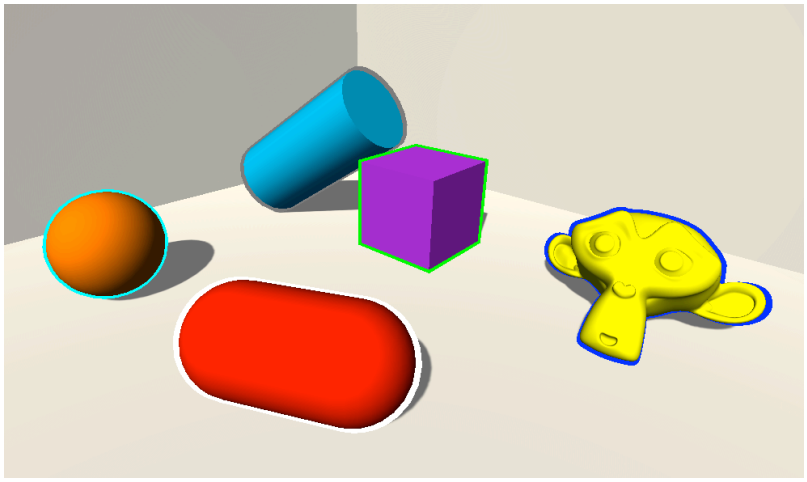
MEDIEN INFORMATIK

Hochschule Harz
Hochschule für angewandte Wissenschaften

Dokumentation
Kontur-Shader



BFO – Ausgewählte Themen der Shaderprogrammierung

Autor: Philipp Schneider (m24335)

Dozent: Alexander Johr

Fachsemester: 8

Abgabedatum: 31.08.2020

Inhaltsverzeichnis

Abbildungsverzeichnis

1. Idee

Meine Idee war es, einen 3D-Shader zu erstellen, wo die einzelnen Objekte eine Kontur bekommen sollten, sodass es einem Spieler zeigen könnte, dass er oder sie mit diesem Objekt interagieren kann. Zudem wollte ich einbauen, dass wenn man mit der Maus über ein Objekt drüber fährt sich die Farben von Kontur und Objekt spiegeln. Dies war mein Grundgedanke für die Abgabe. Da ich zugegebenermaßen nicht der beste Programmierer bin, habe ich mich bewusst für einen in meinen Augen, etwas einfacheren Shader entschieden.

2. Umsetzung

Nachdem ich ein neues Unity Projekt geöffnet hatte, erstellte ich mit drei Planes zwei Wände und einen Boden und fügte verschiedene Formen ein um eine kleine Szene zu bauen, an der ich später den Shader testen wollte. Danach legte ich im Assets-Ordner einen neuen Ordner namens Shader an und erzeugte darin einen neuen Unlit Shader, wies ihm ein Material zu und öffnete ihn in Visual Studio.

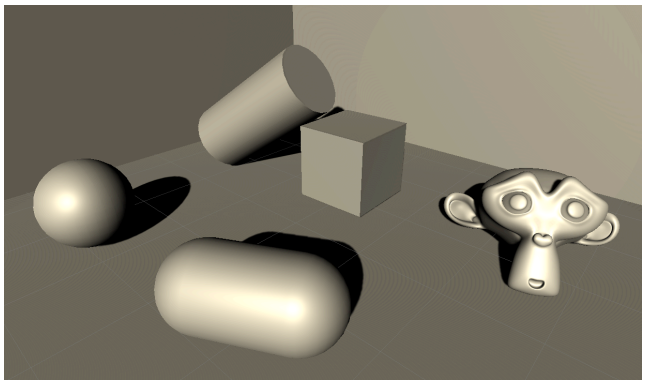


Abb. 1: Szene ohne Shader mit Ausgangsmaterial

Im bereits vorgegebenen Code löschte ich erst einmal alles, was für mich unwichtig war. Übrig blieb somit nur der Shader an für sich und die „Properties“ sprich die Eigenschaften. Sie dienen dem späteren Input für die Funktionen des Shaders. Darin gab ich erst einmal eine „Main Texture“ an. Sie zeigt an, dass eine Eigenschaft die Haupttextur für ein Material ist.

Als Grundfarbe legte ich Grau fest. Zudem benötigte ich für meine spätere Kontur genauso eine Farbe wie zusätzlich noch eine Weite, um sie später auch in der Dicke zu manipulieren. Ich wählte Schwarz als Ausgangsfarbe für die Kontur und einen Bereich von 1,0 bis 1,1 (mit einem Ausgangswert von 1,01) für die Weite. Diese Angaben konnte man dann schon im Inspector des zuvor erzeugten Materials sehen.

```
Properties
{
    _MainTex ("Texture", 2D) = "white" {}
    _Color("Main Color", Color) = (0.5,0.5,0.5,1)
    _OutlineColor("Outline color", Color) = (0,0,0,1)
    _OutlineWidth("Outline width", Range(1.0,1.1)) = 1.01
}
```

Abb. 2: Grundwerte des Shaders

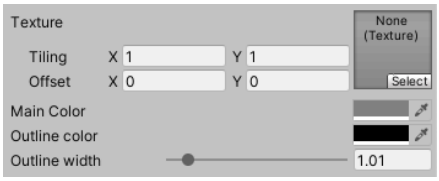


Abb. 3: Ansicht der Einstellmöglichkeiten im Inspector

Unter den Properties deklarierte ich dann einen Sub-Shader, also einen Unter-Shader. Jeder Shader in Unity besteht aus einer Liste von Sub-Shadern. Wenn Unity ein Mesh anzeigen soll, findet es den zu verwendenden Shader und wählt den ersten Sub-Shader. Ein Sub-Shader definiert eine Liste von Rendering-Durchgängen und kann optional jeden beliebigen Zustand einstellen, der bei allen Durchgängen (Passes) gleich ist. Zusätzlich können subshaderspezifische Tags eingerichtet werden. Es kann in einem Sub-Shader auch mehrere Passes geben, welche dann von Unity nacheinander gerendert werden. In meinem Fall gibt es zwei Passes. Einen für die Kontur und einen für das Objekt. In diesen Sub-Shader habe ich dann einen Pass (für die Kontur) mit den Statements: „CGPROGRAM“ und „ENDCG“ eingefügt. Diese Statements sind essenziell für Shader, da Unity für Grafiken seine eigene Script-Sprache (ShaderLab) hat. Darüber legte ich noch in der Render Queue fest, dass es als Transparent gerendert werden solle, um so über dahinter liegenden Objekten angezeigt zu werden. Zudem fügte ich darüber noch ein ZWrite Off ein. Dies steuert, ob Pixel von diesem Objekt in den Depth-Buffer geschrieben werden.

Bei „festen“ Objekten wird dieser in der Regel ein- und bei halbtransparenten ausgeschaltet. Nach dem CGPROGRAM-Statement folgen erst einmal die Deklarationen für die Vertex-Funktion und die Fragment-Funktion. Die Vertex-Funktion dient zum „bilden“ des Objekts, bzw. dem berechnen der Vertex und über die Fragment-Funktion wird es koloriert. Danach musste ich eine „Bibliothek“ einfügen. Die sogenannte UnityCG. Diese Datei ist oft in Unity-Shadern enthalten. Sie deklariert viele eingebaute Hilfsfunktionen und Datenstrukturen. Als Nächstes folgte die appdata-Struktur worin ich einen float4 vertex und einen float3 normal deklarierte. Die Shader von Unity verwenden Strukturen, um Informationen über die Rendering-Pipeline weiterzuleiten. Die erste Struktur gibt Rohinformationen über die gerenderte Geometrie an den Vertex-Shader weiter. Einmal die Position des Vertex im Raum und seine Normale. Die zweite Struktur (v2f) enthält vom Vertex-Shader erzeugte Informationen, die an den Fragment-Shader übergeben werden. Genauer gesagt die Position und die Normale des Vertex nach dessen Transformation. Darunter deklarierte ich noch ein float für die Weite der Kontur und einen float4 für dessen Farbe. Als nächstes implementierte ich einen Vertex-Shader welche die Position und die Normale aus appdata erhalten und multipliziert mit der x-, y- und z-Koordinate der Weite, diese ergeben (genauere Erklärung siehe Punkt 3). Nachfolgend deklarierte ich den Rückgabeparameter für den v2f-Shader. Die Position transformierte ich dann wieder in eine Kamerakoordinate und gab den Wert zurück. Als letztes im ersten Pass-Abschnitt implementierte ich einen Fragment-Shaderin welchem ich die Konturfarbe zurückgab.

```
Tags { "Queue" = "Transparent" }

ZWrite Off

CGPROGRAM
#pragma vertex vert
#pragma fragment frag

#include "UnityCG.cginc"

struct appdata
{
    float4 vertex : POSITION;
    float3 normal : NORMAL;
};

struct v2f
{
    float4 pos : SV_POSITION;
    float3 normal : NORMAL;
};
```

Abb. 4: Oberer Abschnitt im ersten Pass für Kontur

```
float _OutlineWidth;
float4 _OutlineColor;

v2f vert(appdata v)
{
    v.vertex.xyz *= _OutlineWidth;

    v2f o;
    o.pos = UnityObjectToClipPos(v.vertex);
    return o;
}

half4 frag(v2f i) : COLOR
{
    return _OutlineColor;
}

ENDCG
```

Abb. 5: Unterer Abschnitt im ersten Pass für Kontur

Im nächsten Pass (den für das Objekt), deklarierte ich wieder einen ZWrite, stellte ihn hier aber auf On. Nachfolgend gab ich ein Material mit einer diffusen und einer Umgebungsfarbe an, welche sich beim Einstellen im Inspector auf die Grundfarbe auswirken. Danach stellte ich das Lighting des Shaders an. Ohne dies, wäre das Objekt weiß, egal welche Farbe ich zuvor im Inspector eingestellt hätte. Nachfolgend legte ich eine Textur fest, welche die Grundtextur ist und verwendete „ConstantColor“ um eine Volltonfarbe aus den Eigenschaften in die Texturüberblendung zu bekommen. Anschließend multiplizierte ich noch das Ergebnis aus der vorherigen SetTexture sprich previous mit primary, der Farbe aus der Beleuchtungsberechnung, bzw. die Farbe des Scheitelpunkts aus den Materialien. Dieses Ergebnis wird dann mit zwei multipliziert, um die Beleuchtungsintensität zu erhöhen.
Als letztes musste ich noch nach der schließenden Klammer des Sub-Shader, ein FallBack Statement als diffus deklarieren, damit die einzelnen Objekte auch einen Schatten werfen.

```
ZWrite On

Material
{
    Diffuse[_Color]
    Ambient[_Color]
}

Lighting On

SetTexture[_MainTex]
{
    ConstantColor[_Color]
}

SetTexture[_MainTex]
{
    Combine previous * primary DOUBLE
}
```

Abb. 6: Pass für Objekt

Daraufhin wies ich jedem Objekt ein eigenes Material zu, färbte diese mit einer Kontur nacheinander ein und passte die Größe der Konturen an. Damit ich nun die Farbe ändern konnte, wenn ich mit dem Mauszeiger über die einzelnen Objekte fahren würde, musste ich ein zusätzliches C#-Script schreiben. Leider konnte ich es nicht über den Shader direkt lösen. Ich gab in dem Script an, dass wenn ich mit der Maus über das Objekt fahre das zweite bzw. gegenteilige Material angezeigt werden sollte und wenn dies nicht der Fall ist, sollte das eigentliche Material angezeigt werden. Dann erstellte ich für jedes Material ein gegenteiliges Material, in welchen ich die Farbe der Kontur mit der des Objektes tauschte. Das Script und die zwei jeweiligen Materialien wies ich dann im Inspector jedes Objekts zu.

```
public class Hover : MonoBehaviour
{
    public Material firstMaterial;
    public Material secondMaterial;

    void OnMouseOver()
    {
        GetComponent<Renderer>().material = secondMaterial;
    }

    void OnMouseExit()
    {
        GetComponent<Renderer>().material = firstMaterial;
    }
}
```

Abb. 7: Script für das anzeigen der Materialien

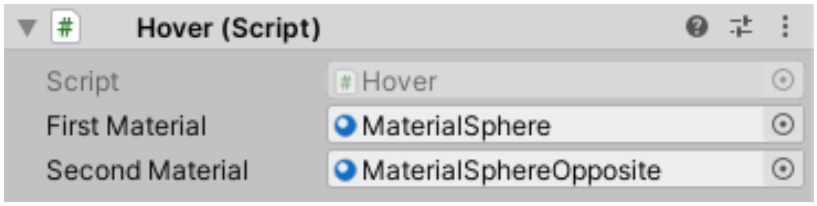


Abb. 8: Ansicht des Scripts im Inspector

Damit war der Shader angeschlossen und die Szene sah nun nicht mehr aus wie in der ersten Abbildung, sondern wie folgt:

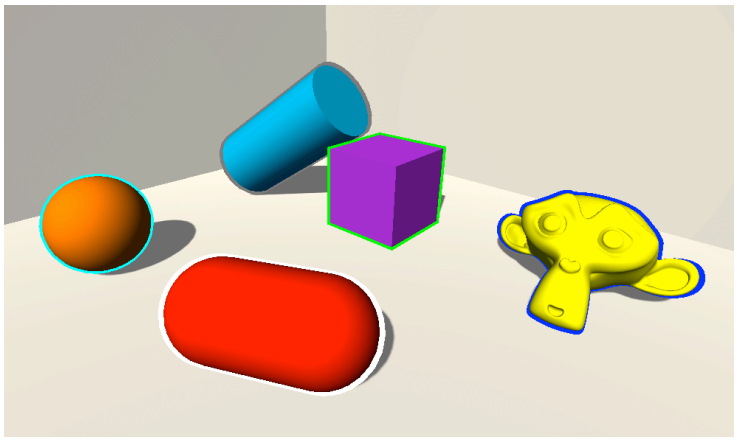


Abb. 9: Ansicht der Szene mit fertigem Shader und Materialien

3. Erklärung

Um zu verstehen, wie der Shader funktioniert, muss man sich noch einmal den Vertex-Shader aus dem Pass für die Kontur genauer ansehen:

```
v2f vert(appdata v)
{
    v.vertex.xyz *= _OutlineWidth;

    v2f o;
    o.pos = UnityObjectToClipPos(v.vertex);
    return o;
}
```

Abb. 10: Vertex-Shader aus Pass für Kontur

Dieser Shader enthält die Position und die Normale aus der Struktur appdata. Die Normale ist im Prinzip ein Vektor der immer Senkrecht zur Oberfläche steht. Und diese Werte werden für die Richtungen der x-, y- und z-Achse des Objekts mit dem eingestellten Wert für die Weite der Kontur multipliziert und als neuer Wert für die Weite eingetragen. Es wird also jeder einzelne, sichtbare Pixel der Oberfläche des Objekts genommen und dann um die tatsächliche Normale „erweitert“. Oder um es grob vereinfacht auszudrücken, könnte man auch sagen, dass das Objekt noch

einmal vergrößert hinter dem eigentlichen Objekt erstellt wird. Ich habe es einmal versucht dies bildlich darzustellen.

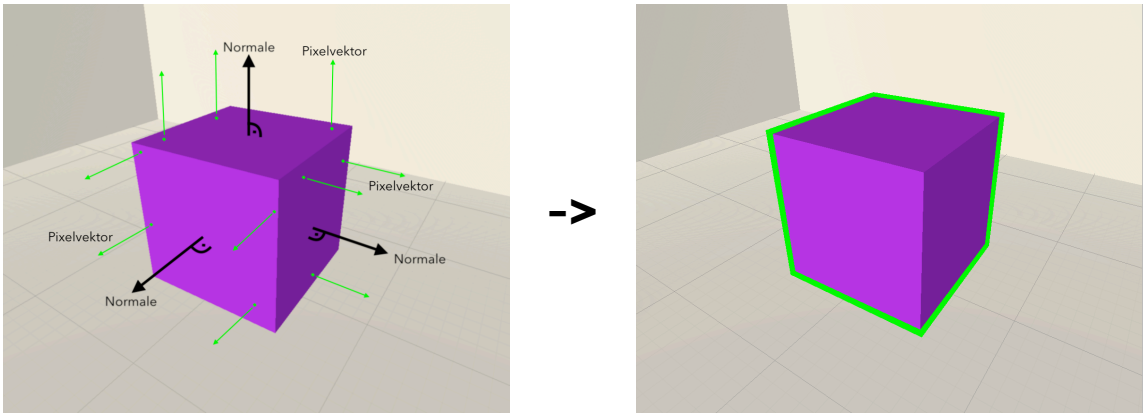


Abb. 11/12: Quader vor und nach der Transformation der einzelnen Pixel

Man könnte sich nun jedoch fragen: Nur woher weiß der Shader, dass er nicht auch die zur Kamera zeigenden Kanten mit einer Kontur versehen soll? Beziehungsweise, warum ragt das Kontur-Objekt hier nicht heraus?

Nun ja, wie ich bereits zuvor erklärt habe, werden in dem Shader Script die Passes nacheinander abgearbeitet. Dadurch, dass der Pass für die Kontur zuerst erzeugt wird und das eigentliche Objekt danach, wird dieses über der Kontur angezeigt. Tauscht man im Script die beiden Pass-Abschnitte, so würden die Objekte wie in Abb. 13 zu sehen, dargestellt.

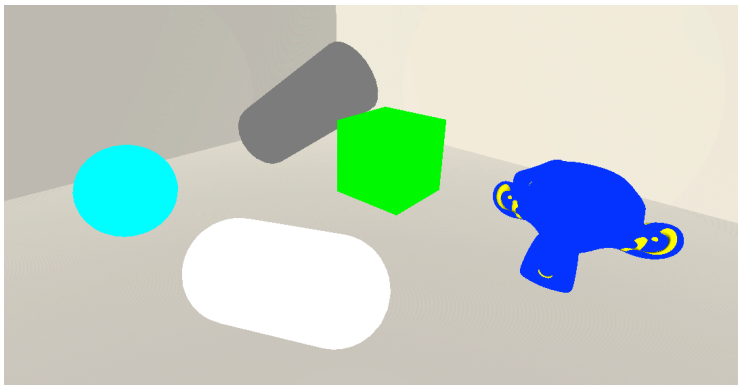


Abb. 13: Ansicht nach vertauschen der Pass-Abschnitte

Nach dem Berechnen der Pixelvektoren musste ich noch, wie bereits erwähnt, die Rückgabeparameter des v2f (vertex to fragment) Shaders von den

Objektkoordinaten in die homogenen Koordinaten der Kamera transformieren. Denn ohne dies würden die Konturen nicht angezeigt werden.

4. Probleme

Als ich den Shader fertig programmiert hatte, sind mir zwei Fehler direkt aufgefallen. Zum einen wirkt die Kontur bei etwas komplexeren Figuren, wie beispielsweise dem Affenkopf, ungleichmäßig. Mal erscheint sie dicker und mal dünner (siehe Abb. 12). Dies hängt damit zusammen, da die beiden übereinander gelegten Formen von Kontur und Objekt sich durch die verschiedenen Größen schneiden können. Ich kann es nicht genau erklären, aber ich nehme an, dass der Shader die Normalen, bzw. Pixelvektoren hier nicht genau berechnen kann.

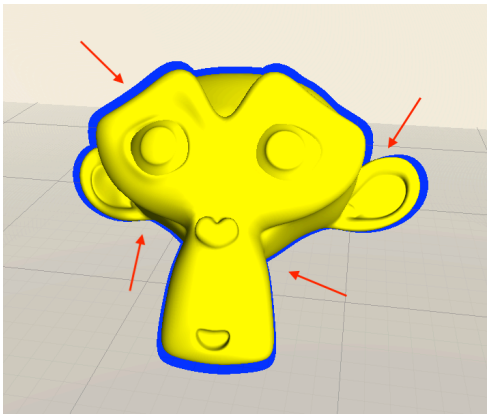


Abb. 14: Teilweise fehlerhafte Darstellung der Kontur

Zudem war mir am Ende aufgefallen, dass die einzelnen Objekte zwar richtig ausgeleuchtet wurden, sprich die Seiten welche zur Lichtquelle gerichtet sind, waren heller, als jene die von dieser fort zeigen. Jedoch warfen die Objekte keine Schatten. Nach langem suchen fiel mir auf, dass ich vergessen hatte einen FallBack nach dem Sub-Shader zu deklarieren. Zumindest war dies die einzige Lösung, die ich finden konnte.
Ansonsten traten zwischendurch beim Programmieren des Shaders immer mal wieder kleine Fehler auf, welche ich meist aber durch Recherche ausbessern konnte.

5. Fazit

Mein Fazit ist etwas gemischt. Auf der einen Seite konnte ich recht viele neue Kenntnisse aus dem Projekt gewinnen, aber auf der anderen Seite bin ich auch etwas enttäuscht, dass ich nur einen recht simplen Shader zustande gebracht habe. Oft fiel es mir schwer, die Formeln von Shadern zu verstehen um diese auch erklären zu können, was sie im Hintergrund eigentlich bewirken.
Ich habe versucht meinen Shader jedoch so ausführlich wie möglich zu erklären, um aufzuzeigen, wie er funktioniert. Dies hat mir aber dabei geholfen, gewisse Vorgänge besser zu verstehen und ich denke, das ist irgendwo auch die Hauptsache.