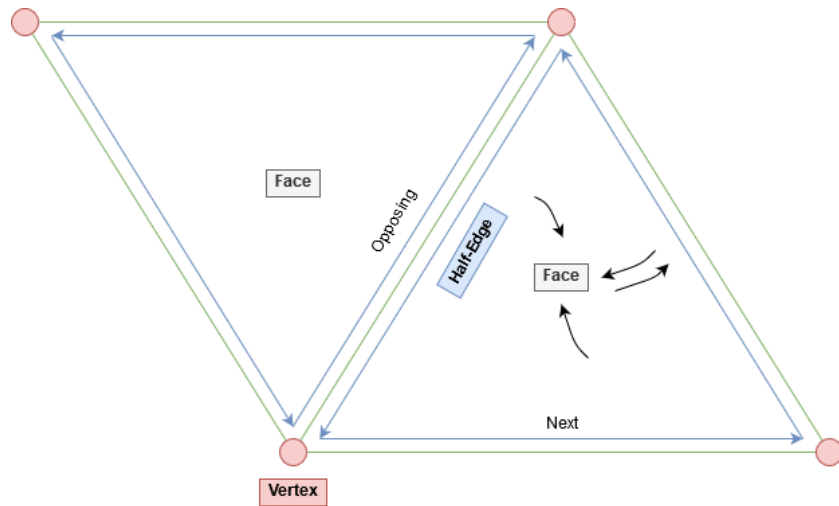


Abbildung 3: Die Elemente eines Half-Edge-Mesh.

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 Eigenschaften: 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 nur eine Referenz auf eine anliegende HalfEdge.

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-Mesh

Die Oben genannten Listenklassen werden von der *HalfEdgeMesh*-Klasse verwendet, um aus der beschriebenen Half-Edge-Datenstruktur ein von Unity renderbares Mesh zu erstellen.

Literatur

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

Literatur

[WP17] David Stasiuk Will Pearson, Daniel Piker. Plankton. <https://github.com/meshmash/Plankton>, 2017. [Online; zuletzt besucht am 13.12.2019].

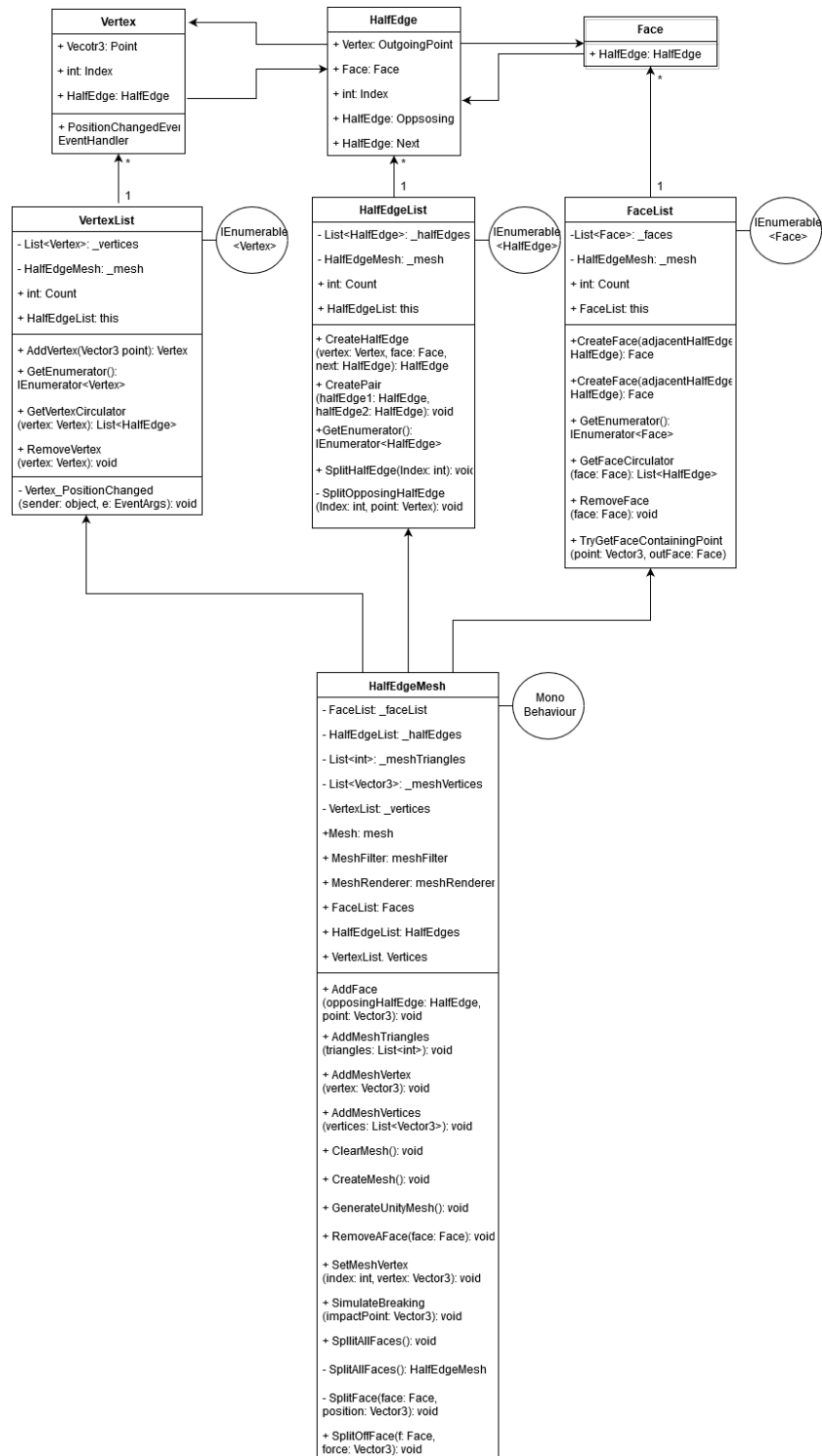


Abbildung 4: UML-Klassendiagramm des Half-Edge-Mesh Projekts