Universität Basel

Computergrafik 2020

Prof. Dr. Thomas Vetter
Departement Mathematik und Informatik
Spiegelgasse 1, CH – 4051 Basel

Patrick Kahr (patrick.kahr@unibas.ch)
Clemens Büchner (clemens.buechner@unibas.ch)
Moira Zuber (moira.zuber@unibas.ch)

Übungsblatt 5

Ausgabe:	30.04.2020
Abgabe:	28.05.2020
Vorführung:	entfällt

Zu erreichende Punktzahl: 30 (Programmieraufgaben)

Bevor Sie die Übungen lösen, lesen Sie bitte dieses Übungsblatt sowie das Infoblatt aufmerksam durch. Es ist wichtig dass die Abgabe rechtzeitig und wie im Infoblatt beschrieben erfolgt.

In der Vorlesung wurde ein alternatives Verfahren zur Erzeugung von realistischen Bildern vorgestellt: Raytracing bzw. Raycasting. Worin unterscheidet sich dieses Verfahren vom auf den Blättern 1–4 verfolgten Ansatz?
Ziel dieses Aufgabenblattes ist es, einen einfachen Raytracer zu implementieren. Auf diesem Blatt gibt es keine Theoriefragen, da ein sorgfältiges Lösen der Programmieraufgaben die Theorie aus der Vorlesung angemessen vertieft.

Aufgabe 1 – Schnitt Strahl - Dreieck / Kugel — 5 Punkte

Implementieren Sie das `Intersectable` Interface in den Klassen `Sphere` und `Triangle`. Vervollständigen Sie dazu jeweils die `intersect(..)` Methode, sodass diese den Schnittpunkt eines Strahls mit dem jeweiligen Objekt berechnet (siehe Folien). Da ein solcher nicht immer existiert, gibt die Funktion zwecks intuitiver Code-Semantik eine Instanz der Container-Klasse `java.Optional<T>` zurück.

Ergänzen Sie als nächstes die Methode `rayCastScene(..)` in der Klasse `Scene`. Sie soll die Schnittpunkte eines Strahls mit allen Objekten der Szene berechnen und – sofern existent – denjenigen zurückgeben, der am nächsten zum Ursprungspunkt des Strahls liegt.

Hinweis 1: Beachten Sie den zweiten Parameter der Methoden `intersect(..)` und `rayCastScene(..)`: Er beschreibt eine Mindestdistanz, die Schnittpunkte vom Ursprung des Strahls haben müssen.

Hinweis 2: Es bietet sich an, die Schnittpunktberechnung der Klasse `Triangle` mithilfe der Methode `barycentricCoords(..)` zu implementieren.

Benötigte Dateien: `utils.Sphere.java`, `utils.Triangle.java`, `raytracing.Scene.java`

Mögliche Tests: Raycasting 101: Intersections, Depth-Sorted Intersections

Unit-Tests: `ex5.IntersectionTest.java`

Aufgabe 2 – Raycasting — 5 Punkte

Nun werden wir eine Szene mittels einfachen Raycastings entsprechend der Vorlesungsfolien rendern. Hierzu benötigen wir die Klasse `RayTracer`.

Schicken Sie in der Methode `render()` mit Hilfe von `followRay(..)` für jedes Pixel einen Strahl von der Kamera (Position $(0,0,0)^T$) durch die virtuelle Position des Pixels auf der Near-Clipping-Plane, welche durch die `PinholeProjection` gegeben ist. Im Grunde wenden Sie dabei die Transformations- und Projektionspipeline rückwärts an und berechnen auf diese Weise die Rücktransformation der Pixel- zu Welt-Koordinaten.

Untersuchen Sie unter Zuhilfename der Funktion `rayCastScene(..)` aus Aufgabe 1 ob der Strahl ein Objekt in der Szene trifft. Ist dies der Fall, färben Sie das Pixel in der Farbe des getroffenen Objekts ein – andernfalls färben Sie es grau. Die Farbe eines Szenenobjekts ist durch seine Materialeigenschaften gegeben, auf welche Sie mit `SceneObject.getMaterial()` zugreifen können. Sofern eine gerichtete Lichtquelle durch `lightSource` definiert ist, färben Sie das Pixel gemäß dem Lambert-Beleuchtungsmodell ein:

$$\textbf{Farbe}\big(\text{Strahl}^{(\text{Pixel})}\big) = c \cdot \big(\max\{-\langle n, l\rangle, 0\} + a\big)$$

Dabei beschreibt c die Farbe und n die Normale der getroffenen Fläche. Die Lichtrichtung ist durch l gegeben und a beschreibt den ambienten Lichtanteil. Dieser ist im Code durch die Membervariable `ambientLight` definiert.

Verglichen mit den vorhergehenden Übungen wird Ihnen sicherlich die längere Renderzeit aufgefallen sein, die unser `RayTracer` zur Bildberechnung benötigt. Überlegen Sie sich, was das Verfahren so rechenintensiv macht und wieso es nicht so einfach ist, den Renderingprozess zu beschleunigen. Konzeptbedingt verzichtet die Klasse `RayTracer` auf das Rendering eines

Korrespondenzbildes und färbt die Pixel des Framebuffers direkt in ihren finalen Farben ein. Überlegen Sie sich, wieso ein Korrespondenzbild nicht sinnvoll mit Raytracing kombinierbar ist und weshalb insbesondere Deferred-Shading in diesem Kontext keine Vorteile mit sich bringt.

Hinweis 1: Beachten Sie die Orientierung der Kamera bei der Richtungsberechnung der Strahlen. In allen Tests werden die Objekte in die negative z-Richtung verschoben, sodass die Kamera ebenfalls entlang der negativen z-Achse ausgerichtet ist.

Benötigte Dateien: `raytracing.RayTracer.java`

Mögliche Tests: Ray-Cast Scene (Unlit), Ray-Cast Scene

Aufgabe 3 – Raytracing

5 Punkte

Verfolgen Sie nun die Strahlen rekursiv weiter, um auf diese Weise spiegelnde Oberflächen darzustellen. Es werden ideal spiegelnde Oberflächen angenommen. Beziehen Sie Reflexionen in die Simulation ein, indem Sie einen Reflexionsterm zum Beleuchtungsmodell der vorherigen Aufgabe addieren:

$$\begin{aligned} \textbf{Farbe}\big(\text{Strahl}^{(\text{Pixel})}\big) &= c \cdot I_l + r \cdot I_r \\ I_l &= \max\{-\langle n, l\rangle, 0\} + a && \textit{(Lambert-Term)} \\ I_r &= \textbf{Farbe}\big(\text{Strahl}^{(\text{Reflexion})}\big) && \textit{(Reflexionsterm)} \end{aligned}$$

Dabei beschreibt r den Reflexionsgrad der getroffenen Oberfläche und I_r die Farbe des aus der Reflexionsrichtung (gemäß dem Reflexionsgesetz) eintreffenden Lichts. Der Reflexionsgrad ist im `RayTracingMaterial` durch ein `RGBA`-Tupel für alle drei Grundfarben individuell definiert.

Verwenden Sie als initiale Rekursionstiefe `rayTraceDepth` und wenden Sie den Rekursionsschritt nur an, wenn die Membervariable `rayTracingEnabled` auf `true` gesetzt ist. Berechnen Sie die Pixelfarben andernfalls wie in Aufgabe 2.

Benötigte Dateien: `raytracing.RayTracer.java`

Mögliche Tests: Ray-Traced Scene

Aufgabe 4 – Environment Mapping

1 Punkt

Nun betrachten wir den Fall genauer, in dem ein Strahl kein Szenenobjekt trifft. Bisher haben wir die Pixel solcher Strahlen lediglich grau eingefärbt. Nun wollen wir eine konkrete Hintergrundfarbe aus einer Environment-Map – genauer einer Cube-Map – entnehmen.

Überprüfen Sie zunächst, ob die Membervariable `environmentMap` eine konkrete Instanz enthält und entnehmen Sie ihr in diesem Fall eine Farbe für alle Strahlen, die auf kein Objekt treffen. Sie können die Richtung des Strahls (`Ray.direction`) direkt an die `access(..)`-Methode der Environment-Map übergeben. Es wird angenommen, dass dieser Hintergrund unendlich weit entfernt ist, darum kommt es auf den Ursprungspunkt des Strahls nicht an.

Hinweis 1: Falls Sie auf Probleme wie `OutOfMemoryError`-Exceptions oder besonders langsame Programmausführung stossen, versuchen Sie, die Environment-Maps in einer niedrigeren Auflösung zu laden, indem Sie `Ex5TestSuite.USE_HD_CUBEMAPS` auf `false` setzen.

Benötigte Dateien: `raytracing.RayTracer.java`

Mögliche Tests: Environment Mapping

Aufgabe 5 – Schatten — 6 Punkte

Momentan werfen die Objekte in unseren Bildern noch keine Schatten. Für den Realismus der gerenderten Bilder sind glaubwürdige Schatten jedoch unerlässlich.

(a) Harte Schatten (4 Punkte)

Um dem nachzukommen, ändern wir unsere Beleuchtungsgleichung wie folgt ab:

$$\begin{aligned} \textbf{Farbe}(\text{Strahl}^{(\text{Pixel})}) &= v \cdot c \cdot I_l + r \cdot I_r \\ I_l &= \max\{-\langle n, l\rangle, 0\} + a && \textit{(Lambert-Term)} \\ I_r &= \textbf{Farbe}\big(\text{Strahl}^{(\text{Reflexion})}\big) && \textit{(Reflexionsterm)} \end{aligned}$$

Der Vorfaktor v beschreibt, ob der vom $\text{Strahl}^{(\text{Pixel})}$ getroffene Punkt p eines Objekts von der Lichtquelle aus sichbar ist. Um dies zu ermitteln, schicken Sie einen $\text{Strahl}^{(\text{Schatten})}$ von p aus in die Richtung l des einfallenden Lichts. Der Punkt p liegt genau dann im Schatten eines Objekts, wenn der $\text{Strahl}^{(\text{Schatten})}$ auf ein solches trifft.

Implementieren Sie Ihr neues Beleuchtungsmodell so, dass die Schattenberechnung durch `shadowsEnabled` ein- und ausgeschaltet werden kann.

Hinweis 1: Indem Sie die Methode `rayCastSceneAny(..)` implementieren und für die Schattierung nutzen, können Sie den Renderingprozess ein wenig beschleunigen.

Benötigte Dateien: `raytracing.RayTracer.java`, `raytracing.Scene.java`

Mögliche Tests: Shadows

(b) Weiche Schatten (2 Punkte)

Unsere Lichtquellen werden momentan als unendlich klein angenommen und werfen entsprechend harte Schatten. Um weiche Schatten zu simulieren, können wir mehrere, leicht ausgelenkte Strahlen zu verschieden Punkten in der Umgebung der Lichtquelle schicken und die Resultate mitteln.

Sie können die Methode `sampleStandardNormal3D(..)` der Klasse `RandomHelper` verwenden, um die Lichtrichtung zur Schattenberechnung stochastisch auszulenken. Skalieren Sie die Auslenkung mit `shadowSoftness`, um die Simulation unterschiedlich grosser Lichtquellen zu erlauben. Der Effekt soll mit `softShadowsEnabled` ein- und ausschaltbar sein, wobei die Anzahl der verwendeten Strahlen über `softShadowSamples` variierbar sein soll.

Benötigte Dateien: `raytracing.RayTracer.java`

Mögliche Tests: Shadows (Soft)

Aufgabe 6 – Brechung

5 Punkte

Raytracing ist eine sehr universelle Rendering-Technik und erlaubt es uns sogar, transparente Materialien realistisch darzustellen. Wir erweitern unser Modell erneut:

$$\begin{aligned}
\textbf{Farbe}(\text{Strahl}^{(\text{Pixel})}) &= v \cdot c \cdot I_l + r \cdot I_r + t \cdot I_b \\
I_l &= \max\{-\langle n, l\rangle, 0\} + a && \textit{(Lambert-Term)} \\
I_r &= \textbf{Farbe}\big(\text{Strahl}^{(\text{Reflexion})}\big) && \textit{(Reflexionsterm)} \\
I_b &= \textbf{Farbe}\big(\text{Strahl}^{(\text{Brechungsrichtung})}\big) && \textit{(Refraktionsterm)}
\end{aligned}$$

Die Brechungsrichtung bestimmen wir auf Grundlage des Brechungsgesetzes:

$$\sin(\theta_1) \cdot n_1 = \sin(\theta_2) \cdot n_2$$

θ_i ist dabei der Winkel zur Normalen auf der Seite i, und n_i ist der Brechungsindex des Materials auf dieser Seite.

Erweitern Sie ihren Raytracer ein weiteres Mal, um auch die gebrochenen Strahlen zu verfolgen, die auf eine transparente Fläche treffen. Letzteres können Sie mittels `RayTracingMaterial.isTransparent()` überprüfen. Die optische Dichte des transparenten Materials erhalten durch `RayTracingMaterial.getDensity()`. Analog zum Reflexionsgrad ist auch die Transparenz t für alle drei Grundfarben individuell definiert und über `RayTracingMaterial.getTransparency()` abfragbar.

Es ist wichtig, dass Sie unterscheiden, ob ein Strahl in ein Objekt eindringt oder es verlässt. Vergleichen Sie dazu die Richtung des Strahls mit der Normalen der getroffenen Oberfläche. In unseren Tests werden die Normalen immer vom Objekt nach aussen zeigen. Darüberhinaus werden sich transparente Objekte in unseren Tests nie überlappen. Insofern ist einer der beiden Brechungsindizes n_i stets 1.0 und entspricht somit in etwa dem Brechungsindex von Luft.

Hinweis 1: Beachten Sie, dass unter gewissen Bedingungen eine Totalreflexion der Strahlen auftreten kann.

Hinweis 2: Für unsere Gold-Standard-Renderings wurden Reflexionen für alle Strahlen unterbunden, wenn sie ein optisch dichtes Objekt mit Brechungsindex grösser 1 verlassen. Hiervon ausdrücklich ausgenommen sind Effekte der Totalreflexion.

Benötigte Dateien: `raytracing.RayTracer.java`

Mögliche Tests: Refraction

Aufgabe 7 – Realistische Effekte **3 Punkte**

In dieser letzten Aufgabe haben Sie die Auswahl aus verschiedenen Übungen. Um die volle Punktzahl für dieses Blatt zu erhalten, können Sie sich eine davon aussuchen und sie implementieren. Sie erhalten Bonuspunkte, falls Sie mehr als eine davon lösen.

(a) Diffuse Reflexion **(3 Punkte)**

Unsere als ideal modellierten Reflexionen lassen die Oberflächen unserer Objekte noch sehr glatt wirken. In dieser Aufgabe wollen wir unseren Raytracer so erweitern, dass auch raue Oberflächen und diffusere Reflexionen dargestellt werden können.

Dazu berechnen wir den Reflexionsterm nun auf Grundlage von mehreren leicht ausgelenkten Reflexionsstrahlen, anstatt nur eines einzigen. Variieren Sie zur Berechnung jedes Reflexionsstrahls die Richtung der Oberflächennormale. Dadurch simulieren Sie ein Material mit mikroskopisch kleinen Oberflächenunebenheiten. Sie können die Methode `RandomHelper.samplePointOnUnitSphere()` verwenden, um die Richtung der Normalen stochastisch auszulenken. Skalieren Sie die Auslenkungen mit dem Produkt aus `roughReflectionRoughness`, sowie der Materialeigenschaft `RayTracingMaterial.getRoughness()` und einer standardnormalverteilten Zufallsvariable `RandomHelper.sampleStandardNormal1D()`. Die Anzahl der Reflexionsstrahlen können Sie der Membervariablen `roughReflectionSamples` entnehmen. Mitteln Sie zuletzt die Farben aller Reflexionsstrahlen um die Farbe der diffusen Reflexion zu ermitteln. Implementieren Sie den Effekt so, dass er über `roughReflectionsEnabled` ein- und ausgeschaltet werden kann.

Benötigte Dateien: `raytracing.RayTracer.java`

Mögliche Tests: Rough Reflections

(b) Tiefenschärfe **(3 Punkte)**

Mit Raytracing lässt sich sehr einfach ein Tiefenschärfe-Effekt implementieren. Dazu schicken Sie für jeden Pixel mehrere, leicht ausgelenkte Strahlen in die Szene und berechnen den Durchschnitt der resultierenden Farben.

Die Strahlen werden dabei in ihrem Ursprungspunkt so ausgelenkt, dass sich alle Strahlen eines Pixels, in einem bestimmten Abstand – der sogenannten Brennweite – treffen. Punkte, die sich in genau diesem Abstand befinden, werden scharf abgebildet, während die Szene sowohl davor, als auch dahinter zunehmend verwaschen wirkt.

Die Brennweite ist gegeben durch `depthOfFieldFocalLength`, die Anzahl der Strahlen pro Pixel wird durch `depthOfFieldSamples` definiert. Nutzen Sie für die Auslenkung der Strahlen die Methode `sampleStandardNormal3D` der Klasse `RandomHelper` und skalieren Sie diese mit dem Wert der Membervariablen `depthOfField`. Implementieren Sie den Effekt so, dass er über `depthOfFieldEnabled` ein- und ausgeschaltet werden kann.

Benötigte Dateien: `raytracing.RayTracer.java`

Mögliche Tests: Depth of Field

(c) Antialiasing (3 Punkte)

In dieser Aufgabe wollen wir etwas gegen den Treppeneffekt tun, der sich an den Kanten unserer gerenderten Objekte bemerkbar macht.

Wir erreichen dies mit der Supersampling-Technik, die bereits am Anfang der Vorlesung vorgestellt wurde. Das heißt, dass wir für jeden Pixel mehrere Strahlen in die Szene schicken und deren Resultate mitteln. Um den notwendigen Rechenaufwand in Grenzen zu halten, wenden wir diese Technik jedoch nur auf diejenigen Bildpixel an, die zu einer Kante im Bild gehören.

Berechnen Sie zu diesem Zweck zunächst wie gehabt ein Bild mit Ihrem Raytracer und ermitteln Sie in einem zweiten Schritt alle Bildpixel, die Teil einer Kante sind. Vergleichen Sie dazu die Farben benachbarter Pixel komponentenweise. Nehmen Sie an, dass zwei benachbarte Pixel genau dann zu einer Kante gehören, wenn die Summe der absoluten Farbkomponenten ihrer Differenzen einen Schwellenwert überschreitet, der durch die Membervariable `adaptiveSupersamplingThreshold` festgelegt wird.

Wenden Sie nun für jedes Pixel einer Kante die Supersampling-Technik an, indem Sie es in $n \times n$ Subpixel unterteilen und den Raytracing-Algorithmus für jedes Subpixel erneut ausführen. Mitteln Sie die so berechneten Farben und färben Sie das ursprüngliche Pixel in deren Durchschnitt ein.

Der Wert für n ist im Code durch das Memberfeld `adaptiveSupersamplingSamples` gegeben.

Benötigte Dateien: `raytracing.RayTracer.java`

Mögliche Tests: Adaptive Supersampling