

Maturaarbeit

Programmierung einer physikalisch basierten Render-Engine

Manuel Baumann

Betreut durch
Stefan Rothe

13. November 2019



Gymnasium Kirchenfeld
Abteilung MN
Klasse M20C

Inhaltsverzeichnis

1 Einleitung	4
1.1 Motivation	5
1.2 Aufbau	5
2 Geschichte	6
3 Technologien	7
3.1 Unity 3D	7
3.2 HLSL	7
3.3 C-Sharp (C#)	8
4 Grundlagen	9
4.1 Physikalisch basiertes Rendern (ray tracing algorithm)	9
4.2 HLSL Basics	10
4.3 Koordinatensysteme	10
4.3.1 Welt-Koordinatensystem	10
4.3.2 Kamera-Koordinatensystem	11
4.3.3 Räumliche Polarkoordinaten (Kugelkoordinaten)	11
4.3.4 Homogene Koordinaten	11
5 Strahlen simulieren	12
5.1 Ziel	12
5.2 Theorie	12
5.2.1 Vom Augpunkt zum Bild	12
5.2.2 Von den Strahlen zu den Pixeln	13
5.3 Implementierung	13
5.3.1 Strahlen	13
6 Hintergrund	15
6.1 Theorie	15
6.1.1 Hintergrund abbilden	15
6.2 Implementierung	15
6.2.1 Hintergrund	15
7 Einfache Objekte im dreidimensionalen Raum erstellen	17
7.1 Ziel	17
7.2 Theorie	17
7.2.1 Grundlegender Aufbau	17
7.2.2 Schnittpunkte	18
7.2.3 Objekte darstellen	18
7.3 Implementierung	19
7.3.1 Schnittpunkt	19
7.3.2 Neue Funktionen	19

7.3.3	Ebene	19
7.3.4	Sphäre/Kugel	21
8	Bewegungssystem	23
8.1	Ziel	23
8.2	Theorie	23
8.2.1	Wie funktioniert es?	23
8.2.2	Bewegen	23
8.2.3	Drehen	24
8.3	Implementierung	24
9	Ausblick	26
10	Definitionsverzeichnis	28

1 Einleitung

In dieser Arbeit soll dargelegt werden, wie man auf einfachster Ebene die Lichtstrahlen der Natur mittels eines Programmes reproduzieren kann. Ein Computerprogramm stellt die Grundlage für den Verstehensprozess dar. Das Ziel der Arbeit besteht darin, die verschiedenen Komponenten, die für Teile des Renderingprozesses zuständig sind, Schritt für Schritt in einem Programmiercode aufzubauen. Der Programmcode ist unter <https://github.com/Edornel10/Render-Engine-Maturaarbeit.git> erhältlich. Er umfasst folgende Teile für das Bereitstellen einer Grundstruktur:

1. Strahlen simulieren
2. Berechnen der Schnittpunkte
3. Auswählen und Darstellung der Schnittpunkte

Diese Teile sind nur dazu da, die Render-Engine zum Laufen zu bringen. Sie können beliebig erweitert werden, um die Endresultate des Programmes realistischer zu machen oder den Renderingprozess zu beschleunigen. Dadurch, dass diese Programme so stark erweiterbar sind, können sie auch immer mehr entwickelt werden. Dieser Tatsache haben wir zu verdanken, dass Filme immer realistischer dargestellt werden.



Abbildung 1.1: Eine mit physikalisch basiertem Rendern komplett erzeugte Szene.

1.1 Motivation

Physikalisches Rendern ist ein sehr schwieriger, aber auch interessanter Vorgang. Das Thema hat sehr viel mit Mathematik zu tun und beinhaltet neue Konzepte, die wir in der Schule noch nicht behandelt haben und werden. Dementsprechend hat diese Herausforderung für mich einen geheimnisvollen Glanz; Ich will die Renderingwelt besser kennenlernen. In meiner Freizeit programmiere ich gerne und hoffe dort, das Thema weiter vertiefen zu können. Ich wünsche Ihnen viel Spaß beim Kennenlernen dieser spannenden Welt.

1.2 Aufbau

Damit die Maturaarbeit so verständlich wie möglich ist, werden im Kapitel 3 alle Technologien kurz erklärt. Im Kapitel 4 werden die Grundlagen gesetzt. Weiter lernen Sie im Hauptteil in den Kapiteln: 5, 6, 7 und 8 die Welt der Strahlen und deren Schnittpunkte kennen. Dort wird immer zuerst die Theorie erklärt und diese dann mit Codebeispielen praktisch eingeführt. Zum Schluss wird dann noch von Erweiterungen der Rendermaschine gesprochen. Im Definitionsverzeichnis 10 hinten in der Arbeit können wichtige Fachbegriffe nachgeschaut werden. Dabei handelt es sich meistens um Begriffe, die vor allem in der Programmierwelt von Bedeutung sind.

2 Geschichte

1970, in den ersten Jahren der Computergrafik, waren die Ressourcen noch sehr begrenzt. Einen Computer mit 1 MB RAM aufzutreiben war sehr teuer und die verfügbare Rechenleistung war dementsprechend auf einem schlechten Niveau. Diese Rechengrundlagen reichten nicht für komplizierte, physikalische Render-Annäherungen. Mit diesen Voraussetzungen wäre ich nicht in der Lage eine physikalische Render-Engine zu programmieren. Andere, beschränkt brauchbare Rendertechniken unterlagen dem Blinschen Gesetz: «As technology advances, rendering time remains constant». Ein Bild braucht immer ungefähr gleich viel Zeit, um generiert zu werden, egal wie gut die dazu verwendete Technologie ist. Die zusätzliche Rechenpower wird dabei in ein realistischeres Ergebnis gesetzt und nicht dafür verwendet den Vorgang zu beschleunigen. Wenn also die zusätzlichen Mittel nur in die Verbesserung des Prozesses investiert werden, müssen die Resultate immer unbefriedigend oder auch zu wenig realistisch ausfallen. Die Forscher wollten also immer ein besseres Bild rendern, als es die Technologie in dieser Zeit erlaubte. 1980 kam die erste physikalische Rendermaschine von Turner Whitted auf den Markt, die als erste in praktischen Bereichen Anwendung fand und nicht nur zu reinen Forschungszwecken verwendet wurde. Der physikalische Renderansatz wurde seither verbessert, was der Filmindustrie aber nicht wirklich reichte. Zu einer signifikanten Verbesserung kam es erst, als der Arnold-Renderer 2001 von Marco Fajardo entwickelt wurde. Zur gleichen Zeit wurde auch die erste Echtzeit-Rendermaschine von Gregory J. Ward und seinen Kollegen entwickelt. Zur Entstehungszeit barg physikalisch basiertes Rendern hauptsächlich drei grosse Vorteile gegenüber anderen Rendertechniken: [2]:

1. Es ist möglich eine Szene schnell zu rendern, um z.B. die Objekte im Raum besser platzieren zu können. Gleichzeitig ist es auch möglich, sehr realistische Bilder mit der gleichen Rendermaschine herzustellen, indem mehr Zeit ins Rendering investiert wird.
2. Damit die Objekte sich realistisch in eine Umgebung eingefügt haben, mussten vor der Zeit von physikalisch basiertem Rendern Oberflächen angepasst werden. Dies war meistens ein sehr langwieriger und auch langweiliger Prozess. Wenn das Objekt mit seinen Einstellungen in eine andere Umgebung verfrachtet wurde, waren alle zuvor getroffenen Lichteinstellungen falsch. So mussten alle Parameter neu eingestellt werden. Physikalisches Rendern bedarf keiner solchen Einstellungen und funktioniert somit fast überall ohne Probleme.
3. Die Schatten von physikalisch basiertem Rendern sehen ohne grosse Nachbearbeitung sehr echt aus. Die Schatten müssen bei physikalischen Annäherungen nicht manuell nachbearbeitet werden, wie es bei den anderen Rendern der Fall ist.

3 Technologien

3.1 Unity 3D

Ich schreibe das Programm in Unity 3D. Es erleichtert den Arbeitsprozess, weil es mir Grundbausteine zu Verfügung stellt. Diese Grundbausteine müssen daher nicht mehr selber programmiert werden. Teile dieser Grundbausteine sind:

1. In Unity können Objekte in Echtzeit verschoben oder verändert werden. In konventionellen Programmen müssen z.B. die Positions-werte einer Kugel im Code angegeben werden. Dann wird das Programm gestartet und man merkt, die Kugel steht am falschen Ort. Die Änderung der Position der Kugel können in Unity bequem durch ein Fenster beobachtet werden, ohne das Programm starten zu müssen.
2. Unity sorgt für ein geordnetes Projekt. Unity zwingt den Nutzer dazu, die Programme aufzu-teilen und auf verschiedene Objekte zu setzen. Dank diesem durchdachten System wird das ganze Programm übersichtlicher und geordneter.
3. Unity unterstützt die Programmiersprache HLSL.

Das Programm basiert auf der Unity-Version 2019.2.2f1.

3.2 HLSL

HLSL ist eine für DirectX entwickelte Programmiersprache. DirectX sind auf Windows präsente Programmierschnittstellen. Sie dienen zur Anbindung von Programmen an das Computersystem, um Grafik, Audio oder Eingaben kontrollieren zu können. HLSL steht abgekürzt für «High Level Shading Language». Sie dient dazu, «Shaders» zu programmieren. «Shaders» sind kleine Recheneinheiten, die für die Grafik auf dem Bildschirm zuständig sind. Eine «High-Level-Sprache» ist eine Programmiersprache, die für Menschen verständlich ist. Sie besteht aus sogenannten höheren Befehlen. Ein höherer Befehl führt sehr viele kleine Befehle auf dem Computer aus, ohne dass wir diese ausschreiben müssen. Das macht es weniger kompliziert und auch verständlicher. HLSL wurde extra für Shader-Programme entwickelt. Mit der Kombination von Unity erleichtert sie den ganzen Programmierprozess. [4]

3.3 C-Sharp (C#)

C#, sprich C-Sharp ist eine weit verbreitete, objektorientierte Allzweckprogrammiersprache. Sie wurde im Jahr 2000 im Auftrag von Microsoft von einem Team um Anders Hejlsberg entwickelt. Am Anfang nutzte man sie hauptsächlich für Microsoft-Software-Plattformen. Die Sprache wurde jedoch in anderen Bereichen immer attraktiver, bis sie zur Allzwecksprache wurde. C-Sharp ist genauso wie HLSL eine höhere Programmiersprache. Die Sprache wird für das Programm genutzt, da sie von Unity unterstützt wird. Java-Script könnte ebenso benutzt werden. [3]

4 Grundlagen

4.1 Physikalisch basiertes Rendern (ray tracing algorithm)

Physikalisch basiertes Rendern verwendet eine Lichtstrahlsimulierung, um eine 3D-Szene in ein 2D-Bild zu synthetisieren. Diese Methode nennt sich «ray tracing algorithm». Eine 3D-Szene ist ein virtueller Raum, in dem virtuelle Objekte vorhanden sind. Es werden Strahlen simuliert, die die Umgebung in der Szene erforschen. Sie können Farb- und Helligkeitswerte in die Pixel schreiben. Diese Strahlen unterliegen virtuell den Gesetzen der Physik. Deswegen sind die Bewegungen der Strahlen verständlich für den Menschen. Das steht im Gegensatz zu anderen Renderannäherungen, die zum Teil nur aus mathematischen Formeln bestehen. Also sind sie unverständlich für Menschen. Die Lichtstrahlsimulierung ist aber nur eine der Möglichkeiten eine Szene physikalisch zu rendern. Die zweite Methode benutzt Bilder, um die Szene echt aussehen zu lassen. Es werden tausende Bilder von einem Objekt aus allen Perspektiven gemacht. In den Bildern wird das Objekt und dessen Schatten ganzheitlich erfasst. Im Renderprozess werden diese Informationen dann genutzt, um jeder Perspektive ein Bild zuweisen zu können. Wie auf den Bildern sehen die Resultate mit dieser Methode echt aus. Der Nachteil ist aber, wenn ein gutes Resultat erzielt werden will, müssen sehr viele Fotos gemacht werden.

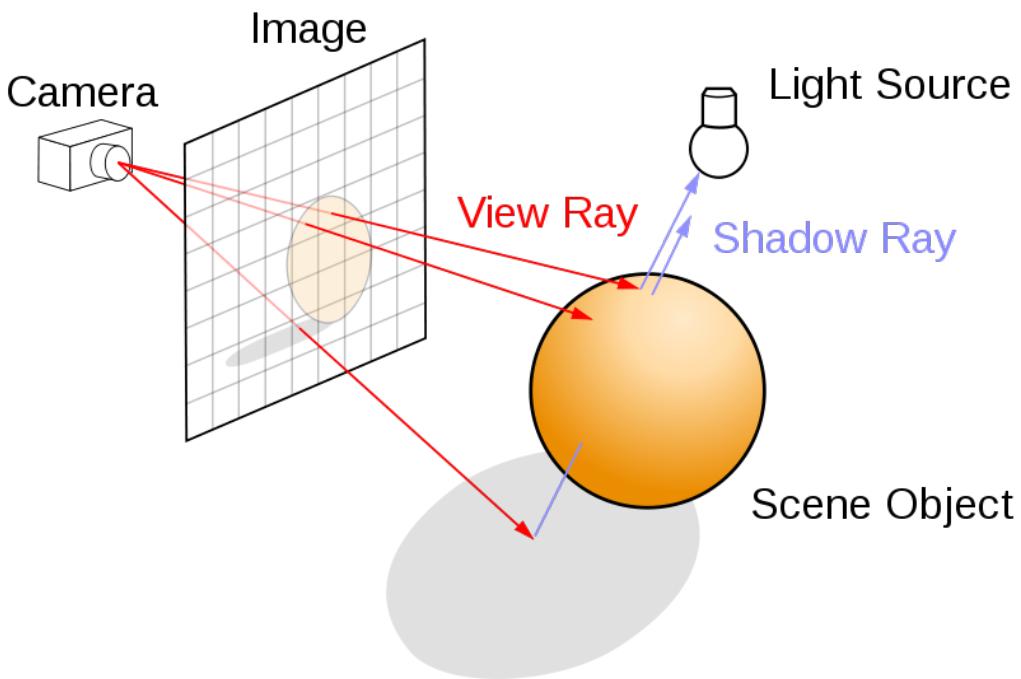


Abbildung 4.1: Es gehen Strahlen durch die Bildebene zu den Objekten, um den Farbwert des Objektes in den Pixel der jeweiligen Strahlen zu schreiben. Auf der Bildebene ist schon die Kugel und dessen Schatten sichtbar.

4.2 HLSL Basics

Um das Programm und seine Funktionsweise zu verstehen, muss die Grundstruktur von HLSL verstanden werden. HLSL vereinfacht das Schreiben eines Shaders. Es gibt eine Funktion, die für jeden Pixel einmal ausgeführt wird. Eine Funktion ist eine kleine Einheit im Code, die sich um eine Aufgabe kümmert. Sie kann aufgerufen werden, damit sie ihre Aufgabe erledigt oder sie kann Werte zurückgeben. Wenn die Funktion ausgeführt wird, kann ein Strahl simuliert werden. Dieser Strahl ist dafür zuständig, für den jeweiligen Pixel einen Farbwert herauszufinden. Das Programm kann so in Einzelteile zerlegt werden und sich immer nur auf einen Strahl konzentrieren. Die Ressourcen des Computers werden sehr effizient genutzt, weil diese Funktion extra dafür optimiert wurde. [4]

4.3 Koordinatensysteme

Ein Koordinatensystem beschreibt wo sich welches Objekt befindet. Mit dem herkömmlichen Koordinatensystem können die Positionen von Punkten einfach beschrieben werden. Der Punkt P1 befindet sich an Ort (3,2). Es folgen Beschreibungen von drei Koordinatensystemen in den Kapiteln 4.3.1, 4.3.2 und 4.3.3. Zum Schluss erwähne ich noch ein Attribut in Kapitel 4.3.4, dass eine Koordinate haben kann. Das sind Homogene-Koordinaten.

4.3.1 Welt-Koordinatensystem

Das Welt-Koordinatensystem kennen wir alle unter dem Namen «kartesisches Koordinatensystem». Es kann zwei bis unendlich viele Dimensionen haben und hat einen Nullpunkt. Die Richtungsachsen stehen orthogonal aufeinander. Die Koordinatenlinien sind Geraden. Sie haben immer den gleichen Abstand voneinander. [6]

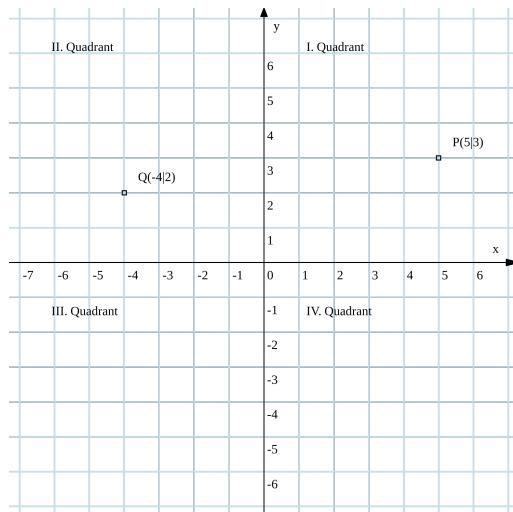


Abbildung 4.2: Ein Koordinatensystem, wie man es aus der Schule kennt. Zusätzlich sieht man noch zwei Punkte. Jeder Quadrant ist beschriftet.

4.3.2 Kamera-Koordinatensystem

Dieses Koordinatensystem ähnelt sehr stark dem Welt-Koordinatensystem. Der Ursprung des Koordinatensystems liegt jedoch auf der Kamera und die Ausrichtung der Achsen legt die Kamera fest. Im Laufe der Arbeit wird noch ein Bewegungssystem hinzugefügt. Damit kann sich die Kamera im Welt-Koordinatensystem bewegen. In diesem Programm muss ich viel Kamera- in Weltkoordinaten und umgekehrt umrechnen. Eine Funktion von Unity bewerkstelligt das automatisch.[2, Kapitel 2.1]

4.3.3 Räumliche Polarkoordinaten (Kugelkoordinaten)

Die räumlichen Polarkoordinaten sind ein Weg, Koordinaten in einem System anders darzustellen. Sie bestehen aus den zwei Winkeln Φ und Θ und dem Abstand «R» vom Ursprung zum Punkt. Θ kann jeden Wert von 0 bis π und Φ jeden von 0 bis 2π in Radianten annehmen. Die Polarkoordinaten werden in Kapitel 6 dazu verwendet, einen Hintergrund um die Kamera herum zu projizieren. Mit ihnen werden Punkte in einem dreidimensionalen Koordinatensystem abgebildet.[7]

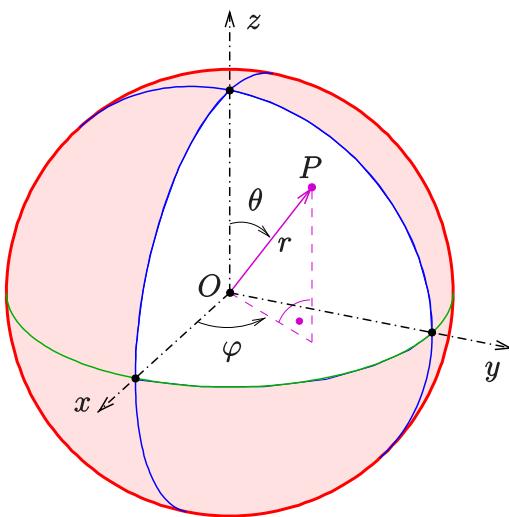


Abbildung 4.3: Zu sehen sind die Polarkoordinaten bestehend aus zwei Winkeln, θ und ϕ und dem Radius r , die die Richtung des Strahles, der zu Punkt P führt, bestimmen.

4.3.4 Homogene Koordinaten

Bei homogenen Koordinaten fügt man eine weitere Dimension hinzu. Diese Dimension wird aber nicht aktiv verwendet. Vielmehr wird sie dafür genutzt, Translationen zu berechnen. Diese Translationen werden mithilfe von Matrizen und linearen Abbildungen beschrieben. Eine Translation ist eine absolute Verschiebung von Punkten oder Objekten. Wenn eine 3D-Koordinate in meinem Programm vier Werte hat, wird dieser vierte Wert nur für Berechnungen genutzt. Er wird meistens nach der Berechnung wieder entfernt. [5]

5 Strahlen simulieren

5.1 Ziel

In diesem Kapitel werden Strahlen eingefügt. Diese stellen die Basis für alle weiter Programmteile dar.

5.2 Theorie

5.2.1 Vom Augpunkt zum Bild

Die Szene wird vom Augpunkt aus gerendert. Von dort aus «schaut» die Kamera auf die Objekte. Die Lochkamera ist die einfachste Annäherung zum System im Code. Sie ist eine der einfachsten Möglichkeiten, ein Bild von der echten Welt zu machen. Die Lochkamera besteht aus einer lichtundurchlässigen Box mit einem kleinen, verdeckten Loch an einem Ende. Wenn ein Bild gemacht werden will, wird dieses Loch aufgedeckt. Das Licht geht durch das Loch, wo es auf das an der anderen Seite befindende Fotopapier trifft. Der Nachteil ist, dass es sehr lange dauert bis genug Licht auf das Fotopapier getroffen ist, um ein Bild entstehen zu lassen.

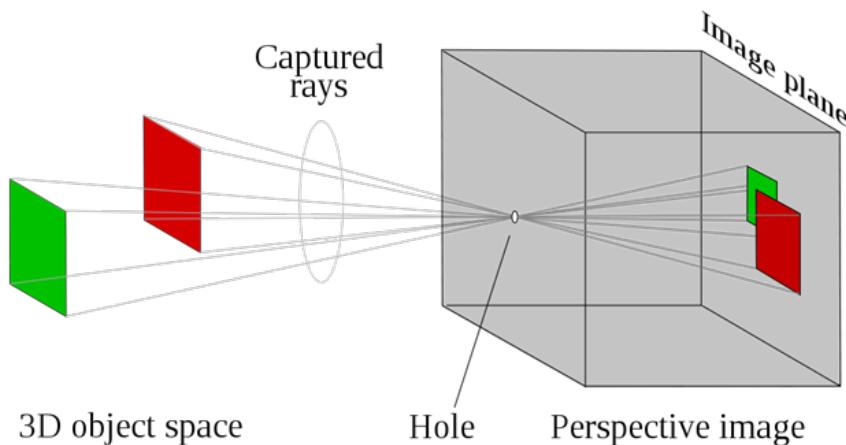


Abbildung 5.1: Ein Beispiel eines Models einer Lochkamera. Links die wirklichen Objekte, die wir darstellen wollen und rechts das Fotopapier.

Dieses Grundsystem ist für Simulationen gut geeignet. Wenn man das Endresultat auf dem Fotopapier anschaut und es mit den sichtbaren Objekten links vom Loch vergleicht, kann man ein Grundprinzip erkennen: Auf dem Bild sieht man nur die Objekte die sich im Kreis «captured rays» befinden. Im Programm wird das sehr ähnlich umgesetzt. Nur werden Strahlen vom Loch aus gegen links projiziert. Das Loch stellt den Augpunkt dar. Die Bildebene wird mit einem gewissen Abstand zum Loch platziert. Das wird hier sichtbar durch die «captured rays» Ebene.

5.2.2 Von den Strahlen zu den Pixeln

Das System mit der Lochkamera kann nicht eins zu eins übernommen werden. Es müssen noch ein paar Vereinfachungen gemacht werden. Bei der Lochkamera gehen sehr viele Lichtstrahlen durch das Loch. Das Programm wäre mit so vielen Lichtstrahlen schnell überfordert. Deswegen werden die Strahlen reduziert, damit nicht zu viele von ihnen berechnet werden müssen. Die Farbinformationen von den Pixeln werden jeweils von einem Strahl pro Pixel gesammelt.

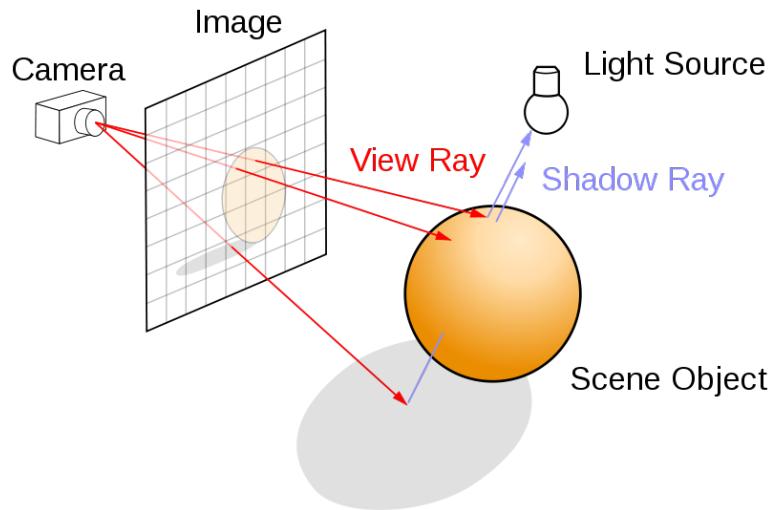


Abbildung 5.2: Die Simulation von Lichtstrahlen. Sie gehen dabei von der Kamera aus durch einen bestimmten Pixel der Bildebene.

5.3 Implementierung

5.3.1 Strahlen

```
struct Ray
{
    float3 origin;
    float3 direction;
};
```

Zuerst wird eine Datenstruktur für die Strahlen erstellt. Er wird wie oben mit dem Codewort «struct» begonnen. In den geschweiften Klammern werden die verschiedenen Variablen definiert. Im Programm besitzen Strahlen zwei Attribute: Einen Startpunkt und eine Richtung. Also wird meine Struktur mit zwei Variablen kreiert. Um die zwei Variablen «origin» und «direction» als einen 3D Punkt zu definieren, wird das Codewort «float3» verwendet. Diese Variablen besitzen noch keinen Wert. Momentan sind sie nur leere Hülsen, die mit Daten gefüllt werden können.

Startpunkt

```
float4 camNull = (0.0f, 0.0f, 0.0f, 1.0f)
float3 origin = mul(_CameraToWorld, CamNull).xyz;
```

Im Programm ist der Startpunkt der Augpunkt, der homogene Nullpunkt des Kamera-Koordinatensystems ist (0,0,0,1). Die ersten drei Nullen beschreiben die Position. Also direkt auf der Kamera. Die vierte Zahl ist die homogene vierte Dimension, die für die Berechnungen gebraucht wird. In der ersten Zeile wird eine 4D-Koordinate «camNull» erstellt. Dann wird diese Variable von Kamerakoordinaten in Weltkoordinaten umgerechnet. Im Programm wird der Nullpunkt des Kamera-Koordinatensystems mal die Umrechnungsmatrix gerechnet. Mit «.xyz» wird die vierte, homogene Koordinate abgeschnitten.

Richtung

```
uint width, height;
Result.GetDimensions(width, height);

float2 uv = float2((id.xy + float2(0.5f, 0.5f)) / float2(width, height) *
2.0f - 1.0f);
```

Um die Richtung des Strahls zu berechnen, braucht man die Position des jeweiligen Pixels. Im Programm werden die 2D-Koordinaten «uv» genannt. «uv» ist ein Objekt, welches zwei Werte speichern kann. Auf der ersten Zeile werden zwei Variablen erstellt. In der zweiten Zeile werden die Variablen mit der Breite und der Höhe des Computerbildschirmes gefüllt. Mit den «uv»-Koordinaten kann ich den Sichtwinkel verändern. Wenn der Nutzer mehr sehen will, muss er nur die x- oder y-Koordinate von «uv» verändern. In der dritten Zeile wird genau das gemacht. Die «uv»-Werte werden an den Bildschirm angepasst.

```
float3 direction = mul(_CameraInverseProjection, float4(uv, 0.0f,
1.0f)).xyz;

direction = mul(_CameraToWorld, float4(direction, 0.0f)).xyz;
direction = normalize(direction);
```

Indem eine Funktion von HLSL einen Strahl vom Augpunkt zu den jeweiligen Pixeln auf der Bildebene projiziert, wird die Richtung zurückgegeben. Das kann man mit der Multiplikation von der «CameralInverseProjections»-Matrix mit den Kamerapixelkoordinaten erreichen. Die Kamerapixelkoordinaten haben eigentlich nur einen x und y Wert, werden aber wegen der Multiplikation mit einer 4 x 4 Matrix als homogene Koordinaten geschrieben. Die Variable «uv» gibt dabei die x und y Position in Pixeln an. Diesen Strahl rechnet man dann von den homogenen Kamera-Koordinaten in die Welt-Koordinaten um. Um schlussendlich allen Strahlen noch die gleiche Länge zu geben, wird der Richtungsvektor normalisiert. Dabei werden die Koordinaten so verändert, dass der Betrag des Vektors eins ergibt. Das wird mit «normalize(direction)» gemacht. Wenn so ein Strahl definiert ist, kann dieser Strahl wie ein Sehstrahl benutzt werden: Alles was die Strahlen sehen, wird auf das 2D-Bild als Farbe aufgetragen.

6 Hintergrund

6.1 Theorie

6.1.1 Hintergrund abbilden

Der Hintergrund wird durch die Funktion SampleLevel, als Kugel um den Augpunkt herum projiziert. Die Funktion SampleLevel kann man sich so vorstellen, dass der Hintergrund als Papier über eine Kugel gestülpt wird. Die Kamera ist in der Mitte dieser Kugel und sieht diesen Hintergrund mit den Strahlen. Die Funktion braucht ein Hintergrundbild und die Polarkoordinaten, um dem Strahl einen Pixel auf dem Bild zuweisen zu können. Die zwei Winkel ϕ und θ müssen als Zahl zwischen null und eins angegeben werden. Darum werden die zwei Winkel noch durch π beziehungsweise durch 2π gerechnet.

6.2 Implementierung

6.2.1 Hintergrund

```
float theta = acos(ray.direction.y) / -PI;
float phi = atan2(ray.direction.x, -ray.direction.z) / -PI * 0.5f;
Result[id.xy] = _SkyboxTexture.SampleLevel(sampler_SkyboxTexture,
    float2(phi, theta), 0).xyz;
```

Mit diesem Programmteil sieht man zum ersten Mal etwas im Programm. Zuvor wurde nur das Grundsystem der Strahlen aufgebaut. Indem die Funktion SampleLevel auf dem Objekt «SkyboxTexture» aufgerufen wird, kann man Punkte auf dem Hintergrundbild bekommen. Die «SkyboxTexture» wurde am Anfang erstellt und beinhaltet nur ein Hintergrundbild. Dazu müssen nur die zuvor berechneten Kugel-Koordinaten als Parameter eingegeben werden. Die zwei Winkel bekommt man mit der Formel $\cos \theta = z$ und $\tan \phi = \frac{y}{x}$. Bis jetzt gibt es aber nur Werte und noch keine Resultate. Also muss in der von HLSL implementierten Variable "Result[id.xy]" die Farbe von jedem Pixel reingeschrieben werden. Das passiert auf der letzten Zeile. Der ganze Prozess findet in der Kernelfunktion «CSMain» statt. Sie dient dazu, die Farben aller Pixel parallel berechnen zu lassen. Das bisher geschriebene Programm wird für jeden Pixel einmal ausgeführt. Hier die ganze Funktion:

```
void CSMain(uint3 id : SV_DispatchThreadID)
{
    uint width, height;
    Result.GetDimensions(width, height);
    float2 uv = float2((id.xy + float2(0.5f, 0.5f)) / float2(width,
        height) * 2.0f - 1.0f);

    Ray ray = CreateCameraRay(uv);

    float theta = acos(ray.direction.y) / -PI;
    float phi = atan2(ray.direction.x, -ray.direction.z) / -PI * 0.5f;
    Result[id.xy] = _SkyboxTexture.SampleLevel(sampler_SkyboxTexture,
        float2(phi, theta), 0);
}
```

Das Resultat sieht so aus:

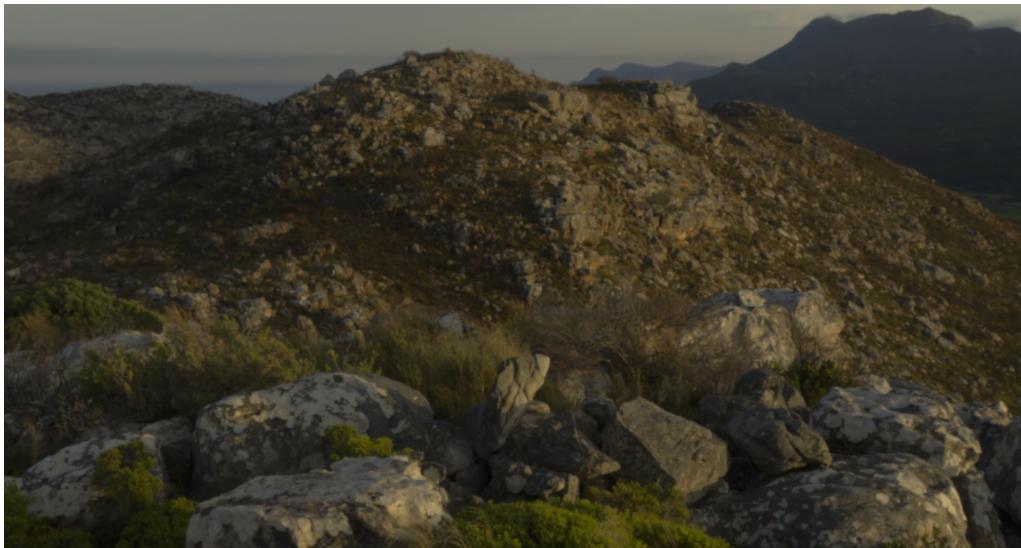


Abbildung 6.1: Eine mit physikalisch basiertem Rendern komplett erzeugte Szene.

7 Einfache Objekte im dreidimensionalen Raum erstellen

7.1 Ziel

Mit diesem Kapitel ist das Programm in der Lage, Objekte abzubilden. Die Grundlagen wurden gelegt. Es existiert das Grundgerüst der Strahlen. Diese Strahlen lernen in diesem Kapitel, mit Objekten zu kollidieren. Daraus leitet das Programm alle wichtigen Informationen ab, um das Objekt abzubilden.

7.2 Theorie

7.2.1 Grundlegender Aufbau

Es braucht zwei Sachen im Programm. Das Programm muss in irgendeiner Form wissen, wo sich die Objekte befinden. Zudem müssen die Strahlen mit den Objekten kollidieren. Im Programm wird es Ebenen und Sphären geben.

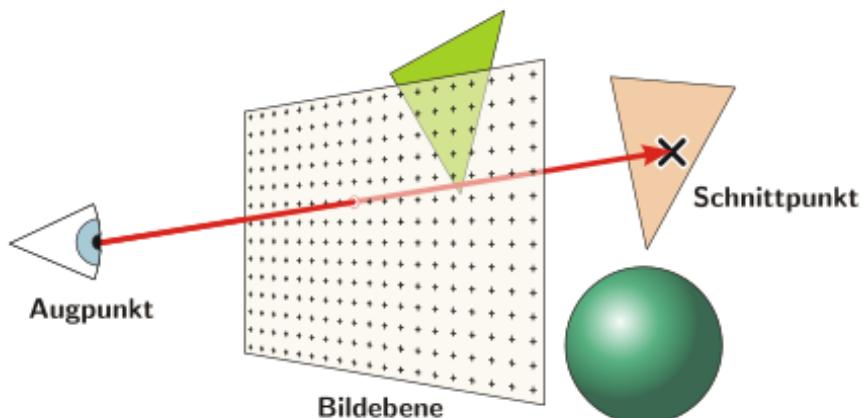


Abbildung 7.1: Vom Augpunkt aus wird ein Strahl simuliert, der durch die Bildebene geht und dann ein Objekt im Raum trifft. Der Pixel, der vom Strahl durchstossen wurde, würde jetzt die Farbe des Objekts bekommen, also hellbraun.

7.2.2 Schnittpunkte

Die Schnittpunkte brauchen drei Informationen, damit den Pixeln auf der Bildebene eine Farbe zugewiesen werden kann. Sie brauchen eine Distanz vom Augpunkt, eine Position und die Normale am Schnittpunkt. Mit der Distanz kann ermittelt werden, welcher Schnittpunkt am nächsten zum Augpunkt steht. Nur der nächste Schnittpunkt wird im Pixel abgebildet. Mit dieser Methode werden nur die Schnittpunkte der nächsten Objekte auf der Bildebene abgebildet. Die Position braucht das Programm, um weiter Informationen über den Punkt zu sammeln. Z.B. kann damit ermittelt werden, ob Licht zu diesem Punkt kommt oder nicht. Aber in diesem Abschnitt wird die Position nur für die Sphäre genutzt. Sie wird für die Berechnung von wichtigen Daten verwendet. Mit der Normale am Schnittpunkt kann das Programm den Pixeln auf der Bildebene die Farbe der Richtung der Normale geben. Somit erscheint ein farbiges Muster auf der Kugel. Das hört sich kompliziert an, ist aber nur eine Methode, die Szene mit Farbe zu verschönern.

7.2.3 Objekte darstellen

Für diese Aufgabe gibt es mehrere mögliche Lösungen. Ein Beispiel wäre Punkte im 3D-Raum zu verbinden und daraus Objekte entstehen zu lassen. Nur würde dabei die Schnittpunktberechnung der Objekte sehr schwierig werden. Deswegen wird das Programm mit einer Vereinfachung geschrieben. Das Programm kann nur einfache Objekte wie Kugeln und Ebenen darstellen. Diese einfachen Objekte haben den Vorteil, dass alle Punkte dieser Objekte durch die gleiche Formel von einem Ausgangspunkt oder auch Mittelpunkt des Objektes berechnet werden können. Kompliziertere Objekte entstehen, indem sie aus einfachen Objekten zusammengesetzt werden.

Ebene

Bei der Ebene gibt es eine einfache Formel, um die Distanz vom Objekt zum Augpunkt herauszufinden:

$$t = \frac{x-, y - \text{oder } z - \text{Startpunkt}}{x-, y - \text{oder } z - \text{Richtung}}$$

«t» steht für die Distanz. Die Position des Schnittpunktes kann mit der Distanz berechnet werden. Dafür folgt man einfach dem Strahl in die Richtung, um die zuvor berechnete Distanz und bekommt so den Punkt.

$$\vec{r} = \vec{p} + t\vec{v}$$

Sphäre/Kugel

Bei der Kugel wird ein komplizierteres Verfahren angewendet. Zuerst muss die Position des Schnittpunktes von dem Strahl und der Kugel herausgefunden werden. Die Formel findet man, indem die Liniengleichung mit der Kreisgleichung gleichgesetzt und ausmultipliziert wird. Daraus ergibt sich folgende Formel.[1, Kapitel 8 und 19]

$$t = -\vec{w} \bullet \vec{v} + -\sqrt{(\vec{w} \bullet \vec{v})^2 - \vec{w} \bullet \vec{w} + r^2}$$

«v» ist die Richtung des Strahles. «w» ist der Vektor vom Mittelpunkt der Kugel zum Augpunkt. «r» ist der Radius der Kugel.

7.3 Implementierung

7.3.1 Schnittpunkt

```
struct RayHit
{
    float3 position;
    float distance;
    float3 normal;
};
```

Es wird eine zweite Struktur definiert. In ihr können die drei zuvor genannten Informationen gespeichert werden: Position, Distanz und Normale.

7.3.2 Neue Funktionen

Damit das Programm verständlicher wird, wird es in Funktionen aufgeteilt. Es werden zwei neue Funktionen neben der Kernelfunktion eingeführt. Diese Kernelfunktion wird für jeden Pixel einmal ausgeführt und schreibt die Farbwerte in die Pixel.

```
Trace(ray);
Shade(ray, hit);
```

Die Funktion «Trace» ermittelt die Schnittpunkte der Strahlen mit den Objekten. Er gibt diese Information mit der zuvor erstellten Struktur «RayHit» zurück. Die Funktion «Shade» gibt die Farbwerte des Hintergrundes zurück, wenn nichts getroffen wurde.

7.3.3 Ebene

```
float t = -ray.origin.y / ray.direction.y;
```

Hier wird die in Kapitel 7.2.3 beschriebene Formel ausgerechnet. Wenn die y-Koordinate verwendet wird, wird π_1 abgebildet. Das ist die xz-Ebene. Wenn die x- oder z-Koordinaten verwendet werden, wird π_2 oder π_3 abgebildet.

```
if (t > 0 && t < bestHit.distance)
{
    // definiere RayHit
}
```

Es wird evaluiert, ob dieser Schnittpunkt am nächsten am Augpunkt ist. Das wird mit einer Verzweigung gemacht. Dazu wird abgefragt, ob die Distanz in der Struktur «RayHit», namens «bestHit»,

grösser ist als die gerade berechnete Distanz. Wenn noch kein Schnittpunkt in «RayHit» gespeichert ist, ist die Distanz unendlich. Somit ist jeder Punkt näher. Wenn aber ein näherer Schnittpunkt schon reingeschrieben wurde, wird der nächste Teil des Programmes ignoriert.

```
bestHit.distance = t;
bestHit.position = ray.origin + t * ray.direction;
bestHit.normal = float3(0.0f, 1.0f, 0.0f);
```

Es werden die Daten des Schnittpunktes in die Struktur «RayHit» namens «bestHit» reingeschrieben. Die Position wird mit der in Kapitel 7.2.3 gezeigten Formel berechnet. Die Normale bei π_1 zeigt nach oben. Deswegen ist der Vektor (0,1,0). Die Ebene sieht im Programm so aus.

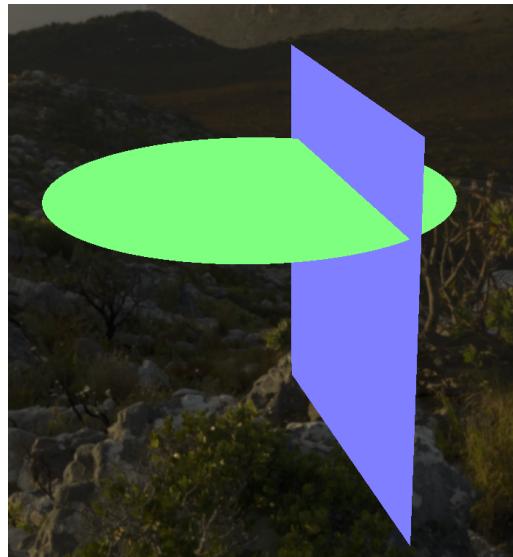


Abbildung 7.2: Eine mit physikalisch basiertem Rendern komplett erzeugte Szene.

7.3.4 Sphäre/Kugel

```
void IntersectSphere(Ray ray, inout RayHit bestHit, float4 sphere)
{
    float3 d = ray.origin - sphere.xyz;
    float p1 = -dot(ray.direction, d);
    float p2sqr = p1 * p1 - dot(d, d) + sphere.w * sphere.w;
    if (p2sqr < 0)
        return;

    float p2 = sqrt(p2sqr);
    float t = p1 - p2 > 0 ? p1 - p2 : p1 + p2;
    if (t > 0 && t < bestHit.distance)
    {
        bestHit.distance = t;
        bestHit.position = ray.origin + t * ray.direction;
        bestHit.normal = normalize(bestHit.position - sphere.xyz);
    }
}
```

Auf den ersten Blick sehen diese Linien kompliziert aus. Dabei wird nur die im Kapitel 7.2.3 beschriebene Formel ausgerechnet. Zweimal wird mit einer Verzweigung etwas überprüft.

```
if (p2sqr < 0)
    return;
```

Im ersten Fall wird ausgerechnet, ob die Determinante, also der Teil der in der Wurzel steht, negativ ist. Wenn nämlich dieser Teil negativ ist, gibt es keine Lösung für diese Formel. Infolgedessen gibt es auch keinen Schnittpunkt. Darum wird in diesem Fall diese Funktion mit einem «return» verlassen und das Programm geht weiter.

```
if (t > 0 && t < bestHit.distance)
{
    bestHit.distance = t;
    bestHit.position = ray.origin + t * ray.direction;
    bestHit.normal = normalize(bestHit.position - sphere.xyz);
}
```

Im zweiten Fall wird berechnet, ob der Schnittpunkt näher ist als alle Schnittpunkte zuvor. Es funktioniert genau gleich wie bei den Ebenen in Kapitel 7.3.3 Wenn es der nächste Schnittpunkt ist, werden die Schnittpunktdaten von «bestHit» geändert. Das sieht am Schluss so aus:



Abbildung 7.3: Eine mit physikalisch basiertem Rendern komplett erzeugte Szene.

8 Bewegungssystem

8.1 Ziel

Mein Augpunkt ist fest. Folglich wird die Szene immer von der gleichen Perspektive gerendert. Wenn ich das ändern will, wird es mühsam die Kamera ständig manuell zu verschieben. Deswegen möchte ich ein einfaches Bewegungssystem programmieren, mit dem der Nutzer sich in Echtzeit in der Szene bewegen kann. Mit den Tasten w, a, s, d, Space und Shift kann sich der Nutzer in alle sechs Richtungen bewegen. Zudem soll der Nutzer die Kamera mit der Maus verändern können.

8.2 Theorie

8.2.1 Wie funktioniert es?

In diesem Teil werden zwei Aufgaben bewältigt. Einerseits muss sich die Kamera drehen, wenn die Maus sich nach oben, unten, rechts oder links bewegt. Andererseits soll sich die Kamera in alle sechs Richtungen bewegen können, wenn w, a, s, d, Space oder Shift gedrückt wird.

8.2.2 Bewegen

Hier gibt es eine nützliche Funktion von Unity. Wenn w gedrückt wird gibt sie 1 aus und wenn s gedrückt wird -1. Das gleiche auch für a und d. Diese Funktion gibt aber nicht nur 1 und -1 aus, sondern erzeugt einen runden Übergang zwischen den zwei Stadien. Das macht die Bewegungen der Kamera weniger abrupt. Diese zwei Varianten mit w, s, a und d werden für die x- und z-Achse genutzt. Für die y-Achse gibt es leider keine solche Funktion. Deswegen wird die Abfrage der Tasten Space und Shift manuell gemacht. Diese Bewegungen sind dementsprechend auch nicht rund. Diese Funktionen geben aber nur einen Wert. Jetzt muss der Wert in eine Positionsveränderung umgewandelt werden. Dafür gibt es eine weitere Funktion. Sie wird je nachdem, welcher Computer genutzt wird, mehr oder weniger als 60 Male ausgeführt. Der Benutzer bewegt sich jede Iteration um den zuvor berechneten Wert. Damit entsteht aber ein Problem. Wenn die Funktion auf einem Computer mehr als auf einem anderen Computer ausgeführt wird, bewegt sich der Nutzer auf dem leistungsstärkeren schneller. Es gibt eine Variable, die dieses Defizit ausgleicht. Somit kann ich die Positionsveränderung mal diese Variable rechnen und die Veränderung wird dementsprechend angepasst.

8.2.3 Drehen

In Unity gibt es eine Schreibweise für Rotationen um Achsen. Also kann ich Objekte um die entsprechenden Achsen drehen lassen. Mit einer Funktion von Unity wird die Veränderung der Mausposition aufgezeichnet. Mit diesen Werten kann ich bestimmen, um wie viel sich die Kamera drehen muss. Hier wird die Anpassungsvariable nicht gebraucht, weil es keine Rolle spielt wie schnell sich die Kamera dreht. Diese Geschwindigkeit wird allein von der Geschwindigkeit der Maus bestimmt.

8.3 Implementierung

```
xAxis = Input.GetAxis("Horizontal");
zAxis = Input.GetAxis("Vertical");
Float ySpeed;
```

«xAxis» und «zAxis» sind die zwei Variablen, die die Änderung der x und z Position mit -1 und 1 beschreiben. «Input.GetAxis()» wird dazu verwendet diese Positionsänderung zu bekommen. Die Variable «ySpeed» brauche ich in den nächsten Zeilen.

```
if (Input.GetKey(KeyCode.Space))
    ySpeed = 1;
else if (Input.GetKey(KeyCode.LeftShift))
    ySpeed = -1;
```

Die Variable «ySpeed», die die Positionsänderung der y-Achse speichert, wird gesetzt. Der Befehl «Input.GetKey(KeyCode.»Taste»)» bestimmt, ob die angegebene Taste gedrückt wird oder nicht. Je nachdem was gedrückt wird, wird der Wert auf 1 oder -1 gesetzt.

```
Vector3 moveVec = new Vector3(new Vector3(xAxis * mvSpeed *
Time.deltaTime, ySpeed * mvSpeed * Time.deltaTime, zAxis * mvSpeed *
Time.deltaTime)
transform.Translate(moveVec);
```

«transform.Translate(«position»)» verändert die Position der Kamera, um den angegebenen Vektor. «mvSpeed» ist eine Variable, mit der die Geschwindigkeit der Kamera verändert werden kann. «Time.deltaTime» ist die in Kapitel 8.2.2 beschriebene Anpassungsvariable. Also wird zuerst der Verschiebungsvektor berechnet und dann wird die Position der Kamera angepasst.

```
yaw += speedH * Input.GetAxis("Mouse X");
pitch -= speedV * Input.GetAxis("Mouse Y");
```

Die Orientierung von den Variablen «yaw» und «pitch» werden angepasst. Sie stehen für die Drehung um die x- und y-Achse. Die Anpassung erfolgt durch das addieren der neu passierten Positionsänderungen der Maus. «Input.GetAxis(«Welche Mauskoordinate»)» gibt dieses Mal nicht mit Zahlen an, ob die jeweilige Taste gedrückt wurde, sondern um wie viel die Maus sich bewegt hat. Einmal die x-Koordinate und einmal die y-Koordinate der Änderung.

```
Vector3 turnVec = new Vector3(pitch, yaw, 0)
transform.eulerAngles = turnVec
```

Zum Schluss wird die Rotation der Kamera mit «transform.eulerAngles» aktualisiert, indem der neue Wert «turnVec» reingeschrieben wird.

9 Ausblick

Mit dieser Maturaarbeit habe ich mein Ziel noch nicht ganz erreicht. Momentan kann das Programm nur einfache Objekte ohne Schatten und Reflektionen abbilden. Ich hätte gerne noch einen harten Schatten eingeführt, damit die Objekte realistischer ausgesehen hätten. Dazu hätte ich noch eine Lichtquelle einfügen müssen. Das wiederum hätte die Strahlenberechnung an den Schnittpunkten schwieriger gemacht. Zudem hätte ich gerne noch Reflektionen in meinem Programm gehabt, was noch anspruchsvoller geworden wäre. Eine sehr interessante Erweiterung sind Nurbs. Nurbs sind runde Oberflächen, die aus Punkten aufgebaut werden. Die Möglichkeiten hätten sich damit potenziert und mich in die Lage versetzt komplizierte Objekte abzubilden. Ich werde mich weiterhin mit diesen Optionen beschäftigen und hoffe diese in Zukunft verbessern zu können.

Selbständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Maturaarbeit selbständig und ohne unerlaubte fremde Hilfe erstellt habe und dass alle Quellen, Hilfsmittel und Internetseiten wahrheitsgetreu verwendet wurden und belegt sind.

10 Definitionsverzeichnis

1. *Rendern*: Der Prozess, indem eine 3D-Szene in ein 2D-Bild verwandelt wird.
2. *Variabel*: Ein Speicherbereich in der ein bestimmter Wert eines Datentypes gespeichert werden kann. Dafür Beispiele sind: ganze Zahlen, Kommazahlen, Zeichenketten, Buchstaben oder Wahrheitswerte.
3. *Augpunkt*: Der Punkt, von dem aus die Szene gerendert wird.
4. *Verzweigung*: Eine Verzweigung legt fest, welcher folgende Codeteil ausgeführt wird. Der Codeteil der ausgeführt werden soll, falls die Voraussetzung wahr ist, folgt dem Codewort mit geschweiften Klammern. Das Codewort ist ein «if».
5. *Struktur*: Ein Objekt oder Speicherort, in denen man mehrere Werte speichern kann. Man kann die Struktur selber definieren und sagen welche Variablen mit welchem Namen man in seiner Struktur will.
6. *Programmierschnittstelle*: Die Anbindung von Programmen an das System, um Grafik, Audio oder auch Eingaben kontrollieren oder abrufen zu können. Sie werden über die Programmierschnittstellen geregelt.
7. *Echtzeit – Rendering*: Meistens werden Bilder über Stunden hinweg berechnet, um ein realgetreues Resultat zu bekommen. Wenn etwas so schnell gerendert wird, dass man mit dem Auge die Verzögerung gar nicht mehr sehen kann, nennt man es Echtzeit-Rendering.
8. *Shader*: Ein Shader ist eine kleine Recheneinheit, die unter anderem für die Erzeugung von 3D-Effekten benutzt werden kann.
9. *Funktion*: Ein ausführbarer Programmbaustein, der Sachen im Programm ändert oder/und einen Wert zurückgibt.
10. *Klasse*: Ein grösserer Programmbaustein, welcher wie ein Objekt von anderen getrennt ist und eigene Variablen und Funktionen hat.
11. *Unity*: Die Entwicklungsumgebung, in der der Autor arbeitet.
12. *HöhereProgrammiersprache*: Höheren Sprachen lassen einem das Programm einfacher lesen, weil die Syntax auf Menschen und nicht auf Computer (wie der Binärkode) abgestimmt ist. Dabei kann ein höherer Befehl sehr viele kleine Befehle für den Computer bedeuten, ohne dass wir diese ausschreiben müssen.

Abbildungsverzeichnis

1.1	Eine gerenderte Szene, Quelle: https://www.google.com/url?sa=i&source=images&cd=&cad=rja&uact=8&ved=2ahUKEwisxPSKueD1AhXBy6QKHQTQAooQjRx6BAGBEAQ&url	4
4.1	Das ganze System in Übersicht, Quelle: https://fr.wikipedia.org/wiki/Ray_tracing#/media/Fichier:Ray_trace_diagram.svg	9
4.2	Kartesisches Koordinatensystem, Quelle: https://de.wikipedia.org/wiki/Kartesisches_Koordinatensystem/media/Datei:Kartesisches_system.svg	10
4.3	Polarkoordinaten, Quelle: https://de.wikipedia.org/wiki/Kugelkoordinaten/media/Datei:Kugelkoord-def.svg	11
5.1	Lochkamera, Quelle: https://www.howtogeek.com/63409/htg-explains-cameras-lenses-and-how-photography-works/	12
5.2	Strahlensimulation, Quelle: https://www.google.com/url?sa=i&source=images&cd=&cad=rja&uact=8&ved=2ahUKEwjBs-LIktXlAhWPY1AKHVBHDhOQjRx6BAGBEAQ&url	13
6.1	Ein Beispiel des Hintergrundes aus dem Programm, Autor: Manuel Baumann	16
7.1	Schnittpunkt, Quelle: https://upload.wikimedia.org/wikipedia/commons/thumb/f/f1/Raytracing.svg/440px-Raytracing.svg.png	17
7.2	Zwei Ebenen aus dem Programm, Autor: Manuel Baumann	20
7.3	Ein Gesamtausschnitt aus dem Programm, Autor: Manuel Baumann	22

Literatur

- [1] Danny Kodicek John P. Flynt. *Mathematics and Physics for Programmers*. 2. Aufl. 20 Channel Center Street, Boston, MA 02210, USA: Course Technology, 2012.
- [2] Matt Pharr. *Physically Based Rendering*. 2018. URL: <http://blog.three-eyed-games.com/2018/05/03/gpu-ray-tracing-in-unity-part-1/#comments>.
- [3] Wikipedia. *C-Sharp*. 2019. URL: <https://de.wikipedia.org/wiki/C-Sharp>.
- [4] Wikipedia. *High-Level Shading Language*. 2019. URL: https://en.wikipedia.org/wiki/High-Level_Shading_Language.
- [5] Wikipedia. *Homogene Koordinaten*. 2019. URL: https://de.wikipedia.org/wiki/Homogene_Koordinaten.
- [6] Wikipedia. *Kartesisches Koordinatensystem*. 2019. URL: https://de.wikipedia.org/wiki/Kartesisches_Koordinatensystem.
- [7] Wikipedia. *Polarkoordinaten*. 2019. URL: <https://de.wikipedia.org/wiki/Polarkoordinaten>.