HOCHSCHULE FURTWANGEN UNIVERSITY | HFU

Bearbeitungsbeginn: 01.03.2021

Vorgelegt am: 30.06.2021

Thesis

zur Erlangung des Grades

Bachelor of Science

im Studiengang Medieninformatik

an der Fakultät Digitale Medien

Luis Keck

Am Rosenhag 2
88709 Meersburg

Matrikelnummer: 256153

Weiterentwicklung des Shader-Systems der Game-Engine FUDGE

Erstbetreuer:	Prof. Jirka Dell'Oro-Friedl
Zweitbetreuer:	Prof. Dr. Dirk Eisenbiegler

Abstract

FUDGE ist eine Game-Engine die unter der Leitung von Prof. Jirka Dell'Oro-Friedl an der Hochschule Furtwangen entwickelt wird und zur Lehre von Studenten, sowie zur schnellen Entwicklung von Prototypen für Spiele genutzt wird.
Diese Arbeit beschäftigt sich damit Shader für FUDGE zu entwickeln und für zukünftige Arbeit an der Game-Engine den Umgang mit dem vorhandenen Shader-System zu erleichtern. Um dies umzusetzen wurden zunächst die vorhandenen Shader und das vorhandene Shader-System untersucht. Dies geschah durch die Betrachtung des Quellcodes und Absprache mit den Entwickler*innen des vorhandenen Codes.
Mit diesem Überblick wurden dann neue Shader geschrieben und Systeme erdacht und umgesetzt, die die Arbeit an Shadern erleichtern. Außerdem wurde damit begonnen ein System zur Modularisierung von Shadern zu entwickeln, das ermöglichen soll Shader auf einfache Weise aus vorhanden Modulen zusammenzusetzen, sowohl in der Game-Engine als auch bei der späteren Entwicklung mit FUDGE.

Inhaltsverzeichnis

1 FUDGE

FUDGE (**F**urtwangen **U**niversity **D**idactic **G**ame **E**ditor) ist eine Open-Source Game-Engine, die unter der Leitung von Prof. Jirka Dell'Oro-Friedl an der Hochschule Furtwangen entwickelt wird. Der geplante Einsatzbereich von FUDGE ist in der Lehre von Studierenden zu den Themen Gestaltung und Entwicklung von Games, sowie ähnlichen interaktiven Anwendungen. Außerdem soll es mit FUDGE möglich sein schnell leicht zugängliche Prototypen zu entwickeln um Ideen für Spiele und Anwendungen auf ihr Potential testen zu können ohne unnötig viel Zeit und Ressourcen zu investieren. Basierend auf diesen Anforderung wurde sich dazu entschieden FUDGE mit Hilfe moderner Web-Technologien zu entwickeln. Dies ermöglicht unter anderem das schnelle erstellen von Builds die in einem Web-Browser getestet werden können. Dabei lassen sich Builds auch auf Server hochladen, sodass sie über das Internet zugänglich gemacht werden können, ohne dass ein Nutzer die Anwendung herunterladen muss oder auf zusätzliche Software angewiesen ist. FUDGE unterscheidet sich auch im Umfang von anderen Game-Engines die zuvor in der Lehre an der Hochschule Furtwangen genutzt wurden. Da das Ziel nicht die Entwicklung vollwertiger Games mit großem Umfang, schöner Grafik und Performance-Optimierung sind, verzichtet FUDGE auf viele Funktionen die bei der Entwicklung von Prototypen nicht notwendig sind. Somit ist die Game-Engine sehr viel übersichtlicher und auch Studierende die noch keine Vorerfahrung mit Spieleentwicklung haben werden nicht von komplexen User-Interfaces und großen Projekten und zahlreichen Funktionen zurückgehalten. Zusätzlich wird bei der Entwicklung von FUDGE darauf geachtet, dass Prozesse und Strukturen im Quellcode möglichst gut lesbar und nachvollziehbar sind, sodass Studierende die Spieleentwicklung lernen nicht nur FUDGE anzuwenden wissen, sondern auch verstehen. FUDGE ist inzwischen weit genug in seiner Entwicklung vorangeschritten, dass es bereits für Lehre an der Hochschule Furtwangen eingesetzt wird. Neben Prof. Jirka Dell'Oro-Friedl sind auch Studierende an der Entwicklung von FUDGE beteiligt, unter anderem auch in Form von Abschlussarbeiten wie dieser. Der Quellcode von FUDGE ist in der Open-Source Programmiersprache TypeScript geschrieben und wird in in JavaScript kompiliert, sodass Projekte direkt in einem Webbrowser gestartet werden können. (vgl. Dell´Oro-Friedl (Stand 08.06.2021): FUDGE Wiki, https://github.com/JirkaDellOro/FUDGE/wiki/Motivation)

2 Ziel der Arbeit

In FUDGE wurde bereits mit der Entwicklung eines Shader-Systems begonnen, dass für die Berechnung der grafischen Darstellung von Objekten in der Anwendung verantwortlich ist. Verschiedene Shadertypen werden definiert von TypeScript-Klassen, die GLSL-Code enthalten, der Vertex- und Fragment-Shader enthält. Dieser ist jedoch nur als String im Quellcode vorhanden, was dazu führt dass die Shader nur schwer lesbar sind und bei der Entwicklung keine Unterstützung durch die Entwicklungsumgebung (Visual Studio Code) stattfindet. Somit werden Syntax-, Schreib- und Logikfehler nicht sichtbar gemacht und fallen oftmals erst während der Laufzeit auf, wodurch der Entwicklungsprozess verlangsamt wird. Ein Ziel dieser Arbeit ist es also den Umgang mit Shaderprogrammen in FUDGE zu erleichtern, sodass Shader lesbarer werden und Entwickler*innen mehr Unterstützung bei der Erstellung neuer oder Veränderung bestehender Shader erhalten.
Zusätzlich soll der Katalog von vordefinierten Shadern im FUDGE-Quellcode erweitert werden, sodass mehr Auswahl bei der Entwicklung von Anwendungen mit FUDGE besteht.
Ein weiteres Ziel war es mit der Modularisierung des Shader-Systems von FUDGE zu beginnen. Es soll möglich sein Shader aus vordefinierten Modulen, also GLSL-Code-Blöcken zusammenzusetzen. Dadurch kann verhindert werden, dass Code-Teile die in verschiedenen Shadern identisch sind, mehrfach im Quellcode auftauchen. Sollten diese editiert werden, muss dies dann auch nur an einer Stelle geschehen, wodurch die Fehleranfälligkeit des Systems verringert wird. Außerdem können Shader-Module genutzt werden um bei der Entwicklung von Anwendungen mit FUDGE neue Shader zu erstellen, die sich von den vordefinierten Optionen unterscheiden und durch neuen GLSL-Code erweitert werden können.

3 Ausgangszustand des Shader-Systems

Die Darstellung von 3D-Grafiken in FUDGE findet durch die JavaScript-Programmierschnittstelle WebGL (Web Graphics Library) statt. Dieser Schnittstelle wird Code übergeben der in GLSL (OpenGL Shader Language), einer C ähnlichen Sprache geschrieben wurde. Es werden zwei Funktionen, der Vertex-Shader und der Fragment-Shader übergeben, die zusammen ein Shaderprogramm bilden. Der Vertex-Shader berechnet die Clip-Space-Position von Vertices die von WebGL genutzt werden um Primitive wie zum Beispiel Punkte, Linien oder Dreiecke zu rastarisieren. Dazu wird der Fragment-Shader genutzt, der für einmal für jeden Pixel des Render-Canvas aufgerufen wird und eine Farbe für diesen berechnet.
Grundlage für Shader in FUDGE ist die statische Superklasse “Shader” die als Repräsentation von WebGL-Shaderprogrammen dient und Attribute sowie Methoden definiert, die von Shadern genutzt werden. Ein Shader in FUDGE wird definiert von einer Klasse, die von “Shader” erbt und mindestens die Werte der Attribute “vertexShaderSource” und “fragmentShaderSource” überschreibt. Diese beiden Attribute enthalten den GLSL-Code der Shader in Form von Strings, die an die Klasse “RenderInjectorShader” weitergegeben werden um diese in WebGL-Shader zu kompilieren, die an den WebGL-Buffer übergeben werden können. Der Aufbau von Vertex- und Fragment-Shader ist der gleiche. Es werden Daten in Form von Attributen, also Daten aus Buffern wie zum Beispiel die Position des Vertex und Uniforms, konstante Werte die für die Berechnungen aller Vertices in einem Draw-Call gleich bleiben. Die eigentlichen Berechnungen finden dann in der Main-Methode des Shaders statt. Hier werden dann die zur Verfügung gestellten Daten genutzt und neue Werte berechnet, die vom Vertex-Shader in den Fragment-Shader zur weiteren Berechnung übergeben und dann wiederum zur Einfärbung der Pixel auf dem Bildschirm genutzt werden.
Zu Beginn dieser Arbeit waren sechs Shader schon in FUDGE vorhanden: “ShaderFlat”, “ShaderMatCap”, “ShaderPick”, “ShaderPickTextured”, “ShaderTexture” und “ShaderUniColor”. Diese werden wie schon erwähnt dargestellt durch eine TypeScript-Klasse die von “Shader” erbt und den GLSL-Code für Vertex- und Fragment-Shader in Form von Strings enthalten. Eine genauere Beschreibung der vorhandenen Shader folgt im Kapitel “Shader in FUDGE”.
Der simpelste Shader in FUDGE ist “ShaderUniColor”. Dieser färbt Objekte einfarbig ein ohne dass diese von Lichtern in der Szene beeinflusst werden, sodass keine Schattierungen entstehen.

In FUDGE sieht dieser Code folgendermaßen aus:

ShaderUniColor.ts:

```
namespace FudgeCore {
  @RenderInjectorShader.decorate
  export abstract class ShaderUniColor extends Shader {
    public static readonly iSubclass: number =

Shader.registerSubclass(ShaderUniColor);

    public static vertexShaderSource: string =
    `#version 300 es
    /**
    * Single color shading
    * @authors Jascha Karagöl, HFU, 2019 | Jirka Dell'Oro-Friedl, HFU,
2019
    */
    in vec3 a_position;
    uniform mat4 u_projection;

    void main() {
        gl_Position = u_projection * vec4(a_position, 1.0);
    }`;

    public static fragmentShaderSource: string =
    `#version 300 es
    /**
    * Single color shading
    * @authors Jascha Karagöl, HFU, 2019 | Jirka Dell'Oro-Friedl, HFU,
2019
    */
    precision mediump float;

    uniform vec4 u_color;
    out vec4 frag;

    void main() {
      frag = u_color;
    }`;
  }
}
```

Hier wird deutlich warum sich der Umgang mit Shadern in FUDGE schwieriger als erwünscht gestaltet. Obwohl es sich nur um wenige Zeilen GLSL-Code handelt, sind sie nur schwer lesbar, da sie von der Entwicklungsumgebung nicht in hilfreicher Form eingefärbt werden wie der restliche TypeScript-Code, da sie eben nur als String vorliegen. Außerdem werden weder Syntax- noch Schreibfehler erkannt, sodass es bei Veränderungen am GLSL-Code schnell Probleme entstehen können die eventuell erst zur Laufzeit einer Anwendung auffallen.

Bedenkt man nun, dass dies der einfachste Shader in FUDGE ist und die anderen mehr GLSL-Code benötigen wird klar warum dieses System verbessert werden soll.

4 Erleichterung des Umgangs mit Shadern

4.1 GLSL Extension für Visual Studio Code

Der erste Schritt zur Verbesserung des Shader-Systems von FUDGE war es also die lesbarkeit des GLSL-Codes zu erhöhen und mehr Unterstützung durch die Entwicklungsumgebung zu ermöglichen. Die für FUDGE empfohlene Entwicklungsumgebung ist der Code-Editor “Visual Studio Code” von Microsoft.
Diese bietet dem Nutzer den sogenannten “Extension Marketplace” der Erweiterungen wie Unterstützung zusätzlicher Programmiersprachen, Debugger und andere Tools anbietet.
Dort zu finden ist unter anderem die Extension “WebGL GLSL Editor” von Rácz Zalán.
Diese bietet Unterstützung für GLSL durch Syntax-Highlighting, also hilfreichem Einfärben von Variablen, Typen, Funktion und so weiter. Außerdem werden Fehler im Code markiert und ungenutzte Variablen und Funktionen ausgegraut. Dadurch haben Entwickler*innen einen besseren Überblick über den GLSL-Code und werden schon während der Bearbeitungszeit auf Probleme hingewiesen. Das Problem in FUDGE ist jedoch, dass der Code nur als String vorliegt, sodass all diese Unterstützungen nicht stattfinden. Die Lösung für dieses Problem war es, die Vertex- und Fragment-Shader in separaten Dateien abzulegen. In diesen ist nur der GLSL-Code vorhanden, die Unterstützung durch die Extension funktioniert also. Jedoch können diese Dateien nicht wieder der restliche TypeScript-Code in JavaScript kompiliert werden und sind somit nicht im fertigen Build enthalten. Es wäre nun möglich den fertigen GLSL-Code zu kopieren und in die TypeScript-Klassen als Strings einzufügen, dies ist jedoch mit zunehmender Anzahl von Shadern zeitaufwendig und Fehleranfällig da Dateien verwechselt werden können. Besser ist ein Prozess, der zur Build-Zeit aufgerufen wird, die GLSL-Dateien einliest und in TypeScript-Klassen schreibt die im fertigen Build verwendet werden können.

4.2 GLSL-Reader/ TypeScript-Writer

Damit die in GLSL-Dateien geschriebenen Shader in ein von FUDGE verarbeitbares TypeScript-Format übertragen werden können sollte nun ein Prozess entwickelt werden der dies auf einfache Weise ermöglicht. Umgesetzt wurde dies mit einer JavaScript-Funktion, die über den Build-Task aufgerufen werden kann. Als erster Schritt wurde das Node.js File System Module eingebunden, dass die Arbeit mit dem Dateisystems des Computers

ermöglicht. Dies geschieht durch die Zeile *var fs = require('fs');*. Da FUDGE ohnehin auf Node.js angewiesen ist, kann dieses Modul ohne zusätzliche Installationen verwendet werden. Shader werden nun in GLSL-Dateien geschrieben. Diese befinden sich im Ordner "../Shader/ShaderSources/". Dort befinden sich zwei weitere Ordner die "Vertex" und "Fragment" heißen. Da die Shader nun in GLSL geschrieben werden müssen sich Vertex- und Fragment-Shader in verschiedenen Dateien befinden. Im Ordner "Vertex" befinden sich dementsprechend eine GLSL-Datei pro Shader, die den jeweiligen den Vertex-Shader enthält und die Dateiendung ".vert" verwendet. Das gleiche gilt für den Ordner "Fragment". Die Dateinamen sind identisch haben jedoch die Endung ".frag". Da jeder Vertex-Shader einen zugehörigen Fragment-Shader benötigt, müssen immer beide Dateien vorhanden sein. Außerdem dienen die Namen der Dateien auch der Benennung der Shader im späteren TypeScript-Code. Um zu prüfen wie viele Shader in TypeScript übertragen werden müssen, wird die Funktion "fs.readdirSync" des Node.js File System Modules genutzt. Als Parameter kann der Pfad zu einem Ordner angegeben werden, wie in diesem Fall der Order "Vertex" in dem sich alle Vertex-Shader-Dateien befinden. Als Rückgabe erhält man ein String-Array mit allen Dateinamen in diesem Ordner. Die Länge des Arrays ist also die Anzahl an Shadern die aktuell in FUDGE existieren. Nun wird für jedes Element in diesem Array, also jeden Shader, die Funktion "writeShader" aufgerufen. Diese nimmt einen String entgegen, der als Name des Shaders dient. Da die Vertex-Shader den gleichen Namen wie der gesamte Shader tragen kann jeweils der Dateiname ([Shadername].vert) am Punkt getrennt werden um die Dateiendung zu entfernen. Wie schon am Beispiel von "ShaderUniColor" gezeigt wird für einen Shader den FUDGE verwenden kann neben dem GLSL-Code in Form von Strings auch TypeScript-Code benötigt. Dieser ist aktuell für alle Shader identisch mit Ausnahme der Strings für die Shader-Sources und kann somit aus einem Template eingelesen werden. Dieses liegt nicht als TypeScript-, sondern als normale Text-Datei vor, damit diese beim späteren Build ignoriert wird. Das Template beinhaltet Dollarzeichen($) an denen der String der beim einlesen der Datei aufgeteilt wird. Das Dollarzeichen wurde gewählt, da es im normalen Code nicht vorkommt und der String somit nicht an unerwünschten Stellen getrennt wird.

ShaderCodeTemplate.txt:

```
namespace FudgeCore {
  @RenderInjectorShader.decorate
  export abstract class $ extends Shader {
        public static readonly iSubclass: number = Shader.registerSubclass($);

        public static vertexShaderSource: string =
        `$`;

        public static fragmentShaderSource: string =
        `$`;
  }
}
```

Dieses Template wird mit der Funktion “fs.readFile” eingelesen, sodass man einen String mit dem Inhalt erhält. Trennt man den String der durch das Einlesen dieses Templates entsteht an den Dollarzeichen, erhält man ein String-Array mit fünf Elementen. An die ersten zwei Stellen an denen ein Dollarzeichen war wird nun der Name des Shaders eingefügt, der als Parameter der Funktion übergeben wurde. Das dritte Dollarzeichen wird mit dem GLSL-Code des Vertex-Shaders ersetzt, den man aus der Datei *[Shadername].vert* erhält. An die letzte Stelle kommt der Fragment-Shader aus der Datei *[Shadername].frag*. Diese Dateiendungen werden von der GLSL-Erweiterung in der Entwicklungsumgebung erkannt, sodass die Unterstützung beim Programmieren stattfindet. Die beiden Dateien werden wie das Template auch mit der Funktion “fs.readFile” eingelesen.

Sind das Template und die Shader-Sources nun in richtiger Reihenfolge zusammengefügt entsteht ein String mit dem gesamten Code der für einen FUDGE-Shader nötig ist. Dieser wird in eine TypeScript-Datei geschrieben und der neue bzw. veränderte Shader steht Entwickler*innen nach dem nächsten Build zur Verfügung. Dies geschieht durch die Funktion “fs.writeFile”, die als Parameter einen Dateipfad und einen String für den Inhalt entgegen nimmt. Gespeichert werden die TypeScript-Dateien im Ordner “Shaders”. Ist eine Datei mit dem gleichen Namen schon vorhanden wird diese überschrieben (Shader-Update). Sollte der Dateiname noch nicht im Ordner existieren wird eine neue TypeScript-Datei erstellt (neuer Shader).

Nun soll es für Entwickler*innen, die neue Shader geschrieben oder vorhandene Shade überarbeitet haben möglich sein, den GLSL-Reader/TypeScript-Writer auf einfache Weise aufzurufen, sodass der neue GLSL-Code in die TypeScript-Dateien übernommen wird. Da sich die JavaScript-Datei, die den Prozess durchführt nicht einfach als Build-Task einbauen lässt, wird ein Windows Command File (Dateiendung .cmd) genutzt um zwei Befehle auszuführen. Der erste (*cd Core\Source\Shader*) öffnet den Ordner in dem sich die

JavaScript-Datei befindet und der zweite (*node UpdateShaders.js*) führt diese dann aus. Dieses Command File lässt sich nun für einen neuen Build-Task nutzen der “tasks.json” hinzugefügt wurde.

```
{
  "type": "process",
  "label": "update shaders - Core/Source/Shader/UpdateShaders.cmd",
  "group": "build",
  "command": "Core\\Source\\Shader\\UpdateShaders.cmd",
  "problemMacher": [],
}
```

Dieser Build-Task kann nun über das GUI von Visual Studio Code gestartet werden (*Terminal -> Run Build Task -> update shaders*) und überträgt alle Veränderungen in den GLSL-Dateien in die entsprechenden TypeScript-Dateien. Außerdem lässt sich dem normalen Build-Task, der FUDGE in eine einzige JavaScript-Datei überträgt eine Abhängigkeit zu dem Shader-Update-Task hinzufügen, sodass dieser vor dem eigentlichen Build ausgeführt wird und alle Shader-Updates auf jeden Fall beachtet werden.

```
{
  "type": "typescript",
  "tsconfig": "Core/Source/tsconfig.json",
  "problemMacher": ["$tsc"],
  "group": "build",
  "label": "tsc: build - Core/Source/tsconfig.json",
  "dependsOn": ["update shaders -
Core/Source/Shader/UpdateShaders.cmd"],
}
```

Betrachtet man nun wieder den UniColor-Shader als Beispiel, wird nicht mehr wie zuvor die TypeScript-Datei (ShaderUniColor.ts) bearbeitet. Stattdessen editiert man die beiden GLSL-Dateien “ShaderUniColor.vert” und “ShaderUniColor.frag”, in den sich der Vertex- und Fragment Shader befinden.

ShaderUniColor.vert:

```
#version 300 es
in vec3 a_position;
uniform mat4 u_projection;

void main() {
    gl_Position = u_projection * vec4(a_position, 1.0);
}
```

ShaderUniColor.frag:

```
#version 300 es
precision mediump float;
uniform vec4 u_color;
out vec4 frag;

void main() {
  frag = u_color;
}
```

Der GLSL-Code in diesen Dateien wird nun durch die Extension eingefärbt um die lesbarkeit zu erhöhen (Abhängig vom Color Theme dass man in VS Code eingestellt hat) und Fehler werden markiert, sodass sie direkt behoben werden können.
Die Datei “UpdateShaders.js” beinhaltet den Code, der die Daten aus den GLSL-Dateien einliest und in TypeScript-Dateien schreibt. Der Code befindet sich im Anhang.

5 Shader in FUDGE

5.1 Datentypen in GLSL

Sowohl Vertex- als auch Fragment-Shader benötigen Daten die sie weiter verarbeiten können. Bei Vertex-Shadern werden diese Daten in Form von Attributes, Uniforms und Textures übergeben. Bei Attributes handelt es sich um Daten aus Buffern, wie zum Beispiel der Position des Vertex oder seine Normale. Uniforms sind Werte die innerhalb eines Draw-Calls für alle Vertices gleich bleiben. In FUDGE sind dies zum Beispiel Lichtobjekte, Materialdaten und Matrizen die für Berechnungen benötigt werden. Bei Textures handelt es sich um Daten aus Pixeln.
Fragment-Shader erhalten ihre Daten aus Uniforms, Textures und Varyings. Uniforms sind hierbei Daten die in einem Draw-Call für alle Pixel gleich bleiben, wie zum Beispiel die Farbe des Materials eines Objektes. Textures sind wie bei Vertex-Shader Daten aus Pixeln. Varyings sind Daten, die vom Vertex-Shader an den Fragment-Shader weitergegeben wurden. Dazu muss die Varying in beiden Shadern deklariert werden. Hierbei werden die Angaben “out” und “in” verwendet um wie im Beispiel des Flat-Shaders eine Farbe weiterzugeben.

```
flat out vec4 v_color;
```
(Farbe wird im Vertex-Shader berechnet und and Fragment-Shader weitergegeben)
```
flat in vec4 v_color;
```
(Fragment-Shader nimmt Daten entgegen und kann sie zum Beispiel zur Interpolation verwenden)
(vgl. Gregg Tavares, 2021, WebGL Shaders and GLSL)

In FUDGE werden diese Daten mit Hilfe der Klasse “RenderInjector” weitergegeben. Weitere Klassen erben von dieser und spezialisieren sich zum Beispiel auf Shader, Meshes und Texturen. In diesen Klassen wird festgelegt mit welchen Namen man in GLSL auf die verschiedenen Daten zugreifen kann. Zum Beispiel erhält man durch die Zeile

```
in vec3 a_position;
```

die Position des Vertex für den der Vertex-Shader aufgerufen wurde. Dieses Attribute muss “a_position” heißen, da es so in der Klasse “RenderInjectorMesh” festgelegt wurde. Ändert man die Namen dieser Attributes und Uniforms werden die Daten nicht an die Shader weitergegeben und fehlen somit in den Berechnungen. Die Daten für die Attribute erhält “RenderInjectorMesh” aus der Klasse “Mesh” mit Hilfe der Methode “getBufferSpecification”. Auf die gleiche Weise werden auch die Attribute-Namen für Face- und Vertexnormalen, sowie Texture-UVs festgelegt und mit Daten versehen. Uniforms die in “RenderInjectorMesh” festgelegt werden sind drei Matrizen “u_world”, “u_projection” und “u_normal”, die in Shadern genutzt werden um Clip-Space-Koordinaten und Transformationen durchzuführen. Die Matrix “u_normal”, also die Normalenmatrix war noch nicht in FUDGE implementiert. Sie ist jedoch für Gouraud- und Phong-Shader nötig um Normalen in den Eye-Space zu transformieren, um Effekte wie Glanzlichter berechnen zu können. Um “u_normal” zu berechnen muss zunächst “u_world” invertiert und anschließend transponiert werden. In der Methode in der “u_normal” festgelegt wird, wird die Worldmatrix als Parameter “_mtxWorld” übergeben und kann somit zur Berechnung verwendet werden. Die Klasse “Matrix4x4” in FUDGE hat bereits eine Methode um eine Matrix zu invertieren. Diese konnte also auf “_mtxWorld” angewendet werden. Eine Methode um Matrizen zu transponieren war jedoch noch nicht vorhanden und musste implementiert werden. Dazu wurde eine neue statische Methode “TRANSPOSE” in “Matrix4x4” angelegt. Sie nimmt als Parameter eine Matrix entgegen, transponiert diese, also vertauscht Zeilen und Spalten und gibt die neue Matrix als Rückgabewert zurück.

```
public static TRANSPOSE(_mtx: Matrix4x4): Matrix4x4 {
      let m: Float32Array = _mtx.data;
      _mtx.data.set([
        m[0], m[4], m[8], m[12],
        m[1], m[5], m[9], m[13],
        m[2], m[6], m[10], m[14],
        m[3], m[7], m[11], m[15]
      ]);
      return _mtx;
}
```

Mit dieser neuen Methode kann nun die Normalenmatrix “u_normal” berechnet und implementiert werden.

```
let uNormal: WebGLUniformLocation = _shader.uniforms["u_normal"];
   if (uNormal) {
     let normalMatrix: Matrix4x4 =
        Matrix4x4.TRANSPOSE(Matrix4x4.INVERSION(_mtxWorld));
     crc3.uniformMatrix4fv(uNormal, false, normalMatrix.get());
}
```

Bei den Attributes kam es im Verlauf dieser Arbeit noch zu einer kleinen Änderungen. Das Attribute “a_normal” war bereits vorhanden. Dabei handelte es sich um Facenormalen eines Meshes, die zum Beispiel für den Flat-Shader verwendet werden. Die Shader die im Laufe dieser Arbeit entwickelt wurden benötigen jedoch Vertexnormalen die aktuell weder berechnet noch implementiert wurden. Worum es sich dabei handelt und wie sie berechnet werden wird im Kapitel “Vertexnormalen” beschrieben. Was jedoch hier erwähnt werden kann ist, dass das Attribute “a_normal” nicht mehr existiert und durch die beiden neuen Attributes “a_normalFace” und “a_nomralVertex” ersetzt wurde, um zwischen den beiden Normalentypen unterscheiden zu können und den Zugriff auf diese zu ermöglichen.

5.2 Vorhandene Shader

Zu Beginn dieser Arbeit waren bereits sechs verschiedene Shader in FUDGE vorhanden, die von Jirka Dell'Oro-Friedl, Jascha Karagöl und Simon Storl-Schulke entwickelt wurden.
Der simpelste Shader ist wie bereits erwähnt “UniColor”. Dieser stellt Objekte einfarbig, ohne Schattierung dar und ignoriert somit sämtliche Beleuchtung die in der Szene vorhanden sein könnte. Im Vertex-Shader wird lediglich die Clip-Space-Position (*gl_Position*) des Vertex berechnet, durch Multiplikation der Vertex-Koordinaten mit der Projektionsmatrix.

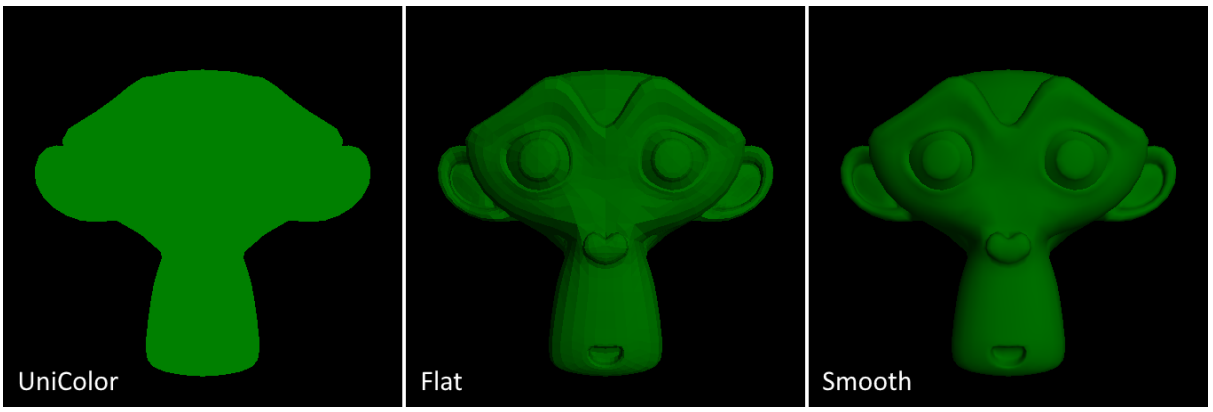
```
gl_Position = u_projection * vec4(a_position, 1.0);
```

Der Fragment-Shader zeichnet das Objekt mit der Farbe des Materials.
Der Shader “Flat” ist schon ein wenig Komplexer, da er die Helligkeit einer Fläche auf Grundlage der Lichtquellen in der Szene berechnet. So entstehen einfache Schattierungen, die einzelnen Flächen bleiben jedoch einfarbig und die Kanten des Meshes sind deutlich sichtbar (vgl. Gambetta, 2021, Kapitel 13). Die Vertex-Position wird gleich wie im UniColor-Shader berechnet. Jedoch werden nun die Daten der Szenenbeleuchtung genutzt um Farben bzw. eine Helligkeit zu berechnen. Die Farbe wird jedoch nur an einer Stelle einer Fläche berechnet und für alle Pixel dieser Fläche verwendet. Dazu wird eine Output-Variable *v_color* erstellt

die später an den Fragment-Shader übergeben wird. Zuerst wird diese der Farbe des Ambient-Licht gleichgesetzt, wodurch eine gewissen Grundhelligkeit für alle Vertices entsteht. Außerdem wird eine Normale berechnet, die in diesem Fall der Normale des Faces zu dem der Vertex gehört entspricht. Dann wird durch alle Directional-Lichter in der Szene iteriert um eine Helligkeit zu berechnen. Dazu wird das negative Skalarprodukt der Normalen und der Richtung des Lichts berechnet. Dieses wird dann mit der Farbe des Lichts multipliziert und auf *v_color* addiert, sollte der Helligkeitswert größer als null sein. Zuletzt wird der Alphawert der Output-Farbe auf 1 gesetzt um unerwünschte Transparenz zu verhindern. Der Fragment-Shader nimmt *v_color* entgegen und multipliziert den Wert mit der Materialfarbe. Um zu sichern, dass auch Meshes bei denen sich mehrere Faces einen Vertex teilen im gewünschten Stil dargestellt werden muss bei der Deklaration von *v_color* das "flat" mit angegeben werden.

```
flat out vec4 v_color; (Vertex-Shader)
flat in vec4 v_color;  (Fragment-Shader)
```

Passiert dies nicht entsteht bei diesen Meshes Smooth-Shading, bei dem weiche Schattierungen gezeichnet werden, die die Kanten des Meshes verstecken oder zumindest retuschieren. Flächen sind hier nicht mehr einfarbig, sondern zeigen weiche Übergänge.
So kann man also mit einer kleinen Änderung am Code des Flat-Shaders einen Smooth-Shader erstellen. Damit dieser jedoch auch mit Meshes funktioniert, bei denen jeder Vertex nur zu einem Face gehört, muss bei der Berechnung der Normalen die Vertexnormale an Stelle der Facenormalen genutzt werden. Die Berechnung dieser war zu Beginn dieser Arbeit noch nicht vorhanden und musste deshalb eingebaut werden. Wie das umgesetzt wurde und wo der Unterschied zwischen Face- und Vertexnomalen liegt wird im Kapitel "Vertexnormalen" erklärt. Der Smooth-Shader war noch nicht in FUDGE vorhanden, konnte aber wie erwähnt mit minimalem Aufwand erstellt werden.
Im folgenden Bild sieht man nun die Unterschiede zwischen den drei Shadern. Durch die Fehlende Schattierung bei UniColor hat der Betrachter keine Information zur räumlichen Tiefe des Meshes. Jedoch kann er genutzt werden Objekte in Szenen ohne Beleuchtung sichtbar zu machen. Bei anderen Shadern würde eine fehlende Beleuchtung in einem schwarzen Bild resultieren.



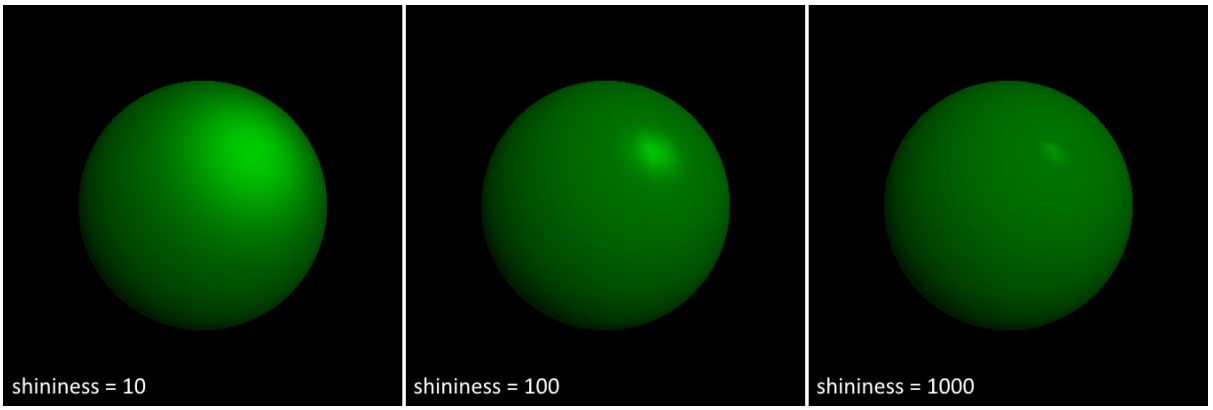
(Bei dem Affenkopf-Modell handelt es sich um "Suzanne" aus der 3D-Grafiksuite Blender. Es wurde genutzt um die FUDGE-Shader zu testen. Die Form des Modells ist ein wenig komplexer als ein Würfel oder eine Kugel und vermittelt somit einen besseren Einblick in die Effekte die von den Shadern erzeugt werden.)

Ein weiterer Shader der schon in FUDGE vorhanden war ist "MatCap" (Material Capture). Für diesen wird eine Sphäre in eine Textur gerendert, welche wiederum als Referenz genutzt wird um festzulegen wie das Objekt dargestellt wird.
Die restlichen Shader sind "Texture", also um Objekte mit einer Textur zu rendern und "Pick" bzw. "PickTextured", die für Raycasting genutzt werden.
Zwei viel verwendete Shader die noch nicht in FUDGE vorhanden waren sind "Gouraud" und "Phong". Beide Shader resultieren in weichen Schattierungen und können basierend auf einem Wert des Materials der festlegt wie "glänzend" ein Objekt sein soll Reflektionspunkte berechnen. Die beiden Shader unterscheiden sich jedoch in Qualität und Berechnungsgeschwindigkeit, da der Großteil der Berechnungen bei Gouraud im Vertex-Shader (also einmal pro Vertex) und bei Phong im Fragment-Shader (einmal pro Pixel) stattfinden.

5.3 Gouraud-Shader

Anders als der Flat-Shader, der die Beleuchtung einer Fläche an einer Stelle mit Hilfe der Facenormalen berechnet, benutzt der Gouraud-Shader Vertexnormalen, um die Beleuchtung an jedem Vertex zu berechnen (vgl. Gambetta, 2021, Kapitel 13). In einem Triangle-Mesh wie in FUDGE verwendet wird, erhält man somit pro Fläche eines Objekts drei Helligkeitswerte zwischen den interpoliert werden kann um fließende Helligkeitsübergänge in der Fläche zu zeichnen. Die Beleuchtungsberechnung bei Gouraud findet im Vertex-Shader

statt. Grundsätzlich funktioniert die Berechnung ähnlich wie im Flat-Shader, nur dass die drei Vertices einer Fläche nun unterschiedliche Farbwerte erhalten können, da an Stelle der Facenormalen Vertexnormalen verwendet werden. Außerdem kommt eine Funktion hinzu, die abhängig von der Betrachtungsrichtung, der Richtung der Directional-Lichter, der Normalen des Vertex und einer Variabel “shininess”, die im Material angegeben werden kann die Helligkeit berechnet. Hierbei können Glanzpunkte entstehen, die ein Objekt je nachdem wie hoch der Wert für “shininess” gewählt wurde glänzender oder matter aussehen lassen.



Um einem Material einen Wert für “shininess” mitgeben zu können erhielt die Klasse “CoatColored” ein neues Attribut “shininess” vom Typ number. Erstellt man nun ein neues Material kann diesem Optional ein Wert mitgegeben werden. Geschieht dies nicht wird “shininess” auf 0 gesetzt. Je höher der gewählte Wert für “shininess” ist, desto glänzender erscheint das Objekt mit diesem Material. Damit Shader auf “shininess” zugreifen können wurde in der Klasse “RenderInjectorCoat” die Uniform “u_shininess” angelegt.

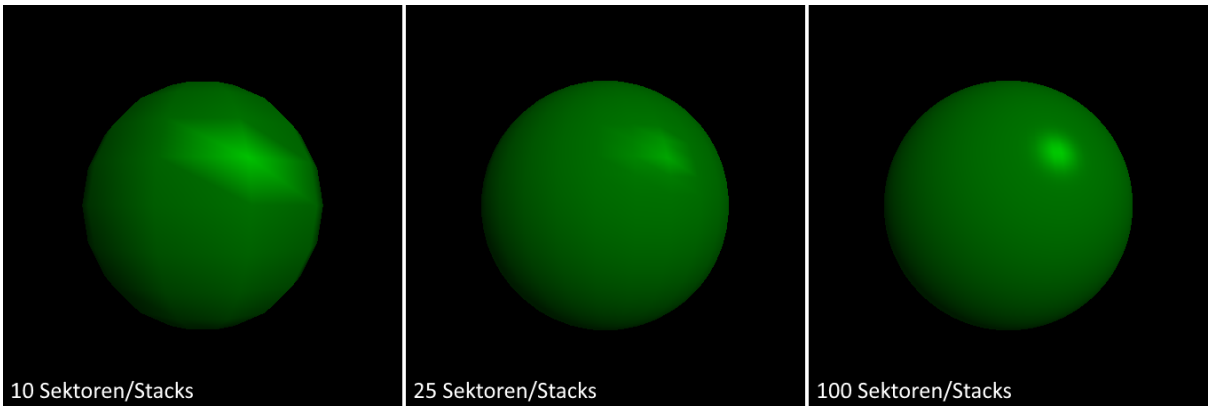
```
let shininessUniformLocation: WebGLUniformLocation =
_shader.uniforms["u_shininess"];
let shininess: number = (<CoatColored>this).shininess;
RenderWebGL.getRenderingContext().uniform1f(shininessUniformLocation,
shininess);
```

Es ist geplant das Material/Coat-System von FUDGE in Zukunft zu überarbeiten. Dadurch wird sich mit hoher Wahrscheinlichkeit auch ändern auf welche Weise einem Material dieser Wert mitgegeben werden kann. Passiert dies muss lediglich in “RenderInjectorCoat” angepasst werden woher der Wert für “u_shininess” kommt. In den Shadern selbst muss nichts verändert werden.

```
vec3 calculateReflection(vec3 light_dir, vec3 view_dir, vec3 normal,
float shininess) {
    vec3 color = vec3(1);
    vec3 R = reflect(-light_dir, normal);
    float spec_dot = max(dot(R, view_dir), 0.0);
    color += pow(spec_dot, shininess);
    return color;
}
```

Zur Berechnung der Glanzpunkte wird zunächst die in GLSL vorhandene Funktion “reflect” verwendet, die eine Reflektionsrichtung basierend auf der Lichtrichtung und der Vertexnormalen berechnet. Nun wird das Skalarprodukt zwischen der Reflektionsrichtung und der Betrachtungsrichtung kalkuliert. Potenziert man dieses Ergebnis mit dem *shininess*-Faktor erhält man eine Farbe, die mit der der normalen Helligkeitsberechnung und der Farbe des Directional-Lichts multipliziert wird um den den endgültigen Farbwert für den Vertex zu erhalten. Diese Berechnung findet einmal pro Directional-Licht statt und die Ergebnisse werden addiert.

Da bei diesem Shader die Beleuchtungsberechnung im Vertex-Shader stattfindet, ist diese zwar effizient, aber in ihrer Qualität stark abhängig vom Detailgrad des Meshes. Die Glanzpunkte in Meshes mit wenigen Polygonen sind deutlich verwaschener, da die Fläche die durch den Vertex beeinflusst wird im Verhältnis größer ist. Im folgenden Bild wurden die Kugeln mit der gleichen Beleuchtung und dem gleichen Wert für “shininess” gerendert. Jedoch erhöht sich von links nach rechts die Polygondichte.



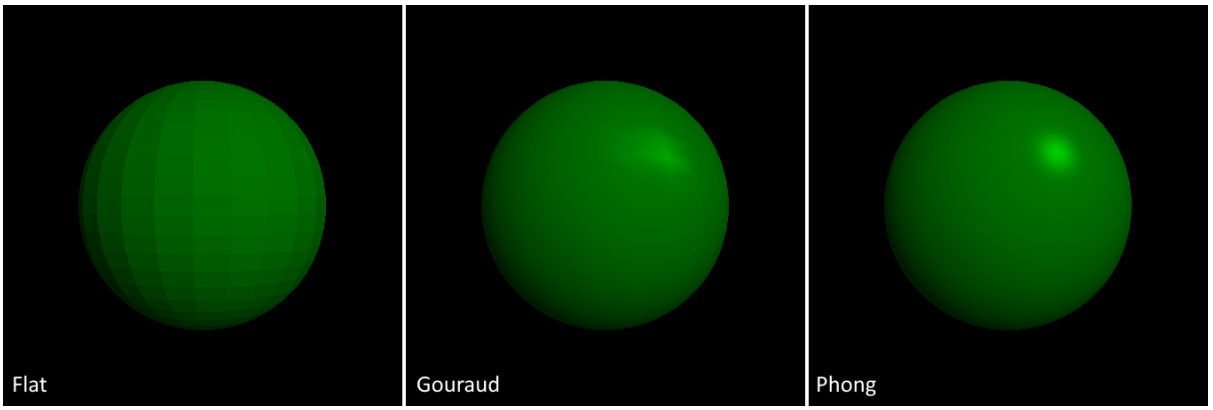
Der vollständige Shader-Code befindet sich im Anhang.

5.4 Phong-Shader

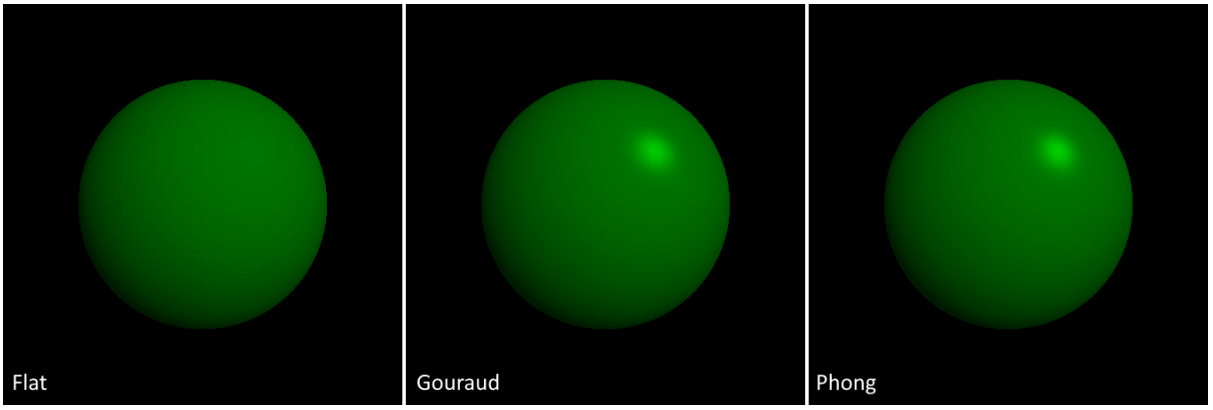
Der Phong-Shader funktioniert ähnlich wie der Gouraud-Shader mit dem Unterschied, dass die Beleuchtungsberechnung nicht im Vertex- sonder im Fragment-Shader stattfindet.
Im Vertex-Shader wird eine Normale “f_normal” berechnet die an den Fragment-Shader übergeben wird.

```
f_normal = vec3(u_normal * vec4(a_normalVertex, 0.0));
```

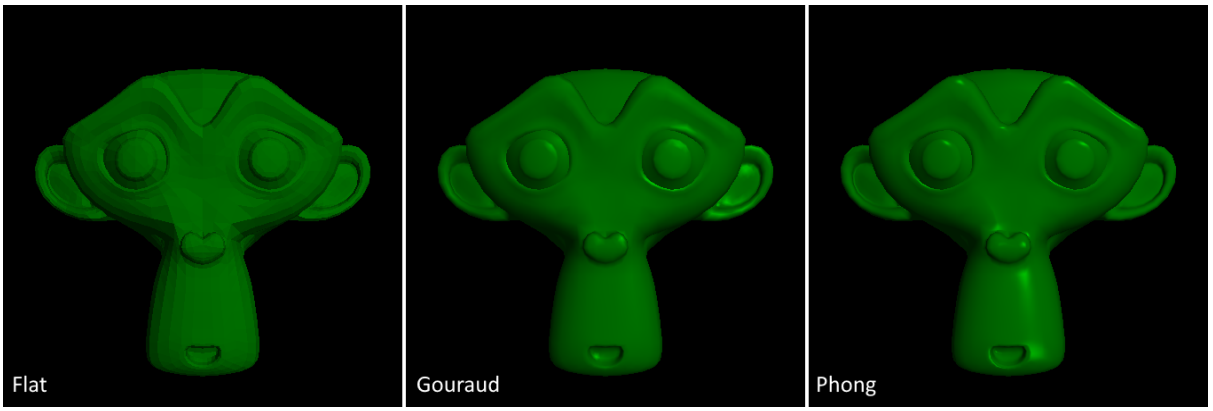
Zur Einfärbung wird eine neue Normale interpoliert, mit der die Beleuchtung ausgewertet wird. Es wird bei Phong also für jeden Pixel das Beleuchtungsmodell ausgewertet, anstatt wie bei Gouraud für jeden Vertex. Dies resultiert in einem qualitativ besserem Ergebnis, erhöht aber auch den Berechnungsaufwand.



Die drei Kugeln im obigen Bild haben eine identische Anzahl an Flächen, den gleichen Wert für “shininess” und wurden mit der gleichen Beleuchtung gerendert. In diesem Beispiel mit einem eher groben Mesh erzielt der Phong-Shader deutlich bessere Ergebnisse. Die Berechnung ist wie erwähnt jedoch aufwändiger und kann somit zu Performanceproblemen führen. Da Performanceoptimierung in FUDGE eine niedrige Priorität hat, sollte dies bei der Wahl des Shaders beachtet werden. Erhöht den Detailgrad des Meshes sinkt der qualitative Unterschied zwischen Gouraud und Phong.



Durch die kleineren Flächen ist die Interpolierung zwischen den Vertex-Werten bei Gouraud näher am gewünschten Ergebnis und der Unterschied zu Phong ist kaum noch sichtbar. Jedoch muss in diesem Fall beachtet werden, dass durch die Erhöhung der Anzahl von Vertices auch der Rechenaufwand für Gouraud steigt, da der Vertex-Shader in dem die Berechnung stattfindet einmal pro Vertex aufgerufen wird. Bei Phong findet sie weiterhin einmal pro Pixel statt, ist also gleichbleibend solange die Canvasgröße nicht verändert wird. Um ein besseres Ergebnis mit Gouraud zu erzielen kann also die Qualität des Meshes erhöht werden, je nach Situation kann aber ein grobes Mesh mit Phong-Shading gleiche Ergebnisse mit weniger Rechenaufwand liefern.



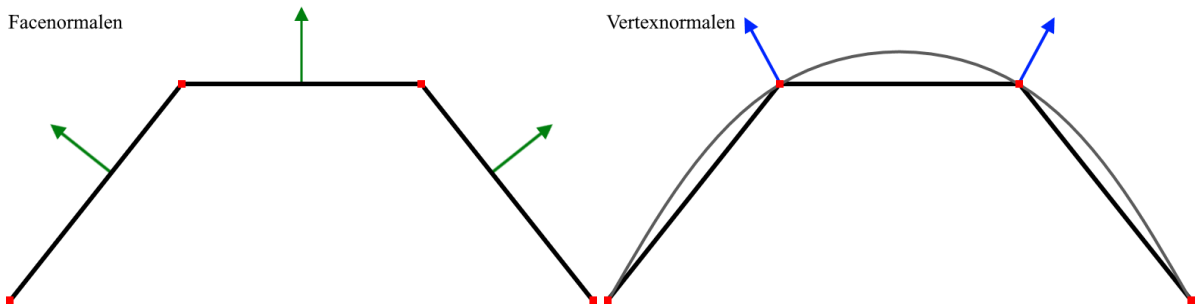
Bei diesem Mesh entstehen mehrfach Glanzpunkte an Stellen mit teilweise unterschiedlich großen Flächen. Dadurch sind die Ergebnisse von Gouraud und Phong an vielen Stellen sehr unterschiedlich. Die Annäherung die Gouraud vornimmt ist nicht immer erfolgreich, sodass Glanzpunkte fehlen oder in ihrer Größe nicht akkurat sind. Phong liefert hier deutlich bessere Ergebnisse. Noch stärker wird der qualitative Unterschied wenn das Mesh sich in der Szene bewegt, zum Beispiel um eine Achse rotiert wird. Während die Glanzpunkte bei Phong gleichmäßig über die Flächen wandern, ist es bei Gouraud eher der Fall das ein Glanzpunkt an einer Stelle schwächer wird bis er verschwindet und an einer anderen Stelle wieder auftaucht. Dieser Effekt entsteht, da sich an Glanzpunkt bei Gouraud immer an der Position eines Vertex befindet und somit nicht mitten auf einer Fläche entstehen kann. Der Phong-Shader ist hingegen dazu in der Lage und liefert somit eine realistischer wirkende Interpretation der Beleuchtung.

6 Vertexnormalen

Für das Rendern mit weicher Schattierung brauchen die entsprechenden Shader (Smooth, Gouraud, Phong) Vertexnormalen. Diese wurden zu Beginn dieser Arbeit noch nicht in FUDGE berechnet. Es mussten also neue Funktionen die dies übernehmen geschrieben werden.

6.1 Unterschied zu Facenormalen

Die Klasse “Mesh” in FUDGE beinhaltet schon eine Methode zur Berechnung der Normalen für das Mesh, um diese unter anderem für Shader zu nutzen. Die Normalen die dabei entstehen sind Facenormalen, die zum Beispiel für den Flat-Shader gebraucht werden. Dabei handelt es sich um Normalenvektoren, die senkrecht zur entsprechenden Fläche stehen. In einem Trianglemesh wie bei FUDGE können sie aus dem Kreuzprodukt zwei der Vektoren die das Triangle einspannen und der Normierung dieses Ergebnisses berechnet werden.
Für Shader wie Gouraud oder Phong, die in weichen Schattierungen resultieren sollen, ist es jedoch nötig auch Vertexnormalen zu berechnen. Hierbei haben die drei Vertices eines Faces nicht den gleichen Normalenvektor. Stattdessen erhält jeder Vertex einen eigenen Normalenvektor, der in einem sinnvollem Verhältnis von allen Flächen die dieser Vertex mit bildet beeinflusst wird. So wird eine Oberfläche simuliert, die runder ist als die Flächen des Meshes durch das sie simuliert wird. (vgl. Gambetta, 2021, Kapitel 13)



6.2 Berechnung für FUDGE-Meshes

Damit Shader in FUDGE die Vertexnormalen eines Meshes erhalten können musste die Klasse Mesh also erweitert werden. Sie erhielt ein neues Attribut “ƒnormalsVertex” und das Attribut “ƒnormals”, das schon vorhanden war und die Daten für Facenormalen enthielt, wurde in “ƒunNormalsFace” umbenannt, damit der Unterschied eindeutig ist. Nun brauchte es noch eine Methode die aus den vorhandenen Daten Vertexnormalen berechnen kann. Da es

sich bei Vertexnormalen um einen Durchschnitt der umliegenden Facenormalen handelt, liegt es nahe, diese zu addieren und das Ergebnis zu normieren. Das Resultat würde dann jedoch jeder anliegenden Fläche die gleiche Gewichtung geben, dass heißt eine sehr kleine Fläche hätte den gleichen Einfluss auf die Vertexnormale wie eine sehr große Fläche. Besser ist es, wenn der Einfluss einer Fläche mit ihrer Größe steigt. Dies kann erreicht werden, indem nicht die Facenormalen der Flächen addiert werden, sondern die Kreuzprodukte die für die Berechnung dieser genutzt werden, bevor sie normiert werden. Bei größeren Flächen sind diese Werte auch größer, verglichen mit kleineren Flächen. Nimmt man als Beispiel ein Dreieck A(0, 0, 0) B(6, 0, 0) C(3, 5, 0) beträgt dessen Fläche 15 Einheiten². Berechnet man nun das Kreuzprodukt von B-A und C-A erhält man den Vektor (0, 0, 30). Bei einem größerem Dreieck zum Beispiel A(0, 0, 0) B(0, 12, 0) C(0, 6, 10) mit einer Flächengröße von 60 Einheiten² ist das Ergebnis (120, 0, 0). Sowohl Flächeninhalt als auch Betrag des Ergebnisvektors stehen bei den beiden Dreiecken im Verhältnis 1 zu 4. Würde man diese beiden Vektoren erst normieren und dann addieren wäre der Ergebnisvektor (1, 0, 1), normiert (0.707, 0, 0.707). Addiert man jedoch die nicht-normierten Vektoren erhält man den Vektor (120, 0, 30) und die Normale (0.97, 0, 0.243). Diese Normale wird nun in ihrer Richtung vom Flächeninhalt des entsprechenden Faces beeinflusst. Auf diese Weise kann also eine Vertexnormale gewichtet werden. Die dafür benötigten Kreuzprodukte werden schon in der Funktion für die Facenormalen berechnet. Deshalb werden sie nun in einem neuen Attribut "*f*faceCrossProducts" gespeichert und dann von den beiden Methoden für die Berechnung von Face- und Vertexnormalen verwendet.
Um die Vertexnormalen zu berechnen werden nun diese Kreuzprodukte aller Flächen an denen der Vertex anliegt addiert und anschließend normiert. Die Zuordnung der Vertices zu den Flächen ist hierbei schon bekannt aus der Methode "createFaceNormals" und konnte somit zurückverfolgt werden. Die gesamte neue Methode "createVertexNormals" ist im Anhang zu finden.

6.3 Berechnung für importierte Meshes

FUDGE wurde eine neue Klasse "MeshObj" hinzugefügt, die es erlaubt Wavefront OBJ Modelle zu importieren und somit auch komplexe Meshes in FUDGE zu verwenden. Mit dieser Klasse wurde zum Beispiel der zuvor gezeigte Affenkopf importiert. Zuvor waren nur simple Meshes wie Würfel, Kugel, Zylinder und ähnliches vorhanden. Komplexere Meshes sind sehr hilfreich beim Testen von Shadern, da viele Effekte die erzeugt werden bei simplen

Körpern nicht sichtbar sind. Nun gab es jedoch einen entscheidenden Unterschied zwischen den von FUDGE erstellten Meshes und den importierten. Ein Mesh das in FUDGE entsteht hat die Eigenschaft, dass ein Vertex zu mehreren Faces gehören kann. Zum Beispiel ist die Ecke eines Würfels ein Vertex, an dem drei Faces anliegen. Ein FUDGE Würfel hat somit acht Vertices. Die importierten Meshes sind anders aufgebaut. Hier hat jedes Face seine eigenen drei Vertices (aktuell ist nur der Import von triangulierten Meshes möglich). Das heißt ein importierter Würfel hat 36 Vertices, wobei sich immer mehrere Vertices an der gleichen Position befinden. Dies hat zur Folge, dass die Methode zur Berechnung der Vertexnormalen nicht mehr wie gewünscht funktioniert, da jeder Vertex nur noch an einem Face anliegt und seine Normale somit nur von einem Face beeinflusst wird. Die Normale der Vertices wäre identisch mit der Normalen ihres Faces, sodass das Ergebnis eher wie beim Flat-Shader aussieht, ohne weiche Schattierungen. Die Klasse “MeshObj” benötigte also eine eigene Methode zur Berechnung der Vertexnormalen, die prüft welche Flächen sich einen Vertex teilen würden. Dazu prüft die Methode für jeden Vertex welche Vertices sich an der gleichen Position befinden und speichert diese in einem Array ab. Dann addiert sie alle entsprechenden Kreuzprodukte die auch hier schon von einer anderen Methode berechnet und einfach separat gespeichert wurden. Der Ergebnisvektor wird normiert und dem Array für die Vertexnormalen hinzugefügt.

```
protected createVertexNormals(): Float32Array {
  let vertexNormals: number[] = [];
  for (let i: number = 0; i < this.vertices.length; i += 3) {
    let vertex: Vector3 = new Vector3(this.vertices[i],
                          this.vertices[i + 1], this.vertices[i + 2]);
    let samePosVerts: number[] = [];
    for (let j: number = 0; j < this.vertices.length; j += 3) {
      if (this.vertices[j] == vertex.x && this.vertices[j + 1] ==
vertex.y
                             && this.vertices[j + 2] == vertex.z)

        samePosVerts.push(j);
      }
      let sum: Vector3 = Vector3.ZERO();
      for (let z: number = 0; z < samePosVerts.length; z++)
        sum = Vector3.SUM(sum, new Vector3(
                        this.faceCrossProducts[samePosVerts[z] + 0],
                        this.faceCrossProducts[samePosVerts[z] + 1],
                        this.faceCrossProducts[samePosVerts[z] + 2]));

      if (sum.magnitude != 0)
        sum = Vector3.NORMALIZATION(sum);

      vertexNormals.push(sum.x, sum.y, sum.z);
  }
  return new Float32Array(vertexNormals);
}
```

7 Modularität

Ein weiteres Ziel dieser Arbeit war es mit der Entwicklung eines modularen Shader-Systems zu beginnen, bei dem Shadermodule erstellt und verwendet werden können. Motivation dafür war zum einen, dass Teile des GLSL-Codes für verschiedene Shader identisch sind, Module an dieser Stelle also sinnvoll sind. Außerdem würde es Entwickler*innen die mit FUDGE arbeiten neue, auf ihre Bedürfnisse angepasste, Shader zu erstellen und dies dies mit der Hilfe vorhandener Module.

7.1 Module verwenden

Shader in FUDGE bestehen aus GLSL-Code der in Form von Strings vorliegt. Das bedeutet, man kann nach belieben Strings zusammenfügen um den Code zu Erweitern. Um Shadermodule zu verwenden ist es zuerst nötig einen neuen Shader zu erstellen. Dies geschieht durch eine neue TypeScript-Klasse die von “Shader” erbt. In dieser können dann die beiden Strings “vertexShaderSource” und “fragmentShaderSource” überschrieben werden.

```
abstract class ShaderCustom extends fudge.Shader {

  public static vertexShaderSource: string = /*Vertex-Shader*/;

  public static fragmentShaderSource: string = /*Fragment-Shader*/;
}
```

An dieser Stelle könnten Entwickler*innen nun eigenen GLSL-Code einfügen, um auf ihre Bedürfnisse angepasste Shader zu verwenden, falls die Shader die standardmäßig in FUDGE vorhanden sind nicht ausreichend sind. Um hier den Schreibaufwand und die Fehleranfälligkeit zu verringern können Entwickler*innen Shader-Module verwenden die FUDGE zur Verfügung stellt. Dabei handelt es sich um eine String-Enum “SHADER_MODULE”, dass den Zugriff auf verschiedene Shader-Module erlaubt die als String vorliegen. Ein Beispiel für einen wahrscheinlichen Anwendungsfall wäre es, dass ein Shader die Daten aus den Lichtquellen in der Szene verwenden soll. Dazu werden Structs und Uniforms verwendet, die Shaderunabhängig immer gleich sind, und somit als ein Modul vorhanden sind. Möchte man dieses Modul in seinen Vertex-Shader einbauen würde dies folgendermaßen funktionieren:

```
abstract class ShaderCustom extends fudge.Shader {

  public static vertexShaderSource: string =
fudge.SHADER_MODULE.LIGHTS;

  public static fragmentShaderSource: string = /*Fragment-Shader*/;
}
```

Damit wird auf das Enum-Member “LIGHTS” zugegriffen welches diesen String repräsentiert:

```
`struct LightAmbient {
    vec4 color;
};
struct LightDirectional {
    vec4 color;
    vec3 direction;
};
const uint MAX_LIGHTS_DIRECTIONAL = 10u;
uniform LightAmbient u_ambient;
uniform uint u_nLightsDirectional;
uniform LightDirectional u_directional[MAX_LIGHTS_DIRECTIONAL];`
```

Auf diese Weise können Module wie Strings addiert werden um vollwertige Shader zu erstellen. Zusätzlich kann eigener GLSL-Code in Form von Strings mit eingebaut werden, um Effekte zu erzielen die in FUDGE (noch) nicht vorhanden sind. Hierbei müssen die Regeln von GLSL beachtet werden. Zum Beispiel müssen bei Shader-Sources mit dem Versionshinweis beginnen und eine Main-Funktion beinhalten.

7.2 Module erstellen

Damit man neue Shader-Module nicht in String-Form in die Enum “SHADER_MODULE” einpflegen muss wurde eine GLSL-Datei “ShaderModules.ts” erstellt. Module liegen hier als GLSL-Code vor und sind mit dem Keyword versehen, dass sie für die Enum erhalten. Eine Fehlerkontrolle durch die VS-Code-Erweiterung findet hier zwar nicht statt, jedoch wird der Code hilfreich eingefärbt. Ein Moduleintrag sind folgendermaßen aus:

```
HEAD_VERT//
#version 300 es
precision highp float;
in vec3 a_position;//
```

In der ersten Zeile befindet sich das Keyword, dass heißt in diesem Fall könnte man mit “SHADER_MODULE.HEAD_VERT” auf diese Modul zugreifen. Danach folgt das eigentliche Modul. Die doppelten Schrägstriche “//” werden hierbei vom Prozess “UpdateShaderModules.js” verwendet um Module und Keywords zu trennen dürfen deshalb

nicht vergessen werden. “UpdateShaderModules.js” ist hierbei ein Prozess der als Build-Task aufgerufen werden kann und ähnlich wie der zuvor beschriebene Prozess “UpdateShaders.js” funktioniert. Seine Aufgabe ist es die GLSL-Datei “ShaderModules.glsl” in eine TypeScript-Enum zu übertragen, sodass man nach dem FUDGE-Build auf die Module zugreifen kann. Dabei liest der Prozess die Datei “ShaderModules.glsl” und teilt ihren Inhalt an den doppelten Schrägstrichen. Danach werden die Keywords mit den Modulen kombiniert um sie zu Enum-Members zu machen. Diese Strings werden mit dem notwendigen TypeScript-Code kombiniert und das Ergebnis ist dann eine TypeScript-Datei die die Enum “SHADER_MODULE” enthält, wobei die Members aus der Datei “ShaderModules.glsl” stammen. Der vollständige Code von “UpdateShaderModules.js”, sowie Beispiele für “ShaderModules.glsl” und “ShaderModules.ts” befinden sich im Anhang.

7.3 GLSL-Macros

Da FUDGE-Shader jetzt in GLSL-Dateien geschrieben werden, entsteht die Möglichkeit weitere Erweiterungen in der Entwicklungsumgebung zu verwenden, die Entwickler*innen die Arbeit an Shadern ein wenig erleichtert. Ein Beispiel ist die Extension “GLSL Include” von pucelle. Diese erlaubt es mit der Hilfe eines “#include” Befehls Shader-Module zu importieren. Um diese Funktion in FUDGE nutzen zu können befinden sich Shader-Module als GLSL-Dateien im Ordner “ShaderSources\Modules”. Nun können Entwickler*innen durch den “#include”-Befehl und Angabe eines Dateipfades Shader-Module zu ihrem Shader hinzufügen ohne sie kopieren oder von Hand schreiben zu müssen.
Ein Beispiel: Das Modul “Lights” ist vorhanden in der Datei “Lights.glsl” und enthält diesen GLSL-Code:

```
#define GLSLIFY 1
struct LightAmbient {
    vec4 color;
};
struct LightDirectional {
    vec4 color;
    vec3 direction;
};

const uint MAX_LIGHTS_DIRECTIONAL = 10u;
uniform LightAmbient u_ambient;
uniform uint u_nLightsDirectional;
uniform LightDirectional u_directional[MAX_LIGHTS_DIRECTIONAL];
```

Nun kann man in der Shader-Datei (z.B. Shader[Name].vert) an der man arbeitet folgende Zeile schreiben:

```
#include ../Modules/Lights.glsl
```

Dabei erscheinen Fenster die das Durchsuchen der Ordner erleichtert und auch anzeigen welche Dateien und Ordner vorhanden sind.
Speichert man nun die Datei beginnt die Erweiterung diese zu kompilieren und ersetzt dabei die “#include”-Zeile mit dem Inhalt von “Lights.glsl”. In den Einstellungen der Erweiterung kann man festlegen in welchem Ordner die kompilierten Dateien gespeichert werden sollen und wie sie benannt werden. Dabei ist es auch möglich diese Einstellung so zu wählen, dass die Datei an der man arbeitet einfach überschrieben und der neue GLSL-Code einfach direkt eingefügt wird. Dies hat den Vorteil, dass die Erweiterung für die Fehlerverbesserung nicht anschlägt weil man auf Uniforms oder Attribute zugreift die noch nicht im Code vorhanden sind. Speichert man die kompilierten Dateien separat ab werden einem nur diese fehlerfrei angezeigt, aber die Datei an der man eigentlich arbeit nicht.

Fazit

Der Umgang mit Shadern in FUDGE ist um einiges einfacher geworden. GLSL-Code kann nun tatsächlich in GLSL-Dateien geschrieben werden und muss nicht in Form von Strings in TypeScript-Dateien editiert werden. Dadurch können Entwickler*innen die Vorteile von Erweiterungen in ihrer Entwicklungsumgebung nutzen um ihren Code lesbarer zu machen und bei der Fehlersuche unterstützt zu werden. Das neue System wurde außerdem genutzt um Gouraud- und Phong-Shader zu schreiben, die zur Auswahl der Standard-Shader in FUDGE gehören sollen. Dabei ist aufgefallen, dass in FUDGE noch Features fehlen um diese Shader implementieren zu können. Dazu gehörte die Berechnung der Vertexnormalen von Meshes, die für die Darstellung weicher Schattierungen nötig sind. Im Rahmen dieser Arbeit wurden die entsprechenden TypeScript-Klassen erweitert und beinhalten nun auch die Methoden die für die neuen Shader nötig waren. Durch das neue Shader-Modul-System können Entwickler*innen die Anwendungen mit FUDGE erstellen eigene Shader schreiben und haben dabei Zugriff auf Shader-Module, also Code-Bausteine, die ihnen die Arbeit erleichtern. Die zukünftige Arbeit mit und an FUDGE wird zeigen welche Module implementiert sein sollten und in welcher Form man das System noch Anpassen kann. Insgesamt wird die Erstellung neuer FUDGE-Shader nun deutlich schneller und einfacher ablaufen als mit dem alten System.

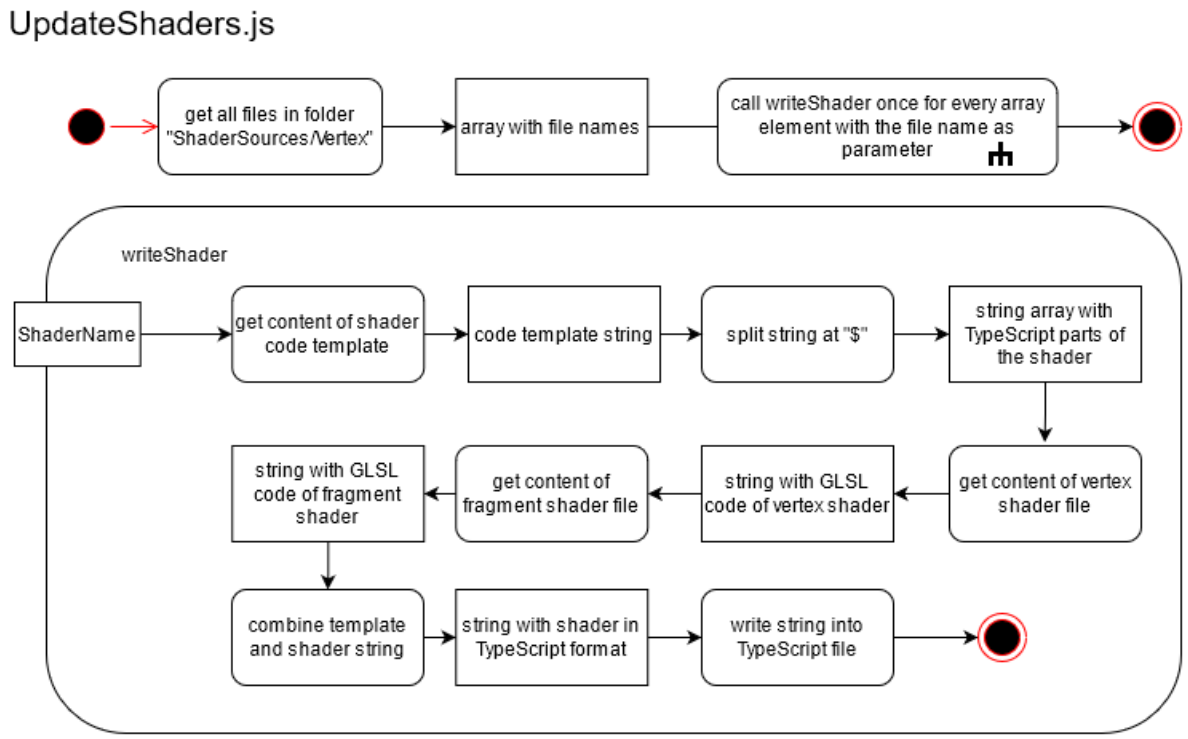
Literaturverzeichnis

- Bailey, Michael & Cunningham, Steve (2012) Graphics shaders : theory and practice (2nd ed.), Boca Raton, FL : CRC Press
- Danchilla, Brian (2012) Beginning WebGL for HTML5, Berkeley, CA : Apress
- Rodríguez, Jacobo (2013) GLSL essentials : enrich your 3D scenes with the power of GLSL!, Packt Pub. : Birmingham, UK
- Halladay, Kyle (2019) Practical shader development : vertex and fragment shaders for game developers, Apress : California
- Matsuda, Kouchi & Lea, Rodger (2013) WebGL programming guide : interactive 3D graphics programming with WebGL, Addison-Wesley : Upper Saddle River, NJ
- Gambetta, Gabriel (2021) Computer Graphics from Scratch (1st edition), No Starch Press : Erscheinungsort nicht ermittelbar, Safari : Boston, MA
- Jirka Dell'Oro-Friedl (2021) FUDGE Quellcode
  https://github.com/JirkaDellOro/FUDGE
  Letzter Abruf: 25.06.2021
- Jirka Dell'Oro-Friedl (2021) FUDGE Wiki
  https://github.com/JirkaDellOro/FUDGE/wiki
  Letzter Abruf: 25.06.2021
- Gregg Tavares (2021) WebGL Fundamentals
  https://webglfundamentals.org/webgl/lessons/webgl-fundamentals.html
  Letzter Abruf: 18.05.2021
- Gregg Tavares (2021) WebGL How It Works
  https://webglfundamentals.org/webgl/lessons/webgl-how-it-works.html
  Letzter Abruf: 18.05.2021
- Gregg Tavares (2021) WebGL Shaders and GLSL
  https://webglfundamentals.org/webgl/lessons/webgl-shaders-and-glsl.html
  Letzter Abruf: 18.05.2021
- University of Marburg (2020) WebGL Example: Phong / Blinn Phong Shading
  https://www.mathematik.uni-marburg.de/~thormae/lectures/graphics1/code/WebGLShaderLightMat/ShaderLightMat.html
  Letzter Abruf: 02.05.2021
- W3Schools (2021) Node.js File System Module
  https://www.w3schools.com/nodejs/nodejs_filesystem.asp
  Letzter Abruf: 20.05.2021
- Rácz Zalán (2020) WebGL GLSL Editor
  https://marketplace.visualstudio.com/items?itemName=raczzalan.webgl-glsl-editor
  Letzter Abruf: 10.06.2021
- pucelle (2019) GLSL Include
  https://marketplace.visualstudio.com/items?itemName=pucelle.glsl-include
  Letzter Abruf: 10.06.2021

Anhang

Hier zu finden ist im Rahmen dieser Arbeit erstellter Code, der zuvor nicht oder nur teilweise gezeigt wurde.

UpdateShaders.js




```
/**
 * Updates the TypeScript files of all shaders to apply changes in the
GLSL files
 * (currently uses the files in folder 'ShaderSources/Vertex/' to
determine the existing shaders)
 * @authors Luis Keck, HFU, 2021
 */
var fs = require("fs");

var files = fs.readdirSync("ShaderSources/Vertex/");//writeShader() is
called once for every file in this folder
for (var i = 0; i < files.length; i++) {
    writeShader(files[i].split(".")[0]);
}

function writeShader(ShaderName) {

    //get the TypeScript code
    fs.readFile("ShaderCodeTemplate.txt", function (err, data) {
        if (err) {
            return console.error(err);
        }
        var code = data.toString().split("$");
```

```
        //get the vertex shader
        fs.readFile("ShaderSources/Vertex/" + ShaderName + ".vert",
            function (err, data) {
            if (err) {
                return console.error(err);
            }
            var vert = data.toString();

            //clean up vertex shader by removing unnecessary lines
            var vertClean = vert.split("#define GLSLIFY 1\n");
            vert = "";
            for (var i = 0; i < vertClean.length; i++)
                vert += vertClean[i];

            //get the fragment shader
            fs.readFile("ShaderSources/Fragment/" + ShaderName +
".frag",
            function (err, data) {
                if (err) {
                    return console.error(err);
                }
                var frag = data.toString();

                //clean up fragment shader by removing unnecessary
lines
                var fragClean = frag.split("#define GLSLIFY 1");
                frag = "";
                for (var j = 0; j < fragClean.length; j++)
                    frag += fragClean[j];

                //combine TypeScript code, vertex shader and fragment
                 shader and write in TypeScript file
                fs.writeFile("Shaders/" + ShaderName + ".ts", code[0] +
                 ShaderName + code[1] + ShaderName + code[2] + vert +
                 code[3] + frag + code[4], function (err) {
                    if (err) {
                        return console.error(err);
                    }
                });
            })
        })
    })
}
```

ShaderGouraud.vert

```
#version 300 es
struct LightAmbient {
    vec4 color;
};
struct LightDirectional {
    vec4 color;
    vec3 direction;
};
const uint MAX_LIGHTS_DIRECTIONAL = 10u;
```

```
uniform LightAmbient u_ambient;
uniform uint u_nLightsDirectional;
uniform LightDirectional u_directional[MAX_LIGHTS_DIRECTIONAL];


vec3 calculateReflection(vec3 light_dir, vec3 view_dir, vec3 normal,
                         float shininess) {
    vec3 color = vec3(1);
    vec3 R = reflect(-light_dir, normal);
    float spec_dot = max(dot(R, view_dir), 0.0);
    color += pow(spec_dot, shininess);
    return color;
}

in vec3 a_position;
in vec3 a_normalVertex;
uniform mat4 u_world;
uniform mat4 u_projection;
uniform mat4 u_normal;
uniform float u_shininess;
out vec4 v_color;

void main() {
    gl_Position = u_projection * vec4(a_position, 1);
    vec4 v_position4 = u_world * vec4(a_normalVertex, 1);
    vec3 v_position = vec3(v_position4) / v_position4.w;
    vec3 N = normalize(vec3(u_normal * vec4(a_normalVertex, 0)));

    v_color = u_ambient.color;
    for(uint i = 0u; i < u_nLightsDirectional; i++) {
        vec3 light_dir = normalize(-u_directional[i].direction);
        vec3 view_dir = normalize(v_position);

        float illuminance = dot(light_dir, N);
        if(illuminance > 0.0) {
            vec3 reflection = calculateReflection(light_dir, view_dir,
N,
                                  u_shininess);
            v_color += vec4(reflection, 1) * illuminance *
                       u_directional[i].color;
        }
    }
    v_color.a = 1.0;
}
```

ShaderGouraud.frag

```
#version 300 es
precision highp float;
uniform vec4 u_color;

in vec4 v_color;
out vec4 frag;

void main()
{
    frag = u_color * v_color;
}
```

ShaderPhong.vert

```
#version 300 es
precision highp float;

in vec3 a_position;
in vec3 a_normalVertex;
uniform mat4 u_world;
uniform mat4 u_projection;
uniform mat4 u_normal;

out vec3 f_normal;
out vec3 v_position;

void main() {
  f_normal = vec3(u_normal * vec4(a_normalVertex, 0.0));
  vec4 v_position4 = u_world * vec4(a_position, 1.0);
  v_position = vec3(v_position4) / v_position4.w;
  gl_Position = u_projection * vec4(a_position, 1.0);
}
```

ShaderPhong.frag

```
#version 300 es
precision highp float;

struct LightAmbient {
    vec4 color;
};
struct LightDirectional {
    vec4 color;
    vec3 direction;
};

const uint MAX_LIGHTS_DIRECTIONAL = 10u;
uniform LightAmbient u_ambient;
uniform uint u_nLightsDirectional;
uniform LightDirectional u_directional[MAX_LIGHTS_DIRECTIONAL];

in vec3 f_normal;
in vec3 v_position;
uniform vec4 u_color;
uniform float u_shininess;
out vec4 frag;

vec3 calculateReflection(vec3 light_dir, vec3 view_dir, vec3 normal,
                         float shininess) {
    vec3 color = vec3(1);
    vec3 R = reflect(-light_dir, normal);
    float spec_dot = max(dot(R, view_dir), 0.0);
    color += pow(spec_dot, shininess);
    return color;
}
```

```
void main() {
    frag = u_ambient.color;
    for(uint i = 0u; i < u_nLightsDirectional; i++) {
        vec3 light_dir = normalize(-u_directional[i].direction);
        vec3 view_dir = normalize(v_position);
        vec3 N = normalize(f_normal);

        float illuminance = dot(light_dir, N);
        if(illuminance > 0.0) {
            vec3 reflection = calculateReflection(light_dir, view_dir,
N,
                                u_shininess);
            frag += vec4(reflection, 1.0) * illuminance *
                   u_directional[i].color;
        }
    }
    frag *= u_color;
    frag.a = 1.0;
}
```

createVertexNormals()

```
protected createVertexNormals(): Float32Array {
      let normals: Vector3[] = [];
      let faceCrossProducts: Float32Array = this.faceCrossProducts;

      for (let v: number = 0; v < this.vertices.length; v += 3)
        normals.push(Vector3.ZERO());

      for (let i: number = 0; i < this.indices.length; i += 3) {
        let trigon: number[] = [this.indices[i], this.indices[i + 1],
                                this.indices[i + 2]];
        let index: number = trigon[2] * 3;
        let normalFace: Vector3 = new Vector3(faceCrossProducts[index],
            faceCrossProducts[index + 1], faceCrossProducts[index + 2]);

        for (let t: number = 0; t < trigon.length; t++)
          normals[trigon[t]] = Vector3.SUM(normals[trigon[t]],
normalFace);
      }
      let vertexNormals: number[] = [];
      for (let n: number = 0; n < normals.length; n++) {
        if (normals[n].magnitude != 0)
          normals[n] = Vector3.NORMALIZATION(normals[n]);
        vertexNormals.push(normals[n].x, normals[n].y, normals[n].z);
      }
      return new Float32Array(vertexNormals);
}
```

Custom Shader Beispiel

Kombination aus Modulen und GLSL-Code um einen Flat-Shader zu erstellen.

```
namespace Test {
    import fudge = FudgeCore;

    export abstract class ShaderCustom extends fudge.Shader {
      public static readonly iSubclass: number =
                              fudge.Shader.registerSubclass(ShaderCustom)
                              ;

      public static vertexShaderSource: string =
        fudge.SHADER_MODULE.HEAD_VERT +
        fudge.SHADER_MODULE.LIGHTS +
        fudge.SHADER_MODULE.NORMAL_FACE +
        `uniform mat4 u_world;` +
        fudge.SHADER_MODULE.MATRIX_PROJECTION +
        fudge.SHADER_MODULE.COLOR_OUT_FLAT +
        fudge.SHADER_MODULE.FLAT_MAIN_VERT;

      public static fragmentShaderSource: string =
        fudge.SHADER_MODULE.HEAD_FRAG +
        fudge.SHADER_MODULE.COLOR_IN_FLAT +
        fudge.SHADER_MODULE.COLOR_U +
        fudge.SHADER_MODULE.FRAG_OUT +
        `void main() {
          frag = u_color * v_color;
        }`;
    }
}
```

UpdateShaderModules.js

```
var fs = require("fs");
fs.readFile("ShaderSources/ShaderModules.glsl", function (err, data) {
    if (err) {
        return console.error(err);
    }
    var modules = ["namespace FudgeCore {\nexport enum SHADER_MODULE
                  {\n\n"];
    var src = data.toString().split("//");
    for (var i = 0; i < src.length - 1; i += 2) {
        var noBreak = src[i+1].split("\n");
        var withBreaks = "";
        for(var a = 1; a < noBreak.length; a++) {
            if (a != noBreak.length - 1)
                withBreaks += noBreak[a] + "\n";
            else
                withBreaks += noBreak[a];
        }

        if (i < src.length - 2)
            modules.push(src[i] + " = `" + withBreaks + "`,");
        else
```

```
            modules.push(src[i] + " = `" + withBreaks + "`");
    }
    modules.push("\n}\n}");
    var out = "";
    for (var j = 0; j < modules.length; j++)
        out += modules[j];

    fs.writeFile("ShaderSources/ShaderModules.ts", out, function (err)
{
        if (err) {
            return console.error(err);
        }
    });
});
```

ShaderModules.glsl

Nur eine kleine Anzahl an Modulen als Beispiel.

```
HEAD_VERT//
#version 300 es
precision highp float;
in vec3 a_position;//

LIGHTS//
struct LightAmbient {
    vec4 color;
};
struct LightDirectional {
    vec4 color;
    vec3 direction;
};
const uint MAX_LIGHTS_DIRECTIONAL = 10u;
uniform LightAmbient u_ambient;
uniform uint u_nLightsDirectional;
uniform LightDirectional u_directional[MAX_LIGHTS_DIRECTIONAL];//

REFLECTION//
vec3 calculateReflection(vec3 light_dir, vec3 view_dir, vec3 normal,
float shininess) {
    vec3 color = vec3(1);
    vec3 R = reflect(-light_dir, normal);
    float spec_dot = max(dot(R, view_dir), 0.0);
    color += pow(spec_dot, shininess);
    return color;
}//
```

ShaderModules.ts

Nur eine kleine Anzahl an Modulen als Beispiel.

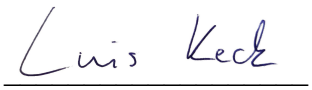
```
namespace FudgeCore {
export enum SHADER_MODULE {

HEAD_VERT = `#version 300 es
precision highp float;
```

```
in vec3 a_position;`,

LIGHTS = `struct LightAmbient {
    vec4 color;
};
struct LightDirectional {
    vec4 color;
    vec3 direction;
};
const uint MAX_LIGHTS_DIRECTIONAL = 10u;
uniform LightAmbient u_ambient;
uniform uint u_nLightsDirectional;
uniform LightDirectional u_directional[MAX_LIGHTS_DIRECTIONAL];`,

REFLECTION = `vec3 calculateReflection(vec3 light_dir, vec3 view_dir,
vec3 normal, float shininess) {
    vec3 color = vec3(1);
    vec3 R = reflect(-light_dir, normal);
    float spec_dot = max(dot(R, view_dir), 0.0);
    color += pow(spec_dot, shininess);
    return color;
}`
}
}
```

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit eigenständig und ohne fremde Hilfe angefertigt habe. Textpassagen, die wörtlich oder dem Sinn nach auf Publikationen oder Vorträgen anderer Autoren beruhen, sind als solche kenntlich gemacht. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Meersburg, 30.06.2021



Luis Keck