Fakultät Angewandte Computer- und Biowissenschaften

Medieninformatik und interaktives Entertainment
GPU Programmierung

Dokumentation Belegarbeit
Tessellation-Shader in Unity

Student:

Martin Schuster

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

A

API ..... *Programmierschnittstelle*

C

CPU ..... *Hauptprozessor*

F

FPS ..... *Frames Per Second*

G

GPU ..... Grafikprozessor

H

HLSL ..... *High-Level Shading Language*

L

LOD ..... *Level of Detail*

O

OpenGL ..... *Open Graphics Library*

T

TCS ..... *Tessellation-Control-Shader*
TES ..... *Tessellation-Evaluation-Shader*
TPG ..... *Tessellation-Primitive-Generator*

1 Einleitung

Die vorliegende Belegarbeit beschäftigt sich mit der Umsetzung des Tessellation-Shader in der Engine Unity. Diese Arbeit ist als Prüfungsleistung für das Modul GPU Programmierung, im 4. Semester des Studienganges Medieninformatik entstanden.

Im Rahmen des Moduls GPU Programmierung wurde die grundlegende Arbeitsweise von Grafikprozessoren (GPU) eines Computers vermittelt. Darauf aufbauend ist das Ziel der Belegarbeit, dass bisher gelernte zu vertiefen und in dem Maße zu erweitern, sodass die Aufgebstellung, Erstellung eines Tessellation-Shader in Unity, umgesetzt werden kann.

Eine Besonderheit stellt die Nutzung der High-Level Shading Language (HLSL) von Unity dar, was eine Abweichung von der bisher genutzten Programmierung in Open Graphics Library (OpenGL) darstellt.

Außerdem soll die Arbeit so gestaltet werden, dass Sie als Anleitung für die Umsetzung des Tessellation-Shader in anderen Projekten genutzt werden kann.

Der Tessellation-Shader ermöglicht es ein Mesh, die Geometrie, eines Objektes kleinteiliger zu unterteilen, sodass ein Nutzen von z. B. Displacement-Maps eine Verformung des Objektes möglich machen. Dabei erlaubt es der Tessellation-Shader, je nach gewählter Umsetzung, eine dynamische Verfeinerung dies ermöglicht eine genaue Anpassung an die Bedürfnisse des Programmes.

In der Belegarbeit wird zunächst auf die Grundlagen und die Funktionsweise genauer eingegangen und als praktisches Beispiel die Umsetzung in Unity erläutert. Das beiliegende Programm dient als interaktives Anwendungsbeispiel.

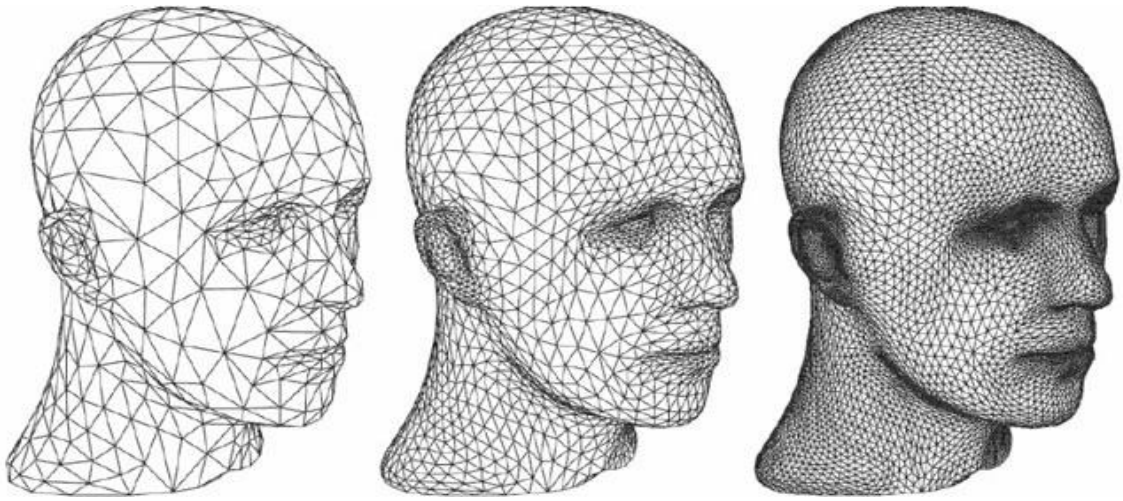

Abbildung 1 Tessellation Effekt
Quelle: https://computergraphics.stackexchange.com/questions/2018/what-is-tessellation-in-computer-graphics (19.09.2019)

2 Problemstellung/Hintergründe

Besonders in Spielen stellt die Darstellung von 3D-Objekten einen essenziellen Bestandteil für die Wirkung der Umgebung auf den Spieler dar. Dabei können nicht alle Objekte immer als hochauflösende 3D-Objekte im Spiel platziert werden, da dies die Speicherkapazitäten der meisten Grafikkarten überschreitet und dadurch die Dauer zur Berechnung und Darstellung der Objekte nicht mehr in der gewünschten Frequenz (Frames Per Seconds (FPS)) gewährleisten werden kann, dies führt zu stockenden Spielablauf.

Deshalb wird in den meisten Spiele-Engines unterschiedliche Modelle der Objekte verwendet. Dies wird meistens als Level of Detail (LOD) beschrieben. Dabei wird abhängig von der Kamera (Spielerfigur) ein jeweils anderes 3D-Modell in die Szene geladen. Diese stellen dasselbe Modell dar jedoch mit unterschiedlicher Polygon Anzahl, die Flächen aus denen das Objekt besteht. Da es die Entfernung zum Objekt erlaubt entsprechend auf Details verzichten zu können, wenn die Kamera entsprechend weitgenug entfernt ist und viele Details erfordert, wenn die Kamera nah am Objekt ist. Dieses Vorgehen erfordert jedoch einen erhöhten Aufwand für die Modellierungsabteilung, jedoch garantiert es nicht das immer ein passendes Modell vorhanden ist. Dies ist bedingt durch die Tatsache es nur eine begrenzte Anzahl an LOD Modellen gibt.

Dieser Problematik nimmt sich der Tessellation-Shader an, dabei kann das Gitternetz eines 3D-Modells nahezu beliebig stark unterteilt werden, je nach gewählter Implementation und vorhandener Hardware. Dabei kommt der Nutzen des dichteren Gittermodells des Objektes erst richtig durch das Nutzen von Normal- oder Displacement-Maps zustande. Diese nutzen die feinteilige Aufteilung und verschieben die Geometrie des Objektes entsprechend der Daten, sodass Tiefeninformationen dem Objekt hinzugefügt werden können.

Abbildung 2 Level of Detail von drei Separaten Modellen
Quelle: Eigene Darstellung

3 Grundlagen

Um die Funktion des Tessellation-Shader erklären zu können, müssen zunächst einige grundlegende Begrifflichkeiten und Funktionsweisen erläutert werden.

3.1 Rendering-Pipeline

Der prinzipielle Ablauf der Render Pipeline kann wie folgt beschrieben werden. Der Hauptprozessor (CPU) sendet Steuerbefehle und die Geometrie-Daten des 3D-Modell an die Grafikkarte. In einem ersten Schritt werden die Eckpunkte (Vertices) der Geometrie-Daten transformiert, was etwa bei der Drehung von dem Objekt notwendig ist. Als Nächstes können die Daten an den Tessellation-Shader weitergereicht werden, dieser sorgt für eine Anpassung der Auflösung (Feinheit) der Geometrie des Objektes. Anschließend kann mittels des Geometrie-Shader, ähnlich zum Tessellation-Shader, neue Geometrie aus vorhandenen Primitiven erzeugt werden. Im nächsten Schritt werden die Eckpunkte mittels des Fragment-Shaders wieder zu Dreiecksflächen zusammengefügt. Als letzten Schritt werden die Flächen entsprechend der gewählten Shading Art, Flat-Shading eine Farbe je Fläche, Gouraud-Shading Interpolation durch die Farbwerte der Eckpunkte der Fläche oder Phong-Shading welcher den Farbwert für jeden Bildpunkt (Pixel) berechnet, eingefärbt und auf dem Bildschirmdargestellt.

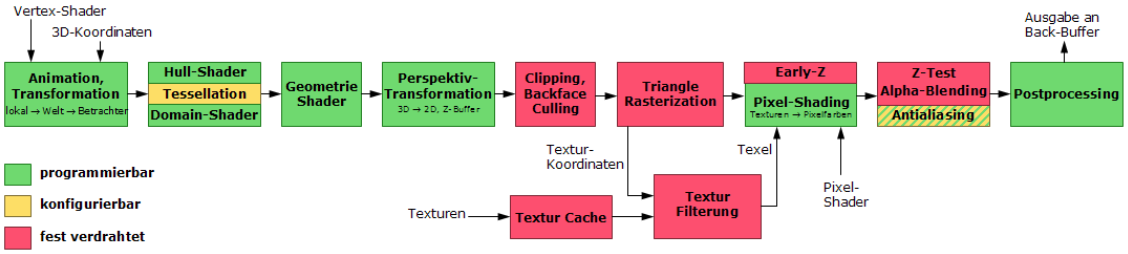

Abbildung 3 Rendering-Pipeline
Quelle: https://de.wikipedia.org/wiki/Grafikpipeline (19.09.2019)

3.2 Shader

Als Shader werden die Teile des Grafikchips benannt, welche an der Verarbeitung der Objektinformationen beteiligt sind. Diese lassen sich in aktuellen Grafikkarten zu einem Großteil frei Programmieren über die Programmierschnittstellen (API) OpenGL und Direct3D.

3.3 Vertex- und Fragment-Shader

Die Vertex- und Fragment-Shader gehören zu den klassischen programmierbaren Shader-Stufen.

Dabei dient der Vertex-Shader dazu die Geometrie, genauer die Eckpunkte dieser, zu verschieben zum Beispiel bei der Bewegung von Objekten. Die Bearbeitung erfolgt dabei für jeden Eckpunkt einzeln.

Der Fragment-Shader, kümmert sich um das Aussehen des Objektes. Dabei können unterschiedliche Arten gewählten werden wie das Objekt texturiert bzw. eingefärbt wird. Als Flat-Shading wird dabei der Vorgang beschrieben anhand der Farbe eines Eckpunktes die gesamte Fläche in der Farbe einzufärben. Das Gouraud-Shading nimmt einen interpolierten Farbwert aus den Eckpunkten einer Dreiecksfläche und färbt diese entsprechend ein. Phong-Shading ermittelt für jeden Bildpunkt über den Normalenvektor und die einfallende Beleuchtung den Farbwert und färbt den Bildpunkt entsprechend ein.

Mit der Einführung der Unified-Shader wurde die klassische Aufteilung in Vertex- und Fragment-Shader eingestellt. Der Unified-Shader kann beide Aufgaben übernehmen und entsprechend nach Bedarf frei programmiert werden.

3.4 Diffuse-Map

Als Diffuse-Map wird die Textur, welche die Farbwerte für die einzelnen Punkte des Objektes beinhaltet bezeichnet.

3.5 Normal-Map

Normal-Maps erzeugen eine Illusion von Tiefendetails auf der Oberfläche des Objekts. Dabei kommen RGB-Informationen welche die X-, Y- und Z-Ache im 3D-Raum widerspiegeln zum Einsatz. Diese Informationen geben der Anwendung die Ausrichtung der Flächen des Objektes an, sodass die Schattierung entsprechend korrekt dargestellt wird.

3.6 Displacement-Map

Anders als bei der Normal-Map wird bei der Displacement-Map die Geometrie des Objektes verändert. Dafür ist die Unterteilung der Geometrie durch den Tessellation-Shader notwendig. Anhand der 16- oder 32-Bit Graustufenwerte werden die einzelnen

Punkte entsprechend verschoben. Das Ergebnis ist dabei überzeugender als das der Normal-Map erfordert aber auch einen erhöhten Rechenaufwand.

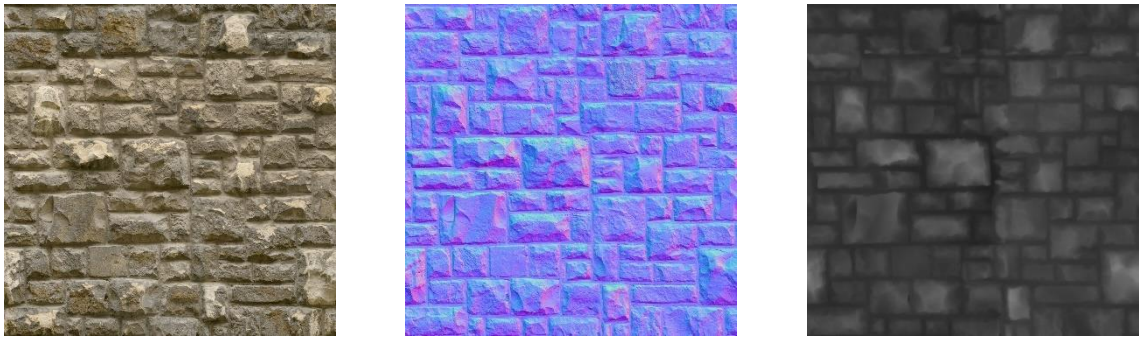

Abbildung 4 Links nach rechts: Diffuse-, Normal-, Displacement-Map
Quelle: https://texturehaven.com/tex/?c=bricks&t=rough_block_wall (19.09.2019)

4 Tessellation Funktionsweise

Als Tessellation wir der optionale Schritt in der Render-Pipeline beschrieben in dem ein Patch (Fläche) der Geometrie eines Objektes unterteilt wird.

Mittels des Tessellation-Control-Shaders wird der Grad der Tessellation festgelegt, dieser wird auch genutzt, um Flächen mit unterschiedliche starker Tessellation so zu gestalten das die Schnittkanten die gleiche Tessellation-Stufe aufweisen.

Der Tessellation-Primitive-Generator führt die Tessellation aufgrund des TCS oder von Standartwerten aus.

Mit dem letzten Shader-Schritt dem Tessellation-Evaluation-Shader, wird für den übergebenen Patch die neu generierten Vertex-Punkte die genauen Werte berechnet.

4.1 Patches

Der Tessellation Vorgang benötigt Patches zum Arbeiten, diese bestehen aus Primitiven Typen. Als primitive typen können Isolines (Linien), Triangles (Dreiecke) oder Quads (Rechtecke) angesehen werden. In einer Liste mit den Vertices (Eckpunkten) eines Objektes bilden dementsprechend alle 2, 3 bzw. 4 Vertices einen neuen Patch.

4.2 Tessellation-Control-Shader (TCS)

Der TCS ist die optionale Stufe, die es ermöglicht die Anzahl der Tessellation die ein Patch haben soll festzulegen, zudem können Transformationen auf die Patches angewendet werden.

Es besteht zudem die Möglichkeit die Größe von Patches zu ändern, dabei kann immer nur ein Patch verarbeitet werden jedoch keins ausgelassen werden zu mindestens nicht im TCS.

4.3 Tessellation-Primitive-Generator (TPG)

Der TPG erzeugt neue primitive von dem ihm übergebenen Patch. Das ist die einzige Aufgabe die er übernimmt, wodurch er auch nicht programmierbar ist. Dabei wird die Funktion nur ausgeführt, wenn auch der Tessellation-Evaluation-Shader vorhanden ist.

Für den TPG sind, die ihm übergeben Patches abstrakt, dies bedeutet das der TPG nur das jeweilige Patch kennt und keine übergreifenden Informationen zur gesamten Geometrie des Objektes hat.

Die Stärke der Tessellation für die abstrakten Patches definiert ein 4-Vektor großes Array für die äußeren Tessellation Stufen und ein 2-Vektor großes Array für die inneren Tessellation Stufen. Dabei nutzen die primitiven unterschiedliche Werte der Arrays, das Quad nutzt alle Werte, während das Triangle nur drei Werte des äußeren Arrays und einen Wert des inneren Arrays nutzt. Dabei sind in dem Array die Werte für die Unterteilung der jeweiligen Seitenkante bzw. inneren Fläche festgelegt. Ein Tessellation Wert von 3 erzeugt unterteilt eine Kante sodass diese nun aus 3 Kanten 4 Eckpunkten.

In diesem Schritt können auch Patches ignoriert werden, in dem für den Tessellation Level null oder ein negativer wert gesetzt wird.

Die maximale Tessellation Stufe wird von der Grafikkarte ermittelt und der gesetzte wird in den inner- bzw. outer-Arrays wird entsprechend der Tessellation-Methode zwischen dem minimal und dem Maximalwert der Grafikkarte festgelegt.

Die Kanten in drei verschiedenen Methoden aufgeteilt werden. Die equal_spacing Methode, unterteilt die kannte gleichmäßig, sodass alle Kanten die gleiche Länge haben. Mittels fractional_even_spacing und fractional_odd_spacing lässt sich ein sanfterer Tessellation Effekt erzielen. Dies ist besonders für dynamisches anhand vom Abstand zur Kamera nützlich. Dabei wird einer bestimmten Kantenlänge eine Unterteilung eingefügt.

Wenn das Patch aus einem Dreieck (Triangle) besteht, werden die neu generierten Eckpunkte in einem baryzentrischen Koordinatensystem gespeichert. Dieses gibt die Koordinaten der Eckpunkte in der Dreiecksfläche als Kombination der Gewichtung der drei Eckpunkte an. Die Eckpunkte werden dabei als U, V und W bezeichnet. Die Gewichtung der Eckpunkte erhöht sich und gleichzeitig nimmt die Gewichtung der anderen Eckpunkte ab, wenn sich ein neuer Eckpunkt diesen nähert.

Die Tessellation basiert darauf die Kante anhand der inneren Ebene zu Unterteilen. Dies bedeutet das vom inneren Punkt sich konzentrische Ringe aus inneren Dreiecken aufbauen. An jeder Ecke des äußeren Dreiecks werden die zwei benachbarten unterteilten Eckpunkte genommen. Aus diesen beiden wird ein Scheitelpunkt berechnet. Senkrechte Linien von den verbleibenden Eckpunkten zu den Kanten des inneren Dreiecks definieren, wo jede Kante des neuen Dreiecks unterteilt ist. Dieser Vorgang wird unter Verwendung des neuen Ringdreiecks wiederholt, um den nächsten inneren Ring zu erzeugen, bis eine von zwei Endbedingungen erfüllt ist. Wenn der Ring keine unterteilten Kanten hat (nur 3 Eckpunkte), stoppt der Prozess. Wenn der Dreiecksring genau 2 unterteilte Kanten hat (nur 6 Eckpunkte insgesamt), ist der von ihm erzeugte "Ring" überhaupt kein Dreieck, sondern es wird ein einzelner neuer Eckpunkt erzeugt. Durch diesen Algorithmus beträgt die Anzahl der inneren Dreiecksringe die

Hälfte der effektiven inneren Tessellations-Stufe. Nach der Herstellung dieser Innenringe wird die Tessellation für das äußere Hauptdreieck vollständig verworfen. Es wird dann in Übereinstimmung mit den drei effektiven äußeren Tessellierungs-Stufen erneut tesselliert.

Aus den neu erzeugten Eckpunkten müssen als Nächstes wieder Dreiecksflächen erzeugt werden. Die Umsetzung wie dies geschieht hängt von der Implementierung ab, erfordert aber das Einhalten gewissen Regeln. Die Tessellation muss die gesamte Fläche des ursprünglichen Dreiecks abdecken und es darf keine Überlappung von Dreiecksflächen geben.

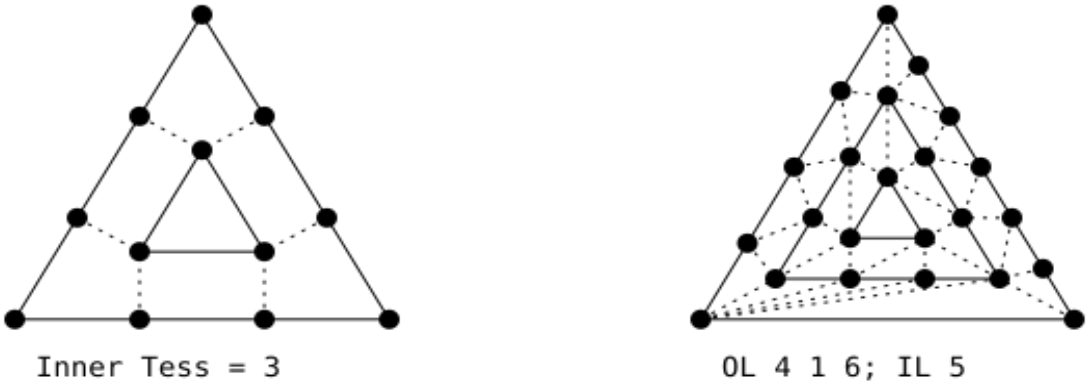

Abbildung 5 Beispielhafte Tessellierung
Quelle: https://www.khronos.org/opengl/wiki/Tessellation

4.4 Tessellation-Evaluation-Shader (TES)

Der TES bekommt die abstrakten Koordinaten von dem TPG und die Ausgabe vom TCS bzw. Vertex-Shader, wenn der TCS nicht verwendet wird und berechnet die tatsächlichen Werte für die Eckpunkt. Darunter fallen die Werte für die Position, normalen, Koordinaten der Eckpunkte. Wenn der TES nicht gesetzt wird, wird auch keine Tessellation ausgeführt.

5 Implementierung als Shader in Unity

In Unity wird für die Verwendung der Tessellation Funktion entsprechende Hardware vorausgesetzt. Dies wird in Unity durch das setzen des „Shader Compilation Target Levels“ auf mindestens das target level 4.6 gewährleistet. Dies wird von Grafikeinheiten welche DirectX 11 + Shader Model 5.0, OpenGL 4.1+, OpenGL ES 3.1+AEP, Vulkan, Metal, PS4/XB1 Konsolen unterstützt. Das setzen des Target Level auf eine bestimmte Version ermöglicht das Nutzen verschiedener aktueller Funktionen von Grafikkarten schließt jedoch die Verwendung von älterer Hardware aus. Dies erfordert eine entsprechende Anpassung an das Programm am Beispiel der Tessellation Funktion müsste auf die Verwendung von unterschiedlichen LOD Modellen zurückgegriffen werden, wenn eine Kompatibilität zu älterer Hardware gewährleistet werden soll.

Um den Shader zu implementieren wird wahlweise ein neues Unity Projekt erstellt oder ein bestehendes Projekt geöffnet, in dem der Tessellation Shader erstellt werden soll.

In dem Unity Projekt wird in dem „Assets folder“ über das Kontextmenü (Rechte Maustaste) oder die Menubar am oberen Bildschirmrand ein neuer „Standard Surface Shader“ erstellt, es empfiehlt sich für die Shader einen eigenen Ordner namens Shader anzulegen, dies sorgt für eine bessere Projektstruktur und mehr Übersicht. Sobald der Shader angelegt wurde, kann ein neues Material, welches den Shader verwendet erstellt werden. Dazu wird wieder mittels des Kontextmenü oder der Menubar ein neues Material erzeugt. Dafür sollte ebenfalls ein eigener Ordner Namens „Materials“ erzeugt werden. Anschließend muss das Material noch der selbsterstellte Shader zugewiesen werden. Dafür wird das Material ausgewählt und im Inspektor und über den Shader Menüpunkt aus der Untergruppe Customs der Shader ausgewählt. Die Vorbereitung sind nun abgeschlossen den Shader mit der Tessellation Funktion zu schreiben. Dafür wird der Shader mittels eines Doppelklicks geöffnet, Unity sollte die in den Optionen festgelegte Entwicklungsumgebung starten (standardmäßig Visual Studio) und es kann begonnen werden den Shader zu schreiben.

Nachfolgend wird beispielhaft ein Edge-Length-Based Tessellation-Shader Aufbau genauer erläutert, welcher auch im beiliegenden Beispiel Programm seine Verwendung findet.

In der Entwicklungsumgebung kann der Name und eine mögliche Gruppierung für den Shader angegeben werden. Dazu wird nach dem Schlüsselwort Shader in Anführungszeichen zunächst eine optionale Gruppierung z.B. Custom und danach getrennt mit einem Schrägstrich der eigentliche Name festgelegt dies dient der besseren Übersicht bei der Auswahl der Shader-Variante für ein Material.

Als ersten Schritt werden die Eigenschaften (Properties) für den Shader festgelegt. Diese stellen eine Schnittstelle zu Unity dar, ähnlich wie bei den Skripten, damit der Shader auch außerhalb der Entwicklungsumgebung beeinflusst werden kann. Als Besonderheit sei angemerkt das nach einem Befehl kein Semikolon als Trennzeichen verwendet wird. Der Grundlegende Aufbau einer Eigenschaft wird wie folgt beschrieben. Mittels eines unterstrich und nachfolgend in CamelCase-Notation wird der intern genutzte Eigenschaftsname deklariert, anschließend folgt in runden Klammern in Anführungszeichen der Name, der im Inspektor-Fenster in Unity angezeigt wird gefolgt vom Typ der Eigenschaft. Als letzten Schritt muss ein Initialwert für die Eigenschaft festgelegt werden.

Für den verwendeten Edge-Length-Based Tessellation-Shader benötigt man mehrere Eigenschaften.

```
Properties
{
   _DiffuseTexture ("Diffuse Texture", 2D) = "white" {}
   _NormalMap ("Normal Map", 2D) = "bump" {}
   _DisplacementTexture ("Displacement Texture", 2D) = "gray" {}
   _DisplacementFactor ("Displacement Factor", Range(0,1.0)) = 1.0
   _EdgeLength ("Edge length", Range(2,50)) = 2
   _Color ("Color", color) = (1,1,1,0)
   _SpecColor ("Spec color", color) = (0.5,0.5,0.5,0.5)
}
```

Der obere Quellcodeabschnitt zeigt die verwendeten Eigenschaften. Darunter fallen die Texturen für Diffuse und Displacement und die Normal Map, welche das Finale aussehen des Objektes beeinflussen. Außerdem zwei Steuerelemente, mit der EdgeLength Eigenschaft lässt sich später die Ziellänge der Kanten festlegen und der DisplacementFactor legt den Einfluss der Displacement Textur auf die Verschiebung der Eckpunkte fest. Mittels der Color und SpecColor Eigenschaft lässt sich zum einen das Aussehen einfärben und die Farbe der Glanzpunkte Festlegen.

```
CGINCLUDE

       float CalcEdgeTessFactor (float3 worldposition0, float3
             worldposition1, float edgeLength)
       {
           float dist = distance(0.5 *
             (worldposition0+worldposition1),
             _WorldSpaceCameraPos);
           float length = distance(worldposition0, worldposition1);
           float factor = max((length * _ScreenParams.y /
             (edgeLength * dist)), 1.0);
           return factor;
       }



    float4 EdgeLengthBasedTess (float4 vertice0, float4 vertice1,
```

```
            float4 vertice2, float edgeLength)
            {
        float3 position0 = mul(unity_ObjectToWorld,
          vertice0).xyz;
        float3 position1 = mul(unity_ObjectToWorld,
          vertice1).xyz;
        float3 position2 = mul(unity_ObjectToWorld,
          vertice2).xyz;
        float4 tessellation;

        tessellation.x = CalcEdgeTessFactor(position1,
          position2, edgeLength);
        tessellation.y = CalcEdgeTessFactor(position2,
          position0, edgeLength);
        tessellation.z = CalcEdgeTessFactor(position0,
          position1, edgeLength);
        tessellation.w = (tessellation.x + tessellation.y +
          tessellation.z) / 3.0f;

        return tessellation;
    }
ENDCG
```

Bevor der Shader geschrieben werden kann müssen zwei Funktionen definiert werden, die für die Berechnung des Faktors der Kantenunterteilung und die Berechnung für die Tessellation-Funktion festlegen. Dazu erstellt man einen Block, der mit CGINCLUDE beginnt und mit ENDCG aufhört, dies teilt der Grafikkarte mit, alles was drinsteht als Funktionen aufgenommen werden soll. Dies ist nötig, da der Shader ansonsten linear abgearbeitet wird und kein Aufruf von Funktionen möglich wäre. So kennen der Shader bzw. die Grafikkarte die angegeben Funktionen.

```
SubShader {
    Tags { "RenderType"="Opaque" }
    CGPROGRAM
    #pragma surface surfaceFunction BlinnPhong addshadow
             fullforwardshadows vertex:displacement
             tessellate:tessellateEdgeLenght nolightmap
    #pragma target 4.6
```

Mit dem Schlüsselwort SubShader wird die Funktionsweise des Shaders eingeleitet. Zunächst wird über den Tags Parameter der Rendertyp des Shaders mitgeteilt. Dieses erlaubt auch das Beeinflussen an welchem Punkt der Render für das Objekt auf der Grafikkarte ausgeführt wird, das ist besonders bei Transparenz Effekten wichtig. Durch den CGPROGRAM Block wird der Quellcode geschrieben, der auf der GPU ausgeführt wird geschrieben. Damit Tessellation genutzt werden kann, muss die Hardwarekompatibilität sichergestellt werden. Dafür wird mittels #pragma das Kompilierziel für den Shader-Compiler auf Version 4.6 gesetzt, dadurch wird

sichergestellt das die Grafikkarte OpenGL 4 bzw. DX11 unterstützt. Zusätzlich werden einige Funktionen über einen weiteren #pragma Parameter festgelegt, die der Shader nutzen soll. Darunter die Tessellation Funktion, ein Surface-Shader und die Vertex Funktion für das Displacement Mapping, des Weiteren werden einige Optionen gesetzt wie die Schattierungsmethode BlinnPhong und das Erstellen einer Schatten-Map für den Surface-Shader mittels addshadow und fullforwardshadows.

```
struct patchData {
    float4 vertex : POSITION;
    float4 tangent : TANGENT;
    float3 normal : NORMAL;
    float2 texcoord : TEXCOORD0;
};

float _EdgeLength;

float4 tessellateEdgeLenght (patchData vertice0, patchData vertice1,
   patchData vertice2)
{
    return EdgeLengthBasedTess(vertice0.vertex, vertice1.vertex,
           vertice2.vertex, _EdgeLength);
}
```

Als nächstes empfiehlt es sich eine Struktur zu erstellen welche alle zu verarbeitende Eingabe Daten aufweist. Zudem muss nun begonnen werden die am Anfang definierten Eigenschaften für den Shader verfügbar zu machen. Dazu wird einfach der Typ festgelegt und dahinter der Eigenschaftenname geschrieben, zu beachten gilt das der Name übereinstimmen muss der typ jedoch auch abweichen kann.

Es folgt nun der Schritt der Tessellation, dazu wird aus der Liste der Eckpunkte jeweils drei Stück ausgewählt und zusammen mit dem Wert für die gewünschte Kantenlänge an die Entsprechende eingangs definierter Funktion übergeben. Dabei wird sich die Tatsache zunutze gemacht das die Flächen und die Eckpunkte dieser nacheinander in einer Liste gespeichert sind. Als Rückgabe wird eine tessellierte Fläche zurückgegeben.

```
sampler2D _DisplacementTexture
float _DisplacementFactor

void displacement (inout patchData vertice)
{
  float displaceEffect = tex2Dlod(_DisplacementTexture,
  float4(vertice.texcoord.xy,0,0)).r * _DisplacementFactor;
  vertice.vertex.xyz += vertice.normal * displaceEffect;
}
```

Der nächste Schritt besteht darin, mittels der Displacement Textur die Position der Eckpunkte der Flächen zu verschieben. Erst dadurch zeigt sich wirklich der Vorteil von

Tessellation. Da außer der fein zerteilten Geometrie sonst keine Veränderung sichtbar wäre. Dazu wird die Displacement Textur geladen und in die UV-Koordinaten beginnend bei 0,0 die Werte des Rotkanals ausgelesen. Anschließend werden diese mit dem Displacement Faktor multipliziert, was den Displacement Effekt für jeden Eckpunkt ergibt. Diese werden dann entlang der Normalen entsprechend verschoben.

```
struct Input {
  float2 uv_DiffuseTexture;
};

sampler2D _DiffuseTexture;
sampler2D _NormalMap;
fixed4 _Color;

void surfaceFunction (Input input, inout SurfaceOutput output) {
  float4 c = tex2D (_DiffuseTexture, input.uv_DiffuseTexture) * _Color;
  output.Albedo = c.rgb;
  output.Specular = _SpecColor;
  output.Gloss = 1.0;
  output.Normal =UnpackNormal(tex2D
          (_NormalMap,input.uv_DiffuseTexture));
}
ENDCG // Ende des Codeblockes
}
FallBack "Diffuse"
```

Abschließend muss noch dafür gesorgt werden das der Shader auf dem Objekt die Diffuse Textur anzeigt. Dafür wird zunächst eine neue Struktur erstellt, die die UV-Koordinaten der DiffuseTextur beinhaltet und als Nächstes die Diffuse Textur und Normal Map geladen bzw. für den Shader initialisiert. In der surfaceFunction Funktion wird ein einfacher beleuchteter Shader erstellt.

Mittels ENDCG wird der Quellcode Block abgeschlossen, als Notlösung, wenn ein Fehler im Shader auftritt oder eine Grafikkarte die Funktionen nicht unterstützt wird mittels Fallback “Diffuse“ eine Ausweichmöglichkeit gegeben das dennoch etwas angezeigt wird.

Damit ist der Shader fertiggestellt und kann genutzt werden, dies stellt aber nur eine Möglichkeit dar einen Tessellation Shader zu erstellen es können noch andere Arten erstellt werden wie zum Beispiel die FixedTessellation oder PhonTessellation Umsetzung.

6 Zusammenfassung

Mit dem Tessellation-Shader wurde ein Thema gewählt, bei dem die Funktionsweise zwar in wenigen Worten zusammengefasst werden kann, Render-Pipeline Stufe bei der die Flächen in kleinere primitive Flächen unterteilt werden, jedoch in seiner Auswirkung und Umsetzung viele Möglichkeiten eröffnet.

Dabei ist auch an der Tessellation Funktion der Trend zu erkennen, dass für viele Aufgaben die früher von Hand erledigt werden mussten immer mehr Funktionen und Prozesse entstehen die automatisiert erledigt werden können. Darunter fällt z.B. das Erstellen von Texturen oder ganzen Spiele Leveln, die sich mittels Prozeduralen Algorithmen automatisiert erstellen lassen. Der Tessellation-Shader ermöglicht es das Erstellen von verschiedenen 3D-Modellen von einem Objekt per Hand zu automatisieren und dynamisch zu gestalten.

Das Modul GPU-Programmierung hat für das Erstellen des Shaders in Unity genügend Vorwissen vermittelt, so das auch dass nutzen einer bisher unbekannten Shadersprache in Form von CG / HLSL mithilfe der angegeben Quellen das Umsetzen bewerkstelligt werden konnte.

Als größtes Problem bei der Bearbeitung der Belegarbeit war das Zeitmanagement und eine Unterschätzung des Aufwandes zur Bearbeitung des Themas. Dies äußert sich vor allem in der Umsetzung des Beispielprogrammes wo noch zusätzliche Beispielszenen geplant waren und die Anzeige des Tessellation Effektes als Wireframe angezeigt werden sollte. Die Umsetzung des Wireframe-Effekts gestaltete sich dabei als schwierig bzw. liefert nicht das gewünscht Resultat. Auch die schriftliche Ausarbeitung hätte noch etwas mehr Zeit gebrauchen können.

Dennoch lässt sich ein positives Fazit für die Belegarbeit ziehen, da es sich um ein interessantes Thema handelt und gut den Fortschritt in dem Bereich der Grafik Technologien aufzeigt.

Für zukünftige Arbeiten an dem Projekt wäre das Experimentieren mit anderen Tessellation Methoden denkbar und eine Analyse der Auswirkung bzw. aufzeigen der Unterschiede in z.B. Leistung und Speicherverbrauch bei der Verwendung von klassischen Level of Detail Modellen gegenüber der Tessellation Funktion denkbar.

7 Quellenverzeichnis

Tessellation Subdividing Triangles, Jasper Flick:

https://catlikecoding.com/unity/tutorials/advanced-rendering/tessellation/ (19.09.2019)

Real-time Rendering Techniques with Hardware Tessellation, M. Nießner et. Al. (2015):

https://graphics.stanford.edu/~mdfisher/papers/realtimeRendering.pdf (19.09.2019)

A Gentle Introduction to Shaders, Alan Zucconi:

https://unity3d.com/de/learn/tutorials/topics/graphics/gentle-introduction-shaders

(19.09.2019)

Direct3D 11 In-Depth Tutorial: Tessellation, Bill Bilodeau (2010):

https://www.gdcvault.com/play/1012740/Direct3D-11-In-Depth-Tutorial (19.09.2019)

Tessellation OpenGL, Original Autor Alfonse:

https://www.khronos.org/opengl/wiki/Tessellation (19.09.2019)

Tutorial 30: Basic Tessellation, Etay Meiri:

http://ogldev.org/www/tutorial30/tutorial30.html (19.09.2019)

8 Selbständigkeitserklärung

Hiermit bestätige ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der Arbeit, die dem Wortlaut oder dem Sinn nach anderen Werken (dazu zählen auch Internetquellen) entnommen sind, wurden unter Angabe der Quelle kenntlich gemacht.

Weinböhla, 19.09.2019 

Ort, Datum Unterschrift