Abbildung 1: Durch Raytracing erzeugtes Bild eines komplexeren, texturierten Modells mit Phong-Shading.

CG/task1 — Simples Raytracing

Raytracing ist ein Verfahren, mit dem die Sichtbarkeit von Objekten von einem Punkt aus in einer gegebenen Szene bestimmt werden kann. Ausgehend von einem Startpunkt wird ein Strahl konstruiert und gegen die Objekte einer Szene auf Schnittpunkte getestet. Auf diese Art kann ein Bild einer virtuellen Szene generiert werden, indem vom Augpunkt einer virtuellen Kamera aus Sichtstrahlen durch alle Pixel in der Bildebene erzeugt und deren erster (d.h. der dem Startpunkt näheste) Schnittpunkt mit einem Objekt der Szene gesucht wird. Die Farbe eines jeden Pixels ergibt sich dann entsprechend des so gefundenen Schnittpunkts auf dem getroffenen Objekt.

1 Raycasting Grundlagen

Ein Strahl ist mathematisch beschrieben durch

$$\mathbf{r}(t) = \mathbf{p} + t \cdot \mathbf{d} \tag{1}$$

wobei $\mathbf{p}$ die Koordinaten des Startpunkes und $\mathbf{d}$ die Richtung des Strahls angeben, der Strahlparameter $t > 0$ entspricht der Distanz des Punktes $\mathbf{r}(t)$ auf dem Strahl vom Startpunkt $\mathbf{p}$ als Vielfaches der Länge der Strahlrichtung $\|\mathbf{d}\|$.

Herzstück eines Raycasters ist die in Abbildung 2 illustrierte Generierung der Sichtstrahlen – auch *Primärstrahlen* genannt – durch die Bildebene. Um ein Rasterbild der Szene mit Auflösung $r_x \times r_y$ zu erzeugen, wird die Szene durch die Bildebene abgetastet. Die Bildebene befindet sich im Abstand f entlang der Blickrichtung $-\mathbf{z}$ vor dem Augpunkt, der abzutastende Bereich der Bildebene ist durch seine Breite w_s und Höhe h_s gegeben. Jedem Pixel des Ausgabebildes entspricht ein Punkt in der Bildebene, durch den ein Strahl geschossen wird. Das erste Objekt, auf das der Strahl trifft, bestimmt die Farbe des Pixels. Zu beachten ist dabei, dass die Pixel eines Bildes in der Regel zeilenweise, beginnend von links oben indiziert werden, der Pixel $(0,0)$ liegt also links oben im Bild, der Pixel $(r_x - 1, r_y - 1)$ rechts unten. Die Samplepositionen ergeben sich aus der Unterteilung der Bildebene in ein regelmäßiges Gitter mit $r_x \times r_y$ Zellen, die Strahlen verlaufen dann durch die Mittelpunkte der Gitterzellen.

Um realistisch wirkende Bilder wie das in Abbildung 1 zu erzeugen, ist es essentiell, den Effekt von Licht und Schatten auf die Objekte in einer virtuellen Szene zu modellieren. Eine vollständige Simulation sämtlicher physikalischer Aspekte der Interaktion von Licht und Materie ist allerdings in der Regel nicht praktikabel. Um mit begrenzten Ressourcen ein möglichst gutes Ergebnis zu erzielen, bedient man sich physikalisch inspirierter, aber stark vereinfachter Beleuchtungsmodelle um die Farben von Objekten zu bestimmen. Ein simples und in der Praxis weit verbreitetes Beleuchtungsmodell ist Phong-Shading.

Einfache geometrische Formen wie Kugeln, Quader und Ebenen sind in der Regel allerdings nicht ausreichend um realistische Szenen darzustellen. Zur Modellierung komplexer Objekte (wie das in Abbildung 1 zu sehende Raumschiff) werden meistens Dreiecksnetze verwendet, die im bereitgestellten Framework bereits unterstützt werden. Desweiteren ist es in der Regel erstrebenswert, die Kamera in der Szene frei platzieren zu können anstatt die Szene um eine fixe Kamera herum zu erstellen.

2 Kameramodell

Wie in Abbildung 2 dargestellt, erfolgt die Generierung der Sichtstrahlen für eine frei platzierbare Kamera prinzipiell gleich wie bei einer fixen Kamera, mit dem einzigen Unterschied, dass die Strahlen nun vom Ursprung $\mathbf{p}_{\text{eye}}$ eines lokalen Kamerakoordinatensystems ausgesandt werden und die Bildebene relativ zu den Basisvektoren $\mathbf{u}$, $\mathbf{v}$, $\mathbf{w}$ dieses lokalen Kamerakoordinatensystems ausgerichtet ist. Die Abmessungen der Bildebene sind durch die Breite w_s und Höhe h_s gegeben, der Abstand zum Augpunkt entspricht der Brennweite

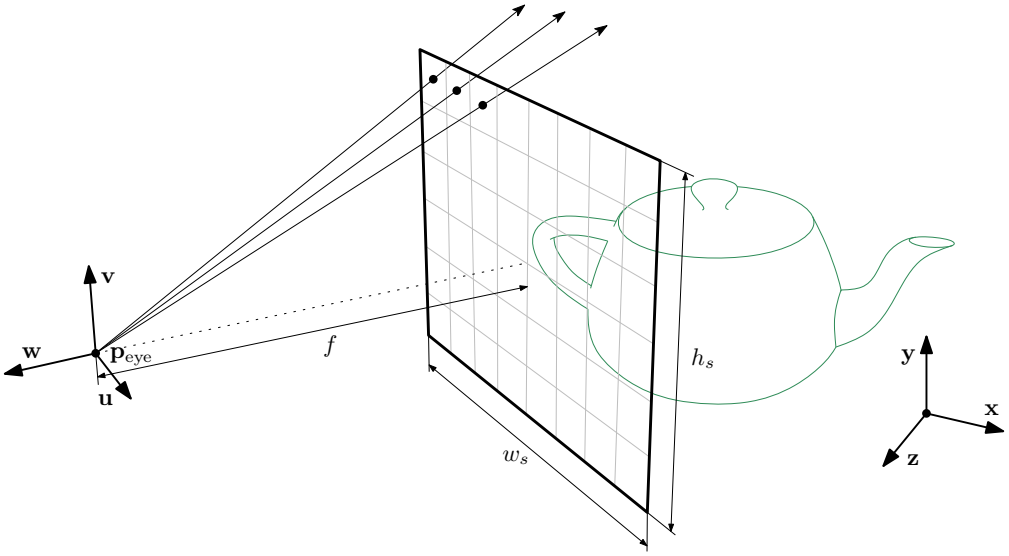


Abbildung 2: Sichtstrahlen werden von der Kameraposition aus durch die Bildebene geschossen. Orientierung und Position von Kamera und Bildebene sind durch das durch $\mathbf{u}$, $\mathbf{v}$, $\mathbf{w}$ und $\mathbf{p}_{\text{eye}}$ gegebene, lokale Kamerakoordinatensystem sowie den Abstand f der Bildebene bestimmt. Achsen und Ursprung können sich dabei stark vom Koordinatensystem der Szene ($\mathbf{x}$,$\mathbf{y}$,$\mathbf{z}$) unterscheiden.

f der virtuellen Kamera. Da eine uniforme Abtastung der Bildebene gewünscht ist, gilt

$$\frac{w_s}{h_s} = \frac{r_x}{r_y},$$

wobei r_x und r_y die Auflösung des Bildes in x bzw. y Richtung sind. Jedem Pixel des Ausgabebildes entspricht ein Punkt in der Bildebene, durch den ein Strahl geschossen wird. Das erste Objekt, auf das der Strahl trifft, bestimmt die Farbe des Pixels. Zu beachten ist dabei, dass die Pixel eines Bildes in der Regel zeilenweise, beginnend von links oben indiziert werden, der Pixel $(0,0)$ liegt also links oben im Bild, der Pixel $(r_x - 1, r_y - 1)$ rechts unten. Die Samplepositionen ergeben sich aus der Unterteilung der Bildebene in ein regelmäßiges Gitter mit $r_x \times r_y$ Zellen, die Strahlen verlaufen dann durch die Mittelpunkte der Gitterzellen.

Eine gängige Methode um die Lage der Kamera zu spezifizieren, ist durch Angabe der Position des Augpunktes $\mathbf{p}_{\text{eye}}$, eines *Lookat-Punktes* $\mathbf{p}_{\text{lookat}}$ und eines *Up-Vektors* $\mathbf{v}_{\text{up}}$. Wie in Abbildung 3 dargestellt, entspricht der Lookat-Punkt dabei dem Punkt auf den die Kamera gerichtet ist, während der Up-Vektor die Rotation der Kamera um die Blickrichtung festlegt (er gibt an, welche Richtung für die Kamera „nach oben“ ist). Mit diesen Informationen können dann die Basisvektoren des Kamerakoordinatensystems berechnet werden:

$$\mathbf{w} = \frac{\mathbf{p}_{\text{eye}} - \mathbf{p}_{\text{lookat}}}{\left\|\mathbf{p}_{\text{eye}} - \mathbf{p}_{\text{lookat}}\right\|} \qquad \mathbf{u} = \frac{\mathbf{v}_{\text{up}} \times \mathbf{w}}{\left\|\mathbf{v}_{\text{up}} \times \mathbf{w}\right\|} \qquad \mathbf{v} = \mathbf{w} \times \mathbf{u}. \tag{2}$$

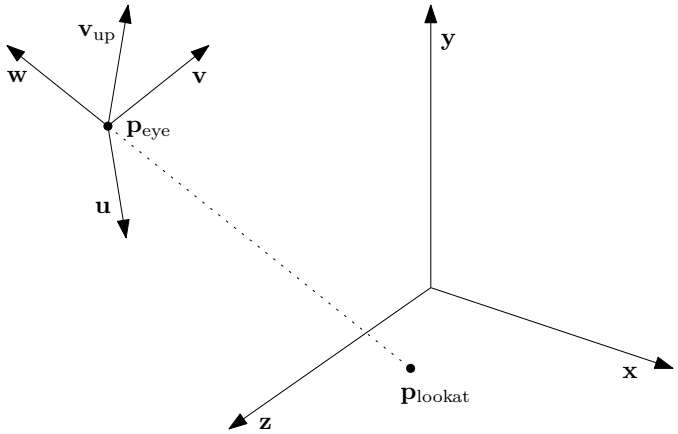


Abbildung 3: Referenzdarstellung eines Kamerakoordinatensystems mit Basisvektoren $\mathbf{u}$, $\mathbf{v}$ und $\mathbf{w}$, sowie den vorgegebenen Konfigurationsparametern $\mathbf{p}_{\text{eye}}$, $\mathbf{p}_{\text{lookat}}$ und $\mathbf{v}_{\text{up}}$.

3 Shading Grundlagen

Der Vorgang der Berechnung für die Farbe eines Punktes auf einem sichtbaren Objekt wird im Allgemeinen als *Shading* bezeichnet. Der Einfachheit halber betrachtet man oft nur den Einfluss der direkten Beleuchtung der Objektoberfläche durch idealisierte, punktförmige Lichtquellen. Wie in Abbildung 4 illustriert, reduziert das Problem sich so auf die Berechnung des in Richtung $\mathbf{v}$ zum Betrachter reflektierten Anteils des aus Richtung $\mathbf{l}$ von einer Lichtquelle auf den Punkt $\mathbf{p}$ einfallenden Lichtes. Die Beleuchtung durch mehrere Lichtquellen folgt dem Superpositionsprinzip und entspricht daher einfach der Summe der Einflüsse der einzelnen Lichtquellen.

Das Reflexionsverhalten einer Objektoberfläche wird in der Regel als aus einem diffusen Anteil und einem spekularen Anteil zusammengesetzt modelliert. Der diffuse Anteil c_d entspricht dem Teil des einfallenden Lichtes, der von einer Oberfläche gleichmäßig in alle Richtungen gestreut wird. Er kann nach dem Lambert'schen Gesetz ermittelt werden und hängt nur vom Lichteinfallswinkel θ auf die Oberfläche ab, deren Ausrichtung zur Lichtquelle durch die Oberflächennormale $\mathbf{n}$ beschrieben ist:

$$c_d = k_d \cdot \max(\cos\theta, 0); \tag{3}$$

k_d ist dabei der diffuse Reflexionskoeffizient des Materials. Der spekulare Anteil entspricht dem Teil des einfallenden Lichtes, der vorwiegend in die um die Oberflächennormale gespiegelte Lichtrichtung $\mathbf{r}$ reflektiert wird (Glanzeffekte). Im Phong-Shading-Modell wird der in Richtung $\mathbf{v}$ reflektierte, spekulare Anteil c_s basierend auf dem Winkel α_r zwischen $\mathbf{v}$ und $\mathbf{r}$ abgeschätzt:

$$c_s = \begin{cases} k_s \cdot (\max(\cos\alpha_r, 0))^m & \text{wenn } \cos\theta > 0 \\ 0 & \text{sonst} \end{cases} \tag{4}$$

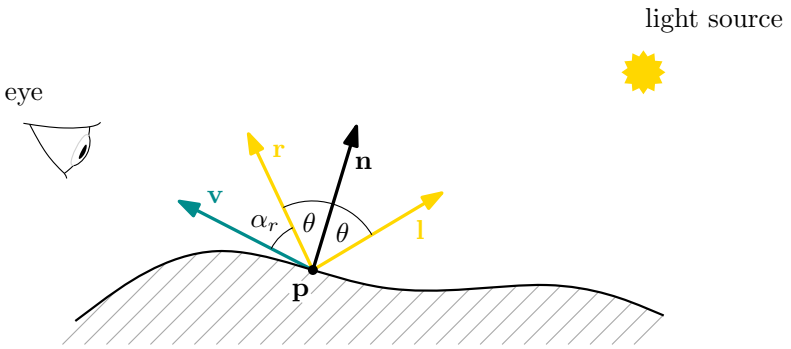
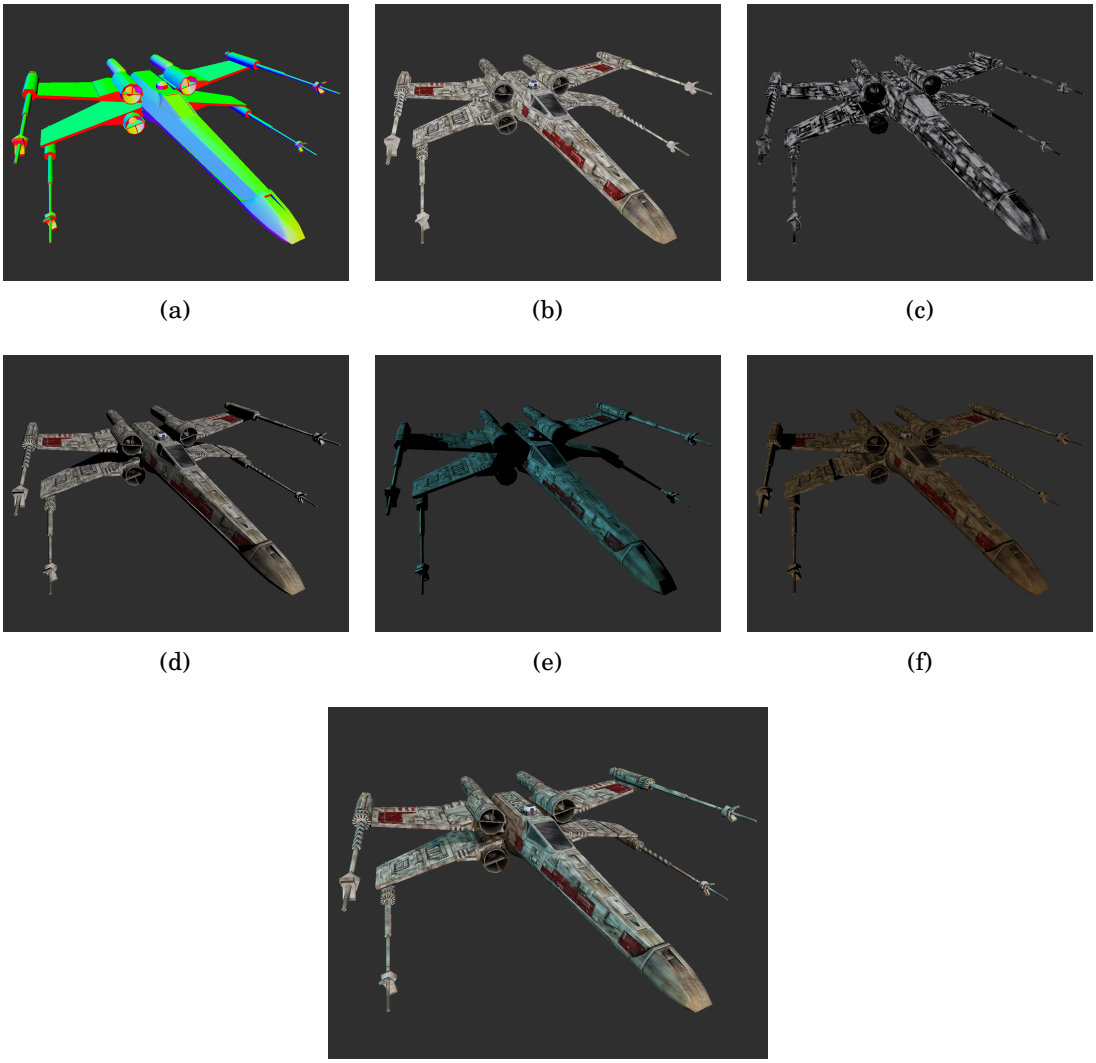


Abbildung 4: Lokale Betrachtung der Beleuchtung eines Punktes **p** auf der Oberfläche eines Objektes durch eine einzelne Lichtquelle. Der in Richtung **v** zum Betrachter reflektierte Anteil des aus Richtung **l** einfallenden Lichts hängt wesentlich vom Einfallswinkel θ ab. Die Normale **n** bestimmt die Ausrichtung der Oberfläche zur Lichtquelle und nimmt daher eine zentrale Stellung in der Beleuchtungsberechnung ein. Der Winkel α_r zwischen der Richtung **v** zum Betrachter und der reflektierten Lichteinfallsrichtung **r** dient als Maß für den spekularen Anteil, während der diffuse Anteil nur von θ abhängt.

wobei k_s der spekulare Reflexionskoeffizient des Materials ist. Der Faktor m moduliert die Schärfe der spekularen Reflexion und entspricht damit der Glattheit der Oberfläche; je größer m, desto schärfer die spekulare Reflexion. Abbildung 5 illustriert die einzelnen Komponenten der Beleuchtungsberechnung für das Rendering aus Abbildung 1.



(a) (b) (c)

(d) (e) (f)

(g)

Abbildung 5: Komponenten der Beleuchtungsberechnung. (a) Oberflächennormalen (als Farbe dargestellt); (b), (c) die diffusen und spekularen Reflexionskoeffizienten, bestehend aus je 3 Werten für die unterschiedlichen Farbkanäle. Die Koeffizienten variieren in diesem Beispiel mit Textur über die Oberfläche; (d), (e), (f) individuelle Beiträge der drei Lichtquellen der Szene; (g) Endresultat.

3.1 Point- und Spot Lights

Point Lights Die einfachsten analytische Lichtquellen sind sogenannte *Point Lights* (Punktlichtquellen). Bei Vernachlässigung des Ausbreitungsmediums (Annahme: ideales Vakuum) ist jede Punktlichtquelle durch eine Position p im Raum, so wie eine Farbe `color` definiert, wie hier ein Beispiel aus der Konfigurationsdatei `cone_shadow.json`:

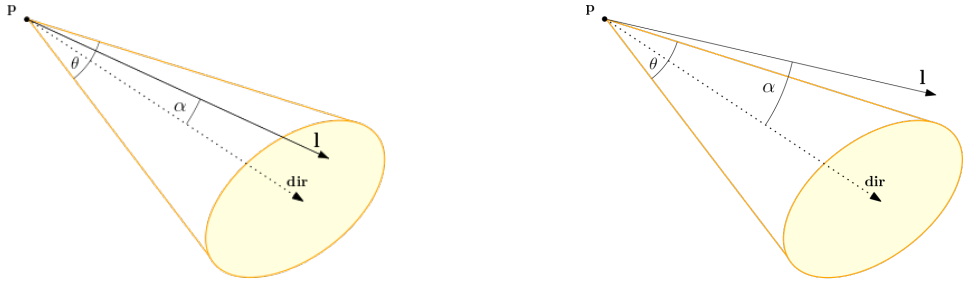
```
"lights":
[
{ "p": [1000, 800, -300], "color": [1.0, 1.0, 1.0] },
{ "p": [-600, 400,  800], "color": [0.1, 0.5, 0.6] },
{ "p": [ 600, 300,  300], "color": [0.3, 0.2, 0.1] }
]
```

Die `color` ist auch gleichzeitig mit der Intensität des Lichts gekoppelt - eine `color` von `[1.0, 1.0, 1.0]` beschreibt eine Intensität von 1.0 in rot, grün und blau, also helles, weißes Licht, während eine `color` von `[0.1, 0.0, 0.0]` sehr schwaches, rotes Licht beschreibt. Durch die Vernachlässigung des Ausbreitungsmediums gibt es keine Dämpfung - jeder Punkt im Raum der nicht direkt von einem anderen Objekt blockiert wird (siehe Abschnitt 6.5) wird durch die volle Intensität jeder Lichtquelle beleuchtet.

Spot Lights Neben Punktlichtquellen werden auch sogenannte Spot Lights ("Scheinwerferlichter") implementiert. Im Gegensatz zur Punktlichtquelle welches in alle Richtungen gleichmäßig strahlt, wird ein Spot Light durch eine Kegelform begrenzt. Eine Taschenlampe bietet eine gute Intuition für ein solches Licht. Ein Spot Light wird beschrieben durch einen Richtungsvektor **dir**, welcher die Orientierung beschreibt, einen Öffnungswinkel θ, welcher den gewünschten Lichtkegel begrenzt. Zusätzlich benötigen Spot Lights, genau so wie Point Lights, eine Position **p** und eine Lichtfarbe `color`, und ein Beispiel für Spot Lights in den Konfigurationsdateien der Übung wäre:

```
"spotlights":
[
{ "p": [0, 10, 0], "color": [0.2, 0.2, 0.2], "dir": [0.0, -1.0, 0.0], "angle": 0.785398 },
{ "p": [20, 20, 0], "color": [1.0, 1.0, 1.0], "dir": [-1.0, -1.0, 0.0], "angle": 0.785398 }
]
```

Wie in Abbildung 6 zu sehen ist, wird mithilfe des Winkels α zwischen dem Vektor **dir** in Richtung des Spot Lights und dem Vektor **l** zur Objektoberfläche (siehe Abbildung 4) die Beleuchtung berechnet. Dieser ergibt sich grundsätzlich aus demselben diffusen Anteil



(a) (b)

Abbildung 6: (a) Ist der Winkel $\alpha \leq \frac{\theta}{2}$, so liegt der Strahl von der Lichtquelle zum Objekt innerhalb der Taschenlampe und das Objekt wird an diesem Punkt belichtet. (b) Ist der Winkel $\alpha > \frac{\theta}{2}$, liegt der Punkt außerhalb des Kegels der Taschenlampe und wird somit nicht beleuchtet.

c_d und spekularen Anteil c_s wie beim Point Light. Beim Spot Light werden beide Anteile allerdings zusätzlich mit dem Faktor s multipliziert, welcher die Bedingung für den Kegel darstellt:

$$s = \begin{cases} 1 & \text{wenn } \alpha \leq \frac{\theta}{2} \\ 0 & \text{sonst} \end{cases} \tag{5}$$

Dadurch werden alle Pixel innerhalb des Kegels komplett beleuchtet. Oberflächenpunkte, die sich außerhalb des Lichtkegels befinden werden entsprechend nicht beleuchtet. Diese Methode führt zu einem sehr scharfen Rand um den Kegelausschnitt. Um einen realistischeren Farbverlauf entlang der Oberfläche zu erzeugen, wird die Belichtung innerhalb des Kegels interpoliert.
Die Beleuchtung soll abgeschwächt werden, umso weiter **l** von **dir** abweicht.

Dafür bedienen wir uns der Skalarprodukte der Vektoren bzw. der Cosinus-Winkel

$$\cos(\alpha) = \mathbf{l} \cdot \mathbf{dir} \tag{6}$$

und schwächen die Beleuchtung mittels Interpolation ab, und erhalten

$$s = \begin{cases} \text{clamp}\left(\frac{\cos(\alpha)-\cos(\frac{\theta}{2})}{1-\cos(\frac{\theta}{2})}, 0, 1\right) & \text{wenn } \alpha \leq \frac{\theta}{2} \\ 0 & \text{sonst} \end{cases}. \tag{7}$$

4 Schnittberechnung zwischen Strahl und Ebene

Unter dem Begriff Ebene versteht man eine unbegrenzt ausgedehnte zweidimensionale Fläche. In Vektor-Notation können wir eine Ebene als $(\mathbf{p} - \mathbf{p}_0) \cdot \mathbf{n} = 0$ definieren, wobei $\mathbf{n}$ die Normale, $\mathbf{p}$ eine Menge an Punkten und $\mathbf{p}_0$ einen Punkt auf der Ebene beschreibt. Dementsprechend können wir eine Ebene durch einen Punkt und eine Normale ausdrücken, wobei $\mathbf{n}$ die Orientierung der Ebene bestimmt.

Ein Strahl $\mathbf{r}(t)$ ist wie bereits erwähnt, definiert durch $\mathbf{r}(t) = \mathbf{p} + \mathbf{d}t$, wobei $\mathbf{p}$ die Startposition und $\mathbf{d}$ die Richtung des Strahls ist. Wenn $\mathbf{d} \cdot \mathbf{n} \neq 0$, schneidet der Strahl die Ebene in genau einem Punkt. Wenn $\mathbf{d} \cdot \mathbf{n} = 0$, liegt der Strahl parallel zur Ebene und es existiert kein relevanter Schnittpunkt. Wenn ein Schnittpunkt mit der Ebene existiert, muss noch der Strahlparameter t bestimmt werden. Durch Einsetzen in die Ebenengleichung erhält man:

$$((\mathbf{p} + \mathbf{d}t) - \mathbf{p}_0) \cdot \mathbf{n} = 0, \tag{8}$$

und Expandieren ergibt

$$(\mathbf{d} \cdot \mathbf{n})t + (\mathbf{p} - \mathbf{p}_0) \cdot \mathbf{n} = 0. \tag{9}$$

Schlussendlich können wir die Gleichung nach t auflösen und erhalten

$$t = \frac{(\mathbf{p}_0 - \mathbf{p}) \cdot \mathbf{n}}{\mathbf{d} \cdot \mathbf{n}}. \tag{10}$$

Im Framework ist eine Ebene definiert durch die Normale $\mathbf{n}$ und einen Offset w in die Richtung der Normalen $\mathbf{n}$. Der Punkt auf der Ebene $\mathbf{p}_0$ kann dann durch w ersetzt werden und man erhält:

$$t = \frac{w - \mathbf{p} \cdot \mathbf{n}}{\mathbf{d} \cdot \mathbf{n}}. \tag{11}$$

5 Schnittberechnung zwischen Strahl und Würfel

Im Zusammenhang mit Raytracing können zwei Arten von Würfeln unterschieden werden: axis-aligned und oriented. Eine sogenannte Axis-Aligned Bounding Box (AABB) ist an den Hauptachsen des Weltkoordinatensystems ausgerichtet. Im Gegensatz dazu ist die sognannte Oriented Bounding Box (OBB) beliebig rotiert und damit nicht an den Hauptachsen des Weltkoordinatensystems ausgerichtet.

5.1 Schnitt mit AABB

Da der Würfel an den Hauptachsen ausgerichtet ist, kann man die Schnittpunkte mit einem Strahl über die 6 Flächen des Würfels relativ einfach bestimmen. Der Würfel ist

durch einen minimalen Eckpunkt $\mathbf{b}_{min}$ und einen maximalen Eckpunkt $\mathbf{b}_{max}$ definiert. Für jede Dimension gibt es die Distanz zum Eintrittspunkt des Strahls in den Würfel `t_near` sowie die Distanz zum Austrittspunkt `t_far`. Der Strahl schneidet den Würfel genau dann, wenn das Maximum aller nahen Distanzen größer oder gleich dem Minimum aller fernen Distanzen ist.

Für einen Würfel, der durch seinen minimalen Eckpunkt $\mathbf{b}_{min}$ und maximalen Eckpunkt $\mathbf{b}_{max}$ definiert ist, und einen Strahl $\mathbf{r}(t) = \mathbf{p} + \mathbf{d}t$ lassen sich die Strahlparameter t_{x_1} und t_{x_2} für die zwei Schnittpunkte entlang der X-Achse wie folgt berechnen:

$$t_{x_1} = \frac{b_{min_x} - p_x}{d_x} \qquad t_{x_2} = \frac{b_{max_x} - p_x}{d_x} \tag{12}$$

Analog kann man für die restlichen Achsen Y und Z dieselben Berechnungen durchführen. Anschließend berechnet man die nahe und ferne Distanz über alle drei Achsen, indem man das Maximum der nahen und das Minimum der fernen Distanzen findet:

$$t_{near} = \max\{\min(t_{z_1}, t_{z_2}), \min(t_{y_1}, t_{y_2}), \min(t_{x_1}, t_{x_2})\} \tag{13}$$

$$t_{far} = \min\{\max(t_{z_1}, t_{z_2}), \max(t_{y_1}, t_{y_2}), \max(t_{x_1}, t_{x_2})\} \tag{14}$$

Der Strahl schneidet den Würfel mit den Strahlparametern t_{near} und t_{far}, wenn $t_{near} \leq t_{far}$. Für die Beleuchtunsberechnung ist in diesem Fall dann nur die Distanz zum Eintrittspunkt t_{near} relevant.

5.2 Schnitt mit OBB

Im lokalen Koordinatensystem des Würfels können die beiden Eckpunkte $\mathbf{b}_{min}$ und $\mathbf{b}_{max}$ über die Skalierung $\mathbf{s}$ des Würfels bestimmt werden:

$$\mathbf{b}_{min} = -\frac{\mathbf{s}}{2} \qquad \mathbf{b}_{max} = \frac{\mathbf{s}}{2} \tag{15}$$

Für die Schnittberechnung wird der Strahl $\mathbf{r}(t) = \mathbf{p} + \mathbf{d}t$ in das lokale Koordinatensystem des Würfels transformiert, da dann einfach ein AABB-Test wie oben beschrieben durchgeführt werden kann. Um $\mathbf{p}$ und $\mathbf{d}$ in das lokale Koordinatensystem des Würfels zu transformieren benötigt man die Inverse der Modelmatrix M des Würfels sowie die Inverse der Rotationsmatrix R:

$$p_{local} = M^{-1} \cdot \mathbf{p} \qquad d_{local} = R^{-1} \cdot \mathbf{d} \tag{16}$$

Nachdem der Würfel an den Hauptachsen seines lokalen Koordinatensystems ausgerichtet ist und der Strahl in dieses lokale Koordinatensystem transformiert wurde, kann dann einfach ein AABB-Test zur Bestimmung des Strahlparameters t_{near} durchgeführt werden.

6 Aufgabe

Ziel dieser Übung ist es, die Kernelemente eines simplen Raytracers zu implementieren. Dieser Raytracer soll eine beliebige Kameraposition unterstützen, Schnittpunkte mit Ebenen und Würfeln sowie Phong-Beleuchtung und Schatten für Point- und Spot- Lights unterstützen.

Das zur Verfügung gestellte Framework lädt bereits eine Szene aus einer JSON Konfigurationsdatei und kümmert sich um die Ausgabe des Ergebnisbildes in eine PNG-Datei. Der Pfad zur Konfigurationsdatei wird dem Programm als Argument übergeben. Verschiedene Testszenen finden Sie im Unterverzeichnis «`data/task1/`». Erfolgreiche Ausführung erzeugt eine Datei «`<config-name>.png`» in «`output/task1/`». Mit den Parametern `-t` bzw. `--num-threads` *<Anzahl>* wird das Framework dazu veranlasst mit mehreren Threads gleichzeitig zu Rendern um die Ausgabe zu beschleunigen. Mit `--num-threads` kann die Anzahl der zu verwendenden Threads per Argument angegeben werden; `-t` verwendet so viele Threads wie es logische Prozessoren im System gibt. Mit `--cache` können die Zwischenergebnisse der Schnittberechnungen sowie der Transformationsmatrizen der Würfel für schnellere Performance gecached werden. Das Framework unterstützt auch bereits Schnittpunkte mit beliebigen Dreiecksnetzen. Dadurch ist es möglich die Beleuchtung und die Schnittpunktberechnung getrennt zu testen.

6.1 Strahlgenerierung für allgemeine Kamera (4 Punkte)

Implementieren Sie die Funktion

```
void render(image2D<float3>& framebuffer,
            int left, int top, int right, int bottom,
            const Scene& scene,
            const Camera& camera,
            const Pointlight* lights,
            std::size_t num_lights,
            const Spotlight* spotlights,
            std::size_t num_spotlights,
            const float3& background_color,
            int max_bounces)
```

in «task1.cpp» so, dass sie für den durch `left`, `top`, `right`, `bottom` gegebenen Bereich des Framebuffers ein Bild der Szene `scene` generiert. Der Ablauf ist dabei so, dass pro Pixel ein Strahl generiert wird (siehe Abschnitt 2 und Abbildung 2 für eine Erklärung wie die einzelnen Sichtstrahlen generiert werden). Die Struktur `camera` enthält dabei die zu verwendenden Kameraparameter `camera.w_s` (Breite der Bildebene), `camera.f` (Brennweite) sowie die Position des Augpunktes `camera.eye`, den Lookat-Punkt `camera.lookat` und Up-Vektor (`camera.up`, siehe Abschnitt 2). Rufen Sie `scene.findClosestHit()` auf, um den nähesten Schnittpunkt zu finden. `scene.findClosestHit()` liefert bereits für Schnitte mit Kegeln und Dreiecksnetzen korrekte Ergebnisse. Für Tests mit Ebenen und Würfeln müssen Sie erst die jeweiligen Funktionen dafür Implementieren (siehe Abschnitte 5 und 4). Wenn ein relevanter Schnittpunkt vorliegt, soll dieser an die Funktion `shade()` übergeben werden. `lights` ist ein Pointer auf das erste Element eines Arrays aus `num_lights` Lichtquellen, welche für die Berechnung der Beleuchtung herangezogen werden sollen. Die Parameter `spotlights` und `num_spotlights` beinhalten die Strukturen für die Beleuchtung mit Spotlights. `background_color` ist jene Farbe, die in den Framebuffer geschrieben werden soll wenn an einem Pixel kein Objekt sichtbar ist. Der Parameter `max_bounces` ist nur für die Lösung des Bonustasks relevant.

6.2 Strahlschnitt mit Ebenen (2 Punkt)

Implementieren Sie die Funktion

```
const Plane* findClosestHitPlanes(const float3& p,
    const float3& d,
    const Plane* planes,
    std::size_t num_planes,
    float& t)
```

in «task1.cpp» so, dass sie für einen gegebenen Strahl den nähesten Schnittpunkt mit einer Ebene in der gegebenen Menge an Ebenen findet. Die Parameter `p` und `d` geben dabei den Startpunkt und die Richtung des zu schneidenden Strahls an. Die Menge an Ebenen

ist in Form eines Arrays `planes` mit `num_planes` Elementen vom Typ `Plane` gegeben. Jede `Plane` enthält die für die Schnittberechnung benötigten Ebenenparameter in `plane->p`. Falls ein Schnittpunkt gefunden wurde, soll `t` auf den entsprechenden Strahlparameter gesetzt werden und ein Pointer auf die näheste getroffene Ebene zurückgeliefert werden. Existiert kein Schnittpunkt, soll die Funktion einen Nullpointer zurückgeben.

Um den Schnittpunkt mit einer einzelnen Ebene zu bestimmen, ist die Funktion

```
bool intersectRayPlane(const float3& p,
   const float3& d,
   const float4& plane,
   float& t)
```

aufzurufen und wie in Abschnitt 4 beschrieben zu implementieren. Die Parameter `p` und `d` geben dabei wieder den Startpunkt und die Richtung des zu schneidenden Strahls an. Die Ebenenparameter **n** und der Offset w sind durch `plane` gegeben, wobei **n** durch `plane.x`, `plane.y` und `plane.z` bestimmt ist und w durch `plane.w` gegeben ist. Falls ein Schnittpunkt gefunden wurde, soll `t` auf den entsprechenden Strahlparameter gesetzt werden und `true` zurückgeliefert werden. Existiert kein Schnittpunkt, soll die Funktion `false` zurückgeben.

6.3 Strahlschnitt mit Würfel (3 Punkte)

Implementieren Sie die Funktion

```
const Cube* findClosestHitCubes(const float3& p,
   const float3& d,
   const Cube* cubes,
   std::size_t num_cubes,
   float& t)
```

in «`task1.cpp`» so, dass sie für einen gegebenen Strahl den nähesten Schnittpunkt mit einem orientierten Würfel (OBB) in der gegebenen Menge an Würfeln findet. Die Parameter `p` und `d` geben dabei den Startpunkt und die Richtung des zu schneidenden Strahls an. Die Menge an Würfel ist in Form eines Arrays `cubes` mit `num_cubes` Elementen vom Typ `Cube` gegeben. Falls ein Schnittpunkt gefunden wurde, soll `t` auf den entsprechenden Strahlparameter gesetzt werden und ein Pointer auf den nähesten getroffenen Würfel zurückgeliefert werden. Existiert kein Schnittpunkt, soll die Funktion einen Nullpointer zurückgeben.

Um den Schnittpunkt mit einem einzelnen orientierten Würfel (OBB) zu bestimmen, ist die Funktion

```
bool intersectRayCube(const float3& p,
const float3& d,
const Cube* cube,
float& t)
```

aufzurufen und wie in Abschnitt 5 beschrieben zu implementieren. Die Parameter `p` und `d` geben dabei wieder den Startpunkt und die Richtung des zu schneidenden Strahls in Weltkoordinaten (world space coordinates) an. Für die Schnittberechnung mit einem orientierten Würfel können die beiden Parameter mit den in der `Cube` Struktur enthaltenen Feldern `cube->inverse_model_matrix` und `cube->inverse_rotation_matrix` in das lokale Koordinatensystem des Würfels transformiert werden. Die Translations-, Rotations- und Modelmatrix werden beim Laden der Konfiguration durch das Framework mit den Rückgabewerten der entprechenden von Ihnen zu implementierenden Funktionen in «`task1.cpp`» initialisiert.

Die Funktion

```
float4x4 getRotationMatrix(const float3& rotation)
```

soll die $4x4$ Rotationsmatrix für die gegebenen Winkel `rotation.x`, `rotation.y`, `rotation.z` zurückliefern. Die Winkel liegen bereits in Radianten vor und müssen dementsprechend nicht konvertiert werden.

Die Funktion

```
getTranslationMatrix(const float3& position)
```

soll die $4x4$ Translationsmatrix für die gegebene 3D-Position `position.x`, `position.y`, `position.z` zurückliefern.

Die Funktion

```
float4x4 getModelMatrix(const float4x4& translation_matrix,
   const float4x4& rotation_matrix)
```

soll die $4x4$ Modelmatrix basierend auf der gegebenen Translationsmatrix `translation_matrix` und Rotationsmatrix `rotation_matrix` zurückliefern.

Für eine korrekte Schnittberechnung mit orientierten Würfeln müssen diese drei Funktionen implementiert werden. Die von Ihnen zurückgegebenen Matrizen werden vom Framework invertiert und in der `Cube` Struktur in den oben beschriebenen Feldern `cube->inverse_model_matrix` und `cube->inverse_rotation_matrix` gespeichert. Im Testcase «`ray_cube_basic.json`» liegt der Würfel ohne Rotation im Ursprung des Weltkoordinatensystems. Dadurch ist die Schnittberechnung mit dem Würfel in dessen lokalen Koordinatensystems äquivalent zu einer Schnittberechnung in Weltkoordinaten. Alle anderen Testcases setzen die Implementierung von Schnitten mit orientierten Würfeln voraus.

Jede `Cube` Struktur beinhaltet das Attribut `cube.scale`. Dementsprechend kann die lokale Spannweite des Würfels **min** und **max** durch die gegebne Skalierung berechnet werden. Falls ein Schnittpunkt gefunden wurde, soll `t` auf den entsprechenden Strahlparameter gesetzt werden und `true` zurückgeliefert werden. Existiert kein Schnittpunkt, soll die Funktion `false` zurückgeben.

6.4 Phong Shading (Point- und Spot Lights) (4 Punkte)

Die Funktion

```
float3 shade(const float3& p,
             const float3& d,
             const HitPoint& hit,
             const Scene& scene,
             const Pointlight* lights,
             std::size_t num_lights
             const Spotlight* spotlights,
             std::size_t num_spotlights
             )
```

soll, wie in Abschnitt 3 beschrieben, die Farbe für einen im Parameter `hit` gegebenen Oberflächenpunkt berechnen. `p` und `d` entsprechen wieder dem Startpunkt bzw. der Richtung des Strahls welcher den Punkt `hit` getroffen hat. `hit.position` enthält die Position und `hit.normal` die Oberflächennormale des getroffenen Punktes. Die Position und Farbe jeder der `num_lights` Punktlichtquellen im Array `lights` findet sich im Member `position` bzw. `color` des jeweiligen `Pointlight` bzw. `SpotLight` Objektes. Die Richtungsvektoren und Öffnungswinkel der Spot Lights finden sich im Member `direction` bzw. `angle` des jeweiligen `SpotLight` Objektes. Die Öffnungswinkel der Spot Lights sind in Radiant gegeben. Berechnen Sie unter Verwendung der diffusen und spekularen Koeffizienten `hit.k_d`, `hit.k_s` und `hit.m` die Beleuchtung der Oberfläche am gegebenen Punkt durch alle relevanten Lichtquellen mittels Superposition und returnieren Sie die entsprechende Farbe.

6.5 Schatten (3 Punkte)

Ein einfacher Weg um festzustellen, ob ein Punkt von einer Lichtquelle beleuchtet wird oder nicht, ist, einen Strahl von diesem Punkt aus zur Lichtquelle zu konstruieren und zu testen, ob dieser Strahl zwischen Punkt und Lichtquelle auf ein Objekt trifft oder nicht. Es ist dabei nicht notwendig, den nähesten Schnittpunkt zu finden, sondern nur, festzustellen, ob irgendein Schnittpunkt existiert.

Erweitern Sie die Funktion `shade()` so, dass die Beleuchtungsberechnung an einem gegebenen Punkt für jede Lichtquelle berücksichtigt, ob der Punkt in deren Schatten liegt oder nicht. Verwenden Sie die Methode `scene.intersectsRay()` um festzustellen, ob ein gegebener Strahl in Richtung der Lichtquelle von einem Objekt blockiert wird oder nicht. Zusätzlich zu Startpunkt und Richtung des zu testenden Strahls erwartet diese Methode in den Parametern `t_min` und `t_max` die untere und obere Grenze des Intervalls in welchem nach Schnittpunkten auf dem Strahl gesucht werden soll.

Aufgrund der begrenzten Genauigkeit von Gleitkommaoperationen, liegt der errechnete Schnittpunkt von Strahl und Dreieck im Allgemeinen nicht exakt in der Ebene des Drei-

ecks, sondern um einen kleinen Rundungsfehler davor oder dahinter, was zu Artefakten wie denen in Abbildung 7 gezeigten führen kann. Verwenden Sie daher nicht direkt den ermittelten Schnittpunkt `hit.position` als Startpunkt des Schattenstrahls, sondern verschieben Sie diesen zunächst noch um die in der Konstante `epsilon` vorgegebene Distanz in Richtung der Oberflächennormale. Auf diese Weise können Sie sicherstellen, dass der Startpunkt des neuen Strahls nicht irrtümlich hinter der zu beleuchtenden Oberfläche liegt und diese damit selbst als Schattenquelle interpretiert wird.

Das Framework besitzt bereits eine Methode für das Detektieren von Schnittpunkten mit Dreiecken und Kegeln. Der Test auf Schnittpunkte mit Ebenen und Würfeln muss erst in den Funktionen

```
bool intersectsRayPlane(const float3& p,
    const float3& d,
    const Plane* planes,
    std::size_t num_planes,
    float t_min,
    float t_max)
```

sowie

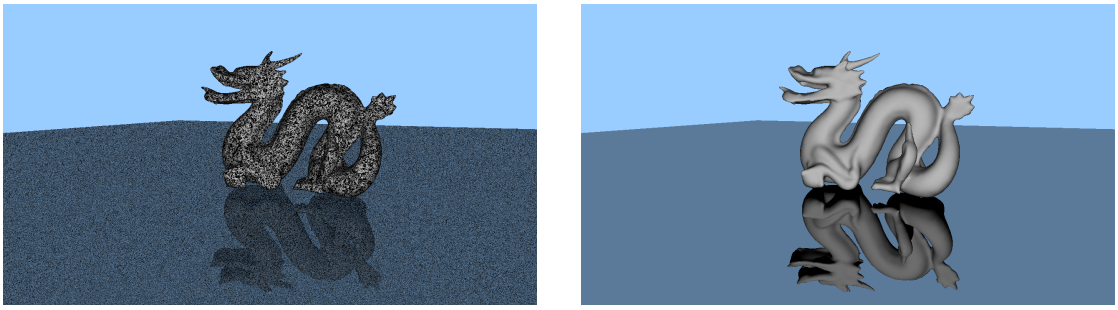
```
bool intersectsRayCube(const float3& p,
   const float3& d,
   const Cube* cubes,
   std::size_t num_cubes,
   float t_min,
   float t_max)
```

implementiert werden. Diese sollen jeweils `true` zurückliefern falls der gegebene Strahl (Startpunkt `p`, Richtung `d`) eine der gegebenen Ebenen bzw. Würfel trifft, ansonsten `false`. Dabei sind jeweils nur Schnittpunkte zu berücksichtigen, die zwischen `t_min` und `t_max` auf dem Strahl liegen.

6.6 Bonus: Rekursives Raytracing (3 Punkte)

Modifizieren Sie die Funktionen `render()` so, dass Oberflächen mit einer Glattheit `hit->m` von ∞[1] als Spiegelflächen behandelt werden. Der spekulare Anteil einer solchen Oberfläche wird bestimmt, indem ein weiterer Strahl entsprechend der Reflexion des auf die Oberfläche einfallenden Strahls $\mathbf{l}$ um die Oberflächennormale $\mathbf{n}$ ausgesandt wird.Versetzen Sie auch hier den neuen Startpunkt wieder um `epsilon` in Richtung der Oberflächennormale, um Artefakte durch Schnitte mit der Ausgangsoberfläche infolge von Rundungsfehlern zu vermeiden. Die Farbe des fertig beleuchteten Oberflächenpunktes ergibt sich aus der Farbe des rekursiv ausgesandten, reflektierten Strahls gewichtet mit dem spekularen Reflexionskoeffizienten `hit.k_s`. Um eine endlose Rekursion (beispielsweise im Falle zweier einander zugewandter Spiegel) zu vermeiden, sollen maximal `max_bounces` solcher Rekursionen

[1] verwenden Sie z.B. die Funktion `std::isinf()` um dies zu Testen



(a) (b)

Abbildung 7: (a) Aufgrund von Gleitkommaungenauigkeiten in der Schnittpunktberechnung liegen die Punkte an denen die Beleuchtung ausgewertet wird nicht exakt auf der Objektoberfläche, sondern teilweise um einen Rundungsfehler dahinter und damit im Schatten des eigenen Objektes, was zu einem stark verrauschten Rendering führen kann. (b) durch Verschieben des errechneten Schnittpunktes um eine kleine Distanz ϵ in Richtung der Dreiecksnormale wird sichergestellt, dass der Schnittpunkt auf oder außerhalb der Objektoberfläche liegt.

durchgeführt werden, dem letzten Strahl ist die Farbe `background_color` zuzuweisen.

7 Abgabesystem, Referenzplatform

Auf dem Abgabeserver `assignments.icg.tugraz.at` wird jeder git push, der auf dem `submission` Branch getätigt wird automatisch auf dem Testserver ausgeführt.

Die Build-Logs sowie der Output dieser Tests finden sich in der CI / CD - Section auf der linken Seite auf der GitLab-Seite des Repositories.

Eine finale Abgabe muss auf dem `submission` Branch getätigt werden, da wir nur den Code in diesem Branch für die finalen Tests heranziehen!

Die Referenzplatform leitet sich aus folgendem Dockerfile ab:

`https://github.com/icg-teaching-bot/cg1-test-system/blob/master/Dockerfile`

und ist im wesentlichen ein default Ubuntu 18.04 System mit wenigen Ergänzungen.