Facharbeit im Mathematik-Leistungskurs

Verwendung des Raytracing-Algorithmus in der Computergrafik zur Bildsynthese aus geometrischen Szenenbeschreibungen

David Albers

Haltern am See, den 28. Februar 2020

betreut am Joseph-König-Gymnasium durch Miriam Krieger

Inhaltsverzeichnis

Abbildungsverzeichnis

Listingverzeichnis

1 Einführung und Gang der Analyse

Jede Grafiksoftware am Computer – sei es ein Programm zur architektonischen Gestaltung, ein CAD[1]-System oder ein Videospiel – muss ein zentrales Problem lösen: Die Darstellung einer dreidimensionalen Umgebung auf einem zweidimensionalen Bildschirm. Diese Umgebung kann niemals exakt und vollumfänglich dargestellt werden; es wird auf modellhafte Annäherungen und diverse „Tricks“ zurückgegriffen. Dieser Prozess heißt *Bildsynthese*. Ein mögliches Verfahren zur Bildsynthese ist der *Raytracing*-Algorithmus (engl. für *Strahlenverfolgung*), der auf der Simulation von durch eine Szene fallenden Lichtstrahlen basiert. Dieser wird in der modernen Computergrafik mittlerweile sehr vielseitig eingesetzt. Nvidia hat ihn mit der RTX-Technologie für den Privatanwender massentauglich gemacht. Mit steigender Komplexität der Verfahren können dadurch teilweise fotorealistische Bildsynthesen durchgeführt werden.

Diese Facharbeit veranschaulicht das zugrundeliegende Prinzip einer Bildsynthese durch Raytracing. Sie beinhaltet nicht die Durchführung einer fotorealistischen Bildsynthese.

Einführend werden zunächst das Konzept der Bildsynthese und ihre Problematik dargestellt. Anschließend werden der Raytracing-Algorithmus und seine mathematischen Hintergründe sowie einige seiner Einsatzgebiete in der Computergrafik erläutert. Seine Funktionsweise wird anschließend anhand des *3D Raytracing Demo Project*[2], welches im Rahmen dieser Facharbeit entwickelt wurde, praktisch veranschaulicht.

[1] Eine Abkürzung für das Englische *computer-aided drafting*; bezeichnet Software zur computerunterstützten Erstellung von technischen Zeichnungen oder Dokumentationen.

[2] Zu finden unter `https://gitlab.com/davidjalbers/raytracing-demo/`.

2 Die Bildsynthese als zentrale Aufgabe der Computergrafik

2.1 Notwendigkeit der Bildsynthese

Die Nutzung von Computergrafik und ihren Verfahren hatte schon immer das Ziel, eine darzustellende dreidimensionale Umgebung glaubhaft zu präsentieren. Dieses Ziel geht nicht zwingend mit realistisch anmutender Darstellung einher, obwohl „aktuelle Echtzeit-Computer-Grafik-Systeme [...] durch die stetig wachsende Leistung von Prozessoren und Grafikkarten annähernd fotorealistische Ergebnisse [erreichen]“ (Schmidt 2006, S.13).

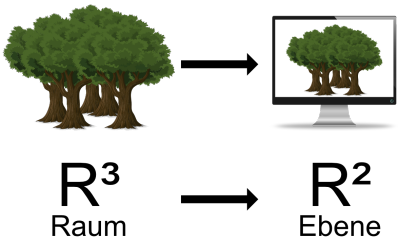

Abb. 2.1: Aufgabe der Bildsynthese (eigene Darstellung)

Es ist dennoch hinderlich, dass die Ausgabe der Software (meistens) über eine zweidimensionale Schnittstelle, wie etwa einen Monitor oder ein Blatt Papier, an den Benutzer gesendet wird, obwohl der Mensch eigentlich zur Wahrnehmung aller drei räumlicher Dimensionen der virtuellen Welt fähig wäre. In der Computergrafik ist es folglich nötig, die Anzahl der Dimensionen einer Welt von drei auf zwei zu reduzieren, um sie dem Benutzer in Form eines Bildes präsentieren zu können. Man könnte auch von der Notwendigkeit sprechen, den Raum auf eine Ebene zu reduzieren (vgl. Abbildung 2.1). Dabei soll jedoch die Illusion einer dritten Dimension erhalten bleiben. Ebendiese Erzeugung eines zweidimensionalen Bildes aus einer dreidimensionalen Welt wird als Bildsynthese (engl. *image synthesis*) bezeichnet (vgl. Glassner 1989, S. 1).

2.2 Grenzen der Bildsynthese

Der Zweck der Bildsynthese ist in Abbildung 2.1 veranschaulicht. Zur Verdeutlichung ihrer Funktionsweise lässt sich ein Vergleich zur Fotografie anstellen (vgl. Glassner 1989, S. 1ff.): Auch hier wird (mithilfe einer Kamera) die dreidimensionale Realität auf einem zweidimensionalen Digitalbild gespeichert. Die Fotografie nutzt jedoch das echte physikalische Phänomen des Lichts, welches am Computer höchstens durch mathematische Berechnungen simuliert werden kann. Außerdem stellt eine fotorealistische und vollumfängliche Simulation des Lichts, gar bis auf Teilchenebene, ungeheure Leistungsanforderungen an ein Grafiksystem.

3 Der Raytracing-Algorithmus als Mittel zur Bildsynthese

3.1 Geschichte und Definition des Algorithmus

Das hinter Raytracing stehende Grundprinzip ist bereits deutlich älter als die Computergrafik selbst. Albrecht Dürer präsentierte im Jahr 1538 im Buch „Underweysung der Messung“ (sic!) ein Verfahren, mit dessen Hilfe ein Bild, gezeichnet durch den Blick des Künstlers durch ein vor dem Motiv aufgebautes Raster, perspektivisch korrekter wurde (vgl. Abbildung 3.1).

Ansätze des Raytracing-Algorithmus und seiner Verwendung zur Bildsynthese wurden jedoch erst 1968 (vgl. Appel 1968) und 1971 (vgl. Goldstein und Nagel 1971) beschrieben: Das physische Raster bei Albrecht Dürer ist hier nun das aus Pixeln bestehende Digitalbild; und der Blick des Künstlers sind Lichtstrahlen, von denen vom Augpunkt des Künstlers aus jeweils einer pro Pixel durch die Szene fällt. Letzterer Punkt macht deutlich, dass der Raytracing-Algorithmus auf der *geometrischen Optik* basiert, den Lichteinfall also zu geraden Strahlen vereinfacht.

Diese Strahlen (engl. *ray* für „Strahl“) werden nun verfolgt (engl. *to trace* für „verfolgen“) und auf Schnittpunkte mit Objekten in der Szene getestet (nach diesem essentiellen Prozess wurde der Raytracing-Algorithmus benannt). Schneidet ein von einem bestimmten Pixel ausgehender Strahl ein Objekt der Szene, so wird dieses Pixel in der Farbe des geschnittenen Objekts eingefärbt. Bei ausreichend hoher Strahlenanzahl (d. h. ausreichender Auflösung des Pixelrasters) wird so die Kontur des Objekts auf dem synthetisierten Bild erkennbar.

Abb. 3.1: Albrecht Dürers „Erfindung“ des Raytracing (Quelle: `https://commons.wikimedia.org/wiki/File:Albrecht_durer_ray_tracing.png`)

3.2 Schritte des Raytracings im Detail

3.2.1 Strahlenerzeugung und Projektion

Um Raytracing betreiben zu können, müssen die Strahlen zunächst vom Blickpunkt des Betrachters (auch *Kamera* genannt) aus erzeugt werden. Ein Strahl im dreidimensionalen Raum ähnelt dabei einer Halbgeraden im zweidimensionalen Raum: Er wird von einem Ursprung einseitig begrenzt, aber verläuft von dort aus entlang einer definierten Richtung ins Unendliche.

Auf einem Strahl mit dem Ursprung $O = (x_O|y_O|z_O)$ und der Richtung $\vec{R} = (x_R|y_R|z_R)$ gilt für alle Punkte $P = (x_P|y_P|z_P)$ mit dem Abstand $t \geq 0$ vom Ursprung:

$$P = O + t * \vec{R} \tag{3.1}$$

$$\Leftrightarrow \begin{pmatrix} x_P \\ y_P \\ z_P \end{pmatrix} = \begin{pmatrix} x_O + t * x_R \\ y_O + t * y_R \\ z_O + t * z_R \end{pmatrix} \tag{3.2}$$

(vgl. Glassner 1989, S. 35)

Die Art und Weise, wie die beiden Parameter O und $\vec{R}$ eines Strahls definiert werden, bestimmt maßgeblich das Aussehen des späteren Bildes. Man spricht von der *Projektion* eines Bildes. Es werden zwei Projektionstypen unterschieden.

Perspektivische Projektion. Bei dieser Projektion entspringen alle Strahlen einem zentralen Punkt und verlaufen durch unterschiedliche Stellen der Bildebene. Als Folge dessen erscheinen für den Betrachter von ihm weiter entfernte Objekte kleiner. Außerdem werden die Objekte verzerrt, je näher sie dem Bildrand kommen. Wird perspektivisch projiziert, so ist der Ursprung O über alle Strahlen des Bildes konstant und vordefiniert. Die Richtung $\vec{R}$ variiert nach jeweiligem Pixel und leitet sich vom zentralsten Pixel ab.

Orthografische Projektion. Hier verlaufen alle Strahlen parallel zueinander. Daher erscheinen für den Betrachter alle Objekte, ungeachtet ihrer Entfernung, gleich groß. Es liegt keine Verzerrung der Objekte vor. Wird orthografisch projiziert, so ist die Richtung $\vec{R}$ über alle Strahlen des Bildes konstant und vordefiniert. Der Ursprung O variiert nach jeweiligem Pixel und leitet sich vom zentralsten Pixel ab.

Die beiden Projektionen sind in Abbildung 3.2 gegenübergestellt.

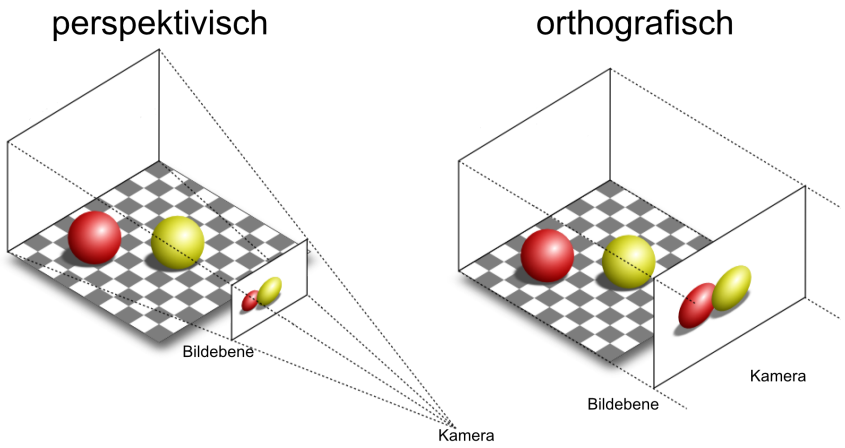

Abb. 3.2: Gegenüberstellung des perspektivischen und orthografischen Projektionstyps (angelehnt an: `https://stackoverflow.com/questions/36573283/from-perspective-picture-to-orthographic-picture`)

Im Folgenden ist es nicht weiter relevant, wie genau für einen Strahl während der Projektion der pro Pixel variierende Parameter bestimmt wird (die Richtung $\vec{R}$ bei perspektivischer und der Ursprung O bei orthografischer Projektion). Das „3D Raytracing Demo Project“ nutzt jedoch die orthografische Projektion. Sie wird in Abschnitt 4.2.1 näher erläutert.

3.2.2 Schnittpunkttest

Nachdem ein Strahl erzeugt wurde, wird während des *Schnittpunkttests* für jedes Objekt in der Szene überprüft, ob sich der Strahl mit diesem Objekt schneidet. Hier wird die Schnittpunktberechnung zwischen einem Strahl und einer Kugel erläutert. Analog zur Definition eines Strahls in Gleichung (3.2) muss dazu zunächst auch eine Kugel definiert werden.

Gemäß des Satzes des Pythagoras gilt:
Ein Punkt $P = (x_P|y_P|z_P)$ liegt auf der Oberfläche einer Kugel mit dem Mittelpunkt $M = (x_M|y_M|z_M)$ und dem Radius r, wenn er die folgende Gleichung erfüllt:

$$(x_P - x_M)^2 + (y_P - y_M)^2 + (z_P - z_M)^2 = r^2 \tag{3.3}$$

(vgl. Glassner 1989, S.36)

Ein Strahl schneidet nun eine Kugel, wenn ein beliebiger Punkt P auf dem Strahl auch auf der Oberfläche der Kugel liegt, also sowohl Gleichung (3.2) des Strahls als auch Glei-

chung (3.3) der Kugel erfüllt. Mit anderen Worten:

Existiert ein Wert $t \geq 0$, für den ein Punkt $P = (x_P|y_P|z_P)$ auf einem Strahl mit dem Ursprung $O = (x_O|y_O|z_O)$ und der Richtung $\vec{R} = (x_R|y_R|z_R)$ und gleichzeitig auf der Oberfläche einer Kugel mit dem Mittelpunkt $M = (x_M|y_M|z_M)$ und dem Radius r liegt, so schneidet jener Strahl diese Kugel im Punkt P.

$$((x_O + x_R * t) - x_M)^2 + ((y_O + y_R * t) - y_M)^2 + ((z_O + z_R * t) - z_M)^2 = r^2 \quad (3.4)$$

$$\Leftrightarrow at^2 + bt + c = 0 \quad (3.5)$$

Setzt man Gleichung (3.2) in Gleichung (3.3) ein, ergibt sich Gleichung (3.4). Durch Termumformungen erhält man daraus eine quadratische Gleichung (3.5) mit den Parametern (3.6), (3.7) und (3.8), die mithilfe einer quadratischen Lösungsformel (a-b-c-Formel in jedem Fall oder p-q-Formel, falls der Richtungsvektor des Strahls ein Einheitsvektor ist) nach t aufgelöst werden kann.

$$a = x_R^2 + y_R^2 + z_R^2 \quad (3.6)$$

$$b = 2 * (x_R * (x_O - x_M) + y_R * (y_O - y_M) + z_R * (z_O - z_M)) \quad (3.7)$$

$$c = (x_O - x_M)^2 + (y_O - y_M)^2 + (z_O - z_M)^2 - r^2 \quad (3.8)$$

(vgl. Glassner 1989, S.35ff.)

3.2.3 Tiefentest

Während der Schnittpunkttests werden pro Strahl alle Objekte der Szene getestet. Bisher erhält man durch Lösen der Gleichung (3.5) nach t lediglich die Erkenntnis, *ob* ein Schnittpunkt existiert. Geht man davon aus, dass sich keine Objekte in der Szene überlappen, ist dieses Vorgehen unproblematisch. Falls ein Objekt in der Szene jedoch ein anderes verdeckt, so muss sichergestellt werden, dass dies auch im synthetisierten Bild der Fall ist – man muss also nicht nur wissen, *ob* ein Schnittpunkt vorliegt, sondern auch, *wo* er liegt. Dazu vergleicht man pro Strahl für jedes neu getestete Objekt den aus Gleichung (3.5) erhaltenen Wert für t mit allen bisher getesten Objekten. Dasjenige Objekt, welches das kleinste t liefert und folglich dem Ursprung des Strahls und damit dem Betrachter am nächsten liegt, bestimmt die Farbe des jeweiligen Pixels.

Die Einzelschritte aus den Abschnitten 3.2.1 bis 3.2.3 sind in Listing 3.1 zusammengefasst. Sie werden zusammen als *Verdeckungsberechnung* bezeichnet. Die bei der Verdeckungsberechnung verwendeten Strahlen lassen sich als *Primärstrahlen* klassifizieren.

```
fuer jedes Pixel i im Bild:
        erzeuge einen Strahl s, der durch i verlaeuft
        fuer jedes Objekt o in der Szene:
                falls s o schneidet und o naeher am Bild ist als bisherige Objekte:
                        vermerke die Entfernung von o zum Bild
                        setze die Farbe von i auf die Farbe von o
```

Listing 3.1: Pseudocode des Raytracing-Algorithmus

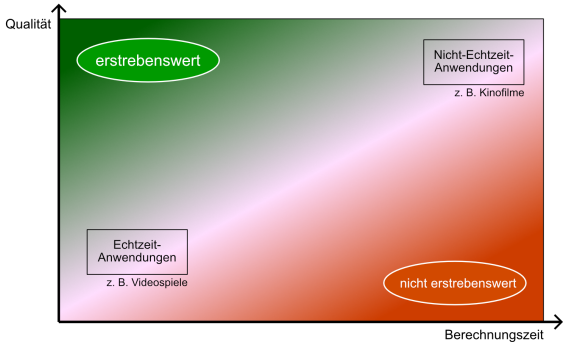

Abb. 3.3: Klassifizierung verschiedener Anwendungsszenarien (eigene Darstellung)

Mithilfe der sogenannten *Sekundärstrahlen* kann der Raytracing-Algorithmus noch weitergeführt werden, um Beleuchtungseffekte, Reflexionen und Lichtbrechungen zu erzeugen (vgl. Schmidt 2006, S. 23); die dabei durchgeführten Berechnungen sind der Verdeckungsberechnung jedoch ähnlich.

3.3 Anwendungsbeispiele für Raytracing

Die Einsatzzwecke für moderne Computergrafik und den Raytracing-Algorithmus sind sehr vielseitig und die durchgeführten Bildsynthesen dennoch ähnlich. Die Anwendungsszenarien werden in echtzeitbasiert (vgl. Abschnitt 3.3.2) und nicht echtzeitbasiert (vgl. Abschnitt 3.3.1) gegliedert, da sich diese beiden Kategorien signifikant voneinander unterscheiden. Das Verhältnis der beiden Kategorien zueinander ist in Abbildung 3.3 in Abhängigkeit von den proportionalen Parametern *Ergebnisqualität* und *Berechnungszeit* veranschaulicht.

3.3.1 Bildsynthesen in Nicht-Echtzeit

Überall dort, wo eine Grafiksoftware nicht in Echtzeit auf veränderte Nutzereingaben reagieren muss, kann viel Zeit für die Bewältigung der in Abschnitt 2.2 angesprochenen Leistungsanforderungen einer fotorealistischen Bildsynthese aufgewendet werden, etwa

zur Simulation von durch die Szene verlaufenden Lichtstrahlen. Dies ist zum Beispiel in der Filmindustrie der Fall: Um einen computergestützten Film zu produzieren, benötigen Filmstudios ganze Computerverbände, sogenannte *render farms*, die dank spezieller Grafikprozessoren weit höhere Rechenleistung aufweisen als etwa heutige Heimcomputer. Dabei kommen (teilweise sogar ausschließlich) auf Raytracing basierende Algorithmen zum Einsatz, die einen recht natürlichen Weg des Lichts simulieren.

Der Sammelbegriff *CGI*[1] bezeichnet den Einsatz von Computergrafik in Nicht-Echtzeit zur Produktion von Filmen. CGI-Technologie hatte weitreichende Auswirkungen auf die Filmindustrie, wie ein Artikel der Huffington Post zeigt (vgl. HuffPost Australia 2020).

3.3.2 Bildsynthesen in Echtzeit

Überall dort, wo eine Grafiksoftware in Echtzeit auf veränderte Nutzereingaben reagieren muss, darf nicht viel Zeit für die Bewältigung der Leistungsanforderungen einer fotorealistischen Bildsynthese aufgewendet werden. Das folgende Beispiel verdeutlicht dies: Folgt man der Annahme, dass etwa ein Videospiel nichts anderes ist als ein Film, der live auf Eingaben des Benutzers reagieren muss, so ergibt sich, dass das Videospiel die Berechnungen, die für den herkömmlichen Film im Vorhinein durchgeführt worden sind, in Echtzeit durchführen muss. Der typische Kinofilm läuft in der Regel mit einer *Bildwiederholfrequenz* von 24 Bildern pro Sekunde (oder *fps*, für das Englische *frames per second*) ab. Das Videospiel muss also mindestens 24 Bildsynthesen in der Sekunde durchführen und kann dadurch zwangsläufig nicht die Grafikqualität eines Films erreichen, für den die Bildsynthesen eine praktisch beliebige Zeit in Anspruch nehmen dürfen. In der Echtzeit-Grafiksoftwareentwicklung spielt Raytracing daher (noch) eine untergeordnete Rolle.

Ein sehr prominentes Verfahren für Echtzeit-Anwendungen ist die *Rasterisierung*. Sie modelliert geometrische Objekte als sogenannte *Primitive*, vorzugsweise als Dreiecke, denn ein Dreieck ist das höchste geometrische Objekt, das im dreidimensionalen Raum noch eine Ebene ergibt. Die Eckpunkte (auch *vertices* genannt) eines jeden Primitivs werden mithilfe von sogenannten *Transformations- und Projektionsmatrizen* so verschoben, dass seine Koordinaten in der Szene zu Koordinaten im Ausgabebild werden. Zwischen den verschobenen Vertices wird dann das Primitv aufgebaut. Die Rasterisierung hat gegenüber dem Raytracing allerdings einen entscheidenden Nachteil: Höhere Modelle, seien es nur Kugeln, müssen in ebene Primitive heruntergebrochen werden. Eine mit Rasterisierung gerenderte Kugel ist daher nie perfekt rund.

[1] Ein Akronym für das Englische *computer-generated imagery*.

4 Raytracing in der Programmierpraxis: Das „3D Raytracing Demo Project“

Die in Kapitel 3 hergeleitete Mathematik wird nun praktisch angewendet. Dazu wurde die Java-Anwendung[1] „3D Raytracing Demo Project“ entwickelt, die eine Bildsynthese unter Verwendung des Raytracing-Algorithmus durchführen kann. Sie soll im Folgenden vorgestellt werden.

Raytracing kann sowohl in der Verdeckungsberechnung (Primärstrahlen) als auch in der Schattierungs- und Reflexionsberechnung (Sekundärstrahlen) eingesetzt werden. Im „3D Raytracing Demo Project“ wurde aus Gründen des Umfangs dieser Arbeit ausschließlich die Verdeckungsberechnung implementiert.

4.1 Grundlagen für die Ein- und Ausgabe

Um die Bildsynthese in einem Programm zu implementieren, müssen zunächst die abzubildende Szene und das Ausgabebild (bzw. Raum und Ebene) in eine für den Computer adäquate Form gebracht werden.

Für die Eingabe wird angenommen, dass eine Szene eine Hintergrundfarbe, eine Kamera und statische geometrische Figuren enthält. Die implementierte Kamera projiziert die Szene orthografisch. Das synthetisierte Ausgabebild wird im PNG[2]-Format gespeichert. Es enthält ein Raster vieler Bildpunkte unterschiedlicher Farbe, genannt *Pixel*. Eine einzelne Farbe kann dabei zum Beispiel mit jeweils 24 Bit im RGB-Farbraum[3] codiert werden, wodurch sich etwa 16 Millionen verschiedene Farben ergeben, die ein Pixel annehmen kann.

[1] Java ist eine universelle objektorientierte Programmiersprache (vgl. Oracle Corporation 2020).

[2] PNG (Abkürzung für *Portable Network Graphics*) ist ein „ein Rastergrafikformat mit verlustfreier Datenkompression“ und „das meistverwendete verlustfreie Grafikformat im Internet“ (Wikipedianer 2020a).

[3] Ein additiver Farbraum, der auf der Mischung der drei Grundfarben Rot, Grün und Blau basiert (vgl. Wikipedianer 2020b).

4.2 Modellierungen und Klassen

Im Folgenden werden grundlegende Kenntnisse des *objektorientierten Programmierparadigmas* vorausgesetzt[4].

4.2.1 Vektoren und Strahlen

Die Mathematik unterscheidet im Raum zwischen Punkten und Vektoren:

$$P(0|8|3) \neq \vec{V} \begin{pmatrix} 0 \\ 8 \\ 3 \end{pmatrix}$$

Für die Modellierung von Vektoren und Punkten wurde im „3D Raytracing Demo Project“ jedoch ein und dieselbe Klasse `Vector3f` verwendet (das `f` steht für `float`, den gewöhnlichen Gleitkommazahl-Datentyp in Java; die `3` für die drei Koordinaten eines Punktes bzw. Vektors). Ein instanziiertes Objekt dieser Klasse mit den Koordinaten x, y und z kann entweder den Punkt $P = (x|y|z)$ oder den Vektor $\overrightarrow{OP}$ vom Koordinatenursprung $(0|0|0)$ zum Punkt P repräsentieren.

Ein Strahl mit einem Ursprungspunkt O und einem Richtungsvektor $\vec{R}$ besteht daher aus zwei Instanzen von `Vector3f`, die in einem Objekt der Klasse `Ray` gehalten werden.

4.2.2 Szenenobjekte

Jedes geometrische Objekt einer Szene hält ebenfalls einen `Vector3f`, der seine Position speichert. Für eine Kugel der Klasse `Sphere` ist diese Position etwa der Mittelpunkt. Dazu kommen objektspezifische Daten, wie etwa einen weiteren `float` für den Radius einer Kugel.

Zusätzlich implementiert jedes geometrische Objekt die Schnittstelle `IGeometry`, sodass die dort deklarierte Methode `intersect(Ray)` überschrieben werden muss. Auf diese Weise enthält jedes geometrische Objekt die Berechnungslogik für Schnittpunkte mit einem Strahl in sich selbst. Der Rückgabetyp der Methode `intersect(Ray)` ist `Intersection`, ein Container mit Informationen über den Schnittpunkt (im Wesentlichen

[4] Eine Einführung in die Thematik der objektorientierten Programmierung findet sich beispielsweise unter: `https://www.itwissen.info/Objektorientierte-Programmierung-object-oriented-programming-OOP.html` (letzter Zugriff: 13. Februar 2020).

```
1  Ray generateRay(int x, int y, int w, int h) {
2          x = x - (w / 2);
3          y = y - (h / 2);
4
5          Vector3f widthVector = dir
6                  .cross(new Vector3f(0, 1, 0))
7                  .normalize();
8          Vector3f heightVector = dir
9                  .cross(widthVector)
10                 .normalize();
11
12         return new Ray(pos
13                 .add(widthVector.multiply(x).divide(zoom)
14                 .add(heightVector.multiply(y).divide(zoom))), dir);
15 }
```

Listing 4.1: Implementierung orthografischer Projektion

wieder ein `Vector3f` für die Koordinaten des Schnittpunktes). Die Implementierung von `intersect(Ray)` in der Klasse `Sphere` löst die Gleichung (3.5) nach t auf, um so einen eventuellen Schnittpunkt zu bestimmen.

4.3 Ablauf und Programmfluss

Die in Abschnitt 4.1 beschriebene Ein- und Ausgabe übernimmt eine Hilfsklasse `SceneIO` (das `I / O` steht für *input / output*). Der restliche Programmfluss orientiert sich an den in den Abschnitten 3.2.1 bis 3.2.3 beschriebenen Teilschritten. Die Erzeugung der Strahlen entscheidet gleichzeitig über die Projektion des Bildes und ist daher Aufgabe der Kamera. Eine Implementierung der Methode `generateRay(int x, int y, int w, int h)` aus der Schnittstelle `ICamera` ist in Listing 4.1 zu sehen. Im Listing werden die Methoden `cross(Vector3f)` und `normalize()` verwendet. Erstere bildet das Kreuzprodukt[5] eines Vektors mit einem anderen und letztere macht aus einem beliebigen Vektor einen Einheitsvektor. Der Schnittpunkttest erfolgt wie in Abschnitt 4.2.2 erläutert innerhalb der Szenenobjekte selbst, da die anzustellenden Berechnungen je nach Objekttyp variieren können.

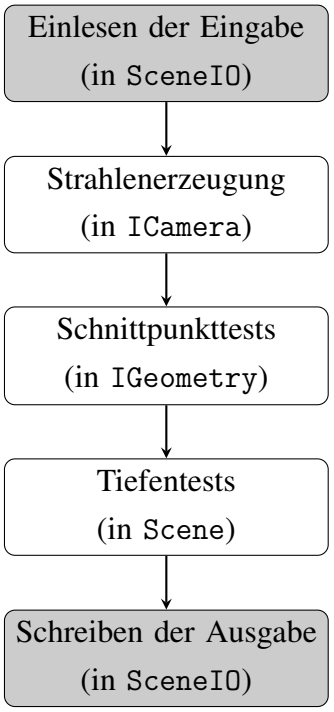

Abb. 4.1: Der Programmfluss (vereinfacht, eigene Darstellung)

Hingegen kann der Tiefentest nicht innerhalb eines Objekts ausgeführt werden, da hier

[5] Das Kreuzprodukt ist „die Verknüpfung zweier Vektoren, dessen Ergebnis wieder ein Vektor ist, der senkrecht auf den beiden Vektoren steht“ (Serlo Education 2020).

mehrere Objekte und ihre Schnittpunkte mit den Strahlen miteinander verglichen werden müssen. Diese Aufgabe übernimmt die Klasse `Scene`. Sie führt nicht nur die drei Einzelschritte Strahlenerzeugung, Schnittpunkttest und Tiefentest zusammen, sondern hält auch Referenzen auf alle in der Szene enthaltenen Objekte. Das gibt einer Instanz der Klasse `Scene` die Möglichkeit zur Methode `render(int w, int h)`. Diese ist in Listing 4.2 zu sehen. In `render(int w, int h)` wird für jedes Pixel ein Strahl erzeugt und in der Methode `trace(Ray)` verfolgt (vgl. Listing 4.3).

Eine mögliche Eingabe für das „3D Raytracing Demo Project“ und die synthetisierte Ausgabe sind in Listing A.1 und Abbildung A.1 dargestellt.

```
for (int x = 0; x < width; ++x) {
        for (int y = 0; y < height; ++y) {
                Ray ray = camera.generateRay(x, y, width, height);
                img.setRGB(x, y, trace(ray).getRGB());
        }
}
```

Listing 4.2: Ausschnitt aus der Render-Methode

```
for (IGeometry object : objects) {
        Intersection intersection = object.intersect(ray);
        if (intersection.happened) {
                float currentDistance = intersection.contactCoord
                        .subtract(ray.pos)
                        .magnitude();
                if (currentDistance < distance) {
                        distance = currentDistance;
                        hit = object;
                }
        }
}
```

Listing 4.3: Ausschnitt aus der Trace-Methode

5 Zusammenfassung

In dieser Facharbeit wurden der Raytracing-Algorithmus, seine Realanwendungen und seine Funktionsweise untersucht. Letztere wurde anhand des „3D Raytracing Demo Project“[1] praktisch veranschaulicht.

Der Algorithmus wurde erstmals definiert in den 1960er-Jahren, obwohl das Grundprinzip dahinter wesentlich älter ist und bis auf Albrecht Dürer zurückgeht. Raytracing basiert auf der Annahme, dass Licht als Strahlen durch eine Szene fällt. Die Strahlen, die dabei von den Objekten der Szene auf das Auge des Betrachters treffen würden, werden von dort aus durch die Szene zurückverfolgt.

Mit entsprechend vielen Strahlen können dadurch komplexe dreidimensionale Objekte als zweidimensionale digitale Bilder aus Pixeln gerendert werden. Die Verdeckungsberechnung ist dabei der erste Schritt: Es werden Primärstrahlen ausgesendet, die bestimmen, welches Objekt in einem Pixel sichtbar ist. Diese Verdeckungsberechnung mit Primärstrahlen wurde im „3D Raytracing Demo Project“ implementiert, welches eine geometrische Szenenbeschreibung einliest und ein Bild mit den Konturen der spezifizierten Objekte ausgibt.

Sekundärstrahlen erlauben echten raytracingbasierten Grafiksystemen anschließend die Simulation von Beleuchtung, Reflexionen und Brechungen von Licht in der Szene. Diese sind für die hohen Leistungsanforderungen eines solchen Systems verantwortlich. Gleichzeitig erzielen raytracingbasierte Grafiksysteme annähernd fotorealistische Ergebnisse und sind daher in der Film-, Software- und Videospielindustrie sehr gefragt. Mit steigender Leistung von Grafikprozessoren wird Raytracing immer vielversprechender – und in der Zukunft sicherlich in vielen Bereichen der Computergrafik zum Einsatz kommen.

[1] Der Quellcode befindet sich unter `https://gitlab.com/davidjalbers/raytracing-demo/`. Das Programm steht dort in der Version 2.0 zur freien Benutzung zur Verfügung. Für das Format der Eingabedatei liefert Listing A.1 eine Orientierung.

Literaturverzeichnis

Appel, Arthur (1968). „Some techniques for shading machine renderings of solids“. In: *AFIPS Spring Joint Computer Conference*.

Arvo, James (1986). „Backward Ray Tracing“. In: *ACM SIGGRAPH '86 Course Notes: Developments in Ray Tracing*. Apollo Computer, Inc., S. 259–263.

Glassner, Andrew S. (1989). *An Introduction to Ray Tracing*. San Diego, Kalifornien: ACADEMIC PRESS.

Goldstein, Robert A. und Roger Nagel (Jan. 1971). „3-D Visual simulation“. In: *SIMULATION (16-1)*.

HuffPost Australia (16. Feb. 2020). *How CGI Changed Movies Forever*. URL: https://www.huffingtonpost.com.au/2016/05/12/how-cgi-changed-movies-forever_n_9155494.html.

Li, Victor (15. Feb. 2020). *Ray Sphere Intersection*. URL: http://viclw17.github.io/2018/07/16/raytracing-ray-sphere-intersection/.

Oracle Corporation (27. Jan. 2020). *Java Software*. URL: https://www.oracle.com/java/.

Schmidt, Björn (Nov. 2006). „Raytracing und Szenengraphen“. Diplomarbeit. Frankfurt am Main: Johann-Wolfgang-Goethe-Universität.

Serlo Education (24. Feb. 2020). *Vektor- oder Kreuzprodukt*. URL: https://de.serlo.org/mathe/geometrie/analytischegeometrie/methoden-vektorrechnung/vektorprodukt/vektor-kreuzprodukt.

Suffern, Kevin (2007). *Ray Tracing from the ground up*. Wellesley, Massachusetts: A K Peters Ltd. ISBN: 978-1-568-81272-4.

Wegscheider, Andreas (2010). „Computergrafik mittels Raytracing“. Schulische Facharbeit. Traunreut: Johannes-Heidenhain-Gymnasium.

Wikipedianer (25. Feb. 2020a). *Portable Network Graphics (PNG)*. URL: https://de.wikipedia.org/wiki/Portable_Network_Graphics.

Wikipedianer (16. Feb. 2020b). *RGB-Farbraum*. URL: https://de.wikipedia.org/wiki/RGB-Farbraum.

A Beispiel für das „3D Raytracing Demo Project“

```
// Die Hintergrundfarbe der Szene: (R|G|B)
ambient:(127|255|255)

// Die Kameraperspektive: (Position):(Blickrichtung):(Zoomfaktor)
orthographic_camera:(0|0|20):(0|0|-1):(50)

// Eine geometrische Figur (hier Kugel): (Position):(Radius):(Farbe)
sphere:(3|8|-1):(2):(0|127|255)

// Eine weitere geometrische Figur: (Position):(Radius):(Farbe)
sphere:(-10|1|5):(5):(255|0|0)
```

Listing A.1: Mögliche Eingabedatei „input_rtd.txt“

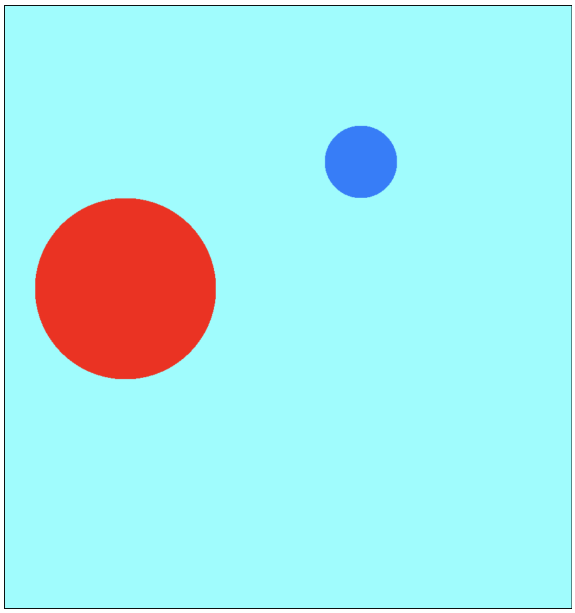

Abb. A.1: Ausgabedatei „output_rtd.png“ zur Eingabedatei „input_rtd.txt“

B Weiterführende Listings

B.1 Implementierung von Vektorrechenoperationen

Alle Rechenoperationen mit Vektoren sind in der Klasse `Vector3f` als statische Methoden definiert. Selbige Klasse verfügt auch über nicht-statische Versionen jeder Rechenoperation.

```
1  /**
2   * Addiert zwei Vektoren.
3   * @param a erster Summand
4   * @param b zweiter Summand
5   * @return Summe als Vektor
6   */
7  public static Vector3f add(Vector3f a, Vector3f b) {
8      return new Vector3f(
9          a.x + b.x,
10         a.y + b.y,
11         a.z + b.z
12     );
13 }
14
15 /**
16  * Subtrahiert zwei Vektoren.
17  * @param a der Vektor, von dem subtrahiert werden soll (Minuend)
18  * @param b der Vektor, der subtrahiert werden soll (Subtrahend)
19  * @return Differenz als Vektor
20  */
21 public static Vector3f subtract(Vector3f a, Vector3f b) {
22     return new Vector3f(
23         a.x - b.x,
24         a.y - b.y,
25         a.z - b.z
26     );
27 }
28
29 /**
30  * Multipliziert einen Vektor mit einem Skalar.
31  * @param a zu multiplizierender Vektor
32  * @param b zu multiplizierendes Skalar
33  * @return das Produkt der Skalarmultiplikation
34  */
35 public static Vector3f multiply(Vector3f a, float b) {
36     return new Vector3f(
37         a.x * b,
38         a.y * b,
39         a.z * b
40     );
41 }
42
43 /**
44  * Dividiert einen Vektor durch ein Skalar.
45  * @param a zu dividierender Vektor
46  * @param b zu dividierendes Skalar
```

```
47   * @return der Quotient der Skalardivision
48   */
49  public static Vector3f divide(Vector3f a, float b) {
50      return new Vector3f(
51          a.x / b,
52          a.y / b,
53          a.z / b
54      );
55  }
56
57  /**
58   * Bildet das Skalarprodukt zweier Vektoren.
59   * @param a Vektor a
60   * @param b Vektor b
61   * @return das Skalarprodukt der beiden Vektoren
62   */
63  public static float dot(Vector3f a, Vector3f b) {
64      return
65          (a.x * b.x) +
66          (a.y * b.y) +
67          (a.z * b.z);
68  }
69
70  /**
71   * Bildet das Kreuzprodukt zweier Vektoren.
72   * @param a Vektor a
73   * @param b Vektor b
74   * @return das Kreuzprodukt der beiden Vektoren
75   */
76  public static Vector3f cross(Vector3f a, Vector3f b) {
77      return new Vector3f(
78          a.y * b.z - a.z * b.y,
79          a.z * b.x - a.x * b.z,
80          a.x * b.y - a.y * b.x
81      );
82  }
83
84  /**
85   * Bestimmt die Laenge eines Vektors.
86   * @param a der Vektor, dessen Laenge bestimmt werden soll
87   * @return die Laenge des Vektors
88   */
89  public static float magnitude(Vector3f a) {
90      return (float) Math.sqrt(
91          a.x * a.x +
92          a.y * a.y +
93          a.z * a.z
94      );
95  }
96
97  /**
98   * Erzeugt aus einem Vektor einen neuen Vektor mit der Laenge 1, ohne dabei die
         Richtung des urspruenglichen Vektors zu veraendern (normalisieren).
99   * @param a der Vektor, der normalisiert werden soll (Laenge sollte nicht 1 sein)
100  * @return der normalisierte Vektor mit der Laenge 1
101  */
102 public static Vector3f normalize(Vector3f a) {
103     float magnitude = magnitude(a);
104     return new Vector3f(
```

```
105             a.x / magnitude,
106             a.y / magnitude,
107             a.z / magnitude
108         );
109     }
110
111     /**
112      * Kehrt die Richtung um, in die ein Vektor zeigt, indem all seine Werte negiert
             werden.
113      * @param a der Vektor, der umgekehrt werden soll
114      * @return der umgekehrte Vektor
115      */
116     public static Vector3f inverse(Vector3f a) {
117         return new Vector3f(
118             -a.x,
119             -a.y,
120             -a.z
121         );
122     }
```

Listing B.1: Implementierung von Vektorrechenoperationen

B.2 Oberklasse aller Szenenobjekte

Die Klasse `SceneObjectBase` stellt die Basis für alle Objekte in einer Szene dar, die durch das Programm als Bild ausgegeben werden kann. Jedes solche Objekt definiert sich mindestens über seine Position in der Szene.

```
1   /**
2    * Eine Implementierungshilfe fuer alle Objekte,
3    * die sich in einer Szene befinden und eine Position haben.
4    */
5   public abstract class SceneObjectBase {
6
7       public final Vector3f pos;
8
9       /**
10       * Erzeugt ein neues Objekt an der gegebenen Position.
11       * @param x x-Wert der Position
12       * @param y y-Wert der Position
13       * @param z z-Wert der Position
14       */
15      public SceneObjectBase(float x, float y, float z) {
16          pos = new Vector3f(x, y, z);
17      }
18
19  }
```

Listing B.2: Oberklasse aller Szenenobjekte

C Erklärung der selbstständigen Anfertigung der Arbeit

Hiermit versichere ich, dass ich die Arbeit selbstständig angefertigt, keine anderen als die angegebenen Hilfsmittel benutzt und die Stellen der Facharbeit, die im Wortlaut oder im wesentlichen Inhalt aus anderen Werken entnommen wurden, mit genauer Quellenangabe kenntlich gemacht habe.

David Albers

Haltern am See, den 28. Februar 2020