Universität Basel

Computergrafik 2021

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

Patrick Kahr (patrick.kahr@unibas.ch)
Renato Farruggio (renato.farruggio@unibas.ch)
Martin Ramm (martin.ramm@unibas.ch)

Übungsblatt 4

Ausgabe: 29.04.2021
Abgabe: 20.05.2021

Zu erreichende Punktzahl: 30 (Programmieraufgaben + Theoriefragen)

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.

Beachten Sie, dass zukünftige Aufgaben auf solche, die mit einem ⋆ gekennzeichnet sind, aufbauen können. Es empfiehlt sich also, die ⋆ Aufgaben besonders sorgfältig zu lösen.

Wir können nun Objekte auf die Bildebene projizieren und zeichnen. Bis jetzt haben wir die Farbe eines Pixels aus den Farben der Eckpunkte interpoliert. Im ersten Teil dieses Übungsblatts implementieren Sie verschiedene Beleuchtungsmodelle und untersuchen deren Unterschiede. Im zweiten Teil implementieren Sie ein Verfahren, um Schatten zu berechnen.

Aufgabe 1 – Perfekter Lambert'scher Strahler — 4 Punkte

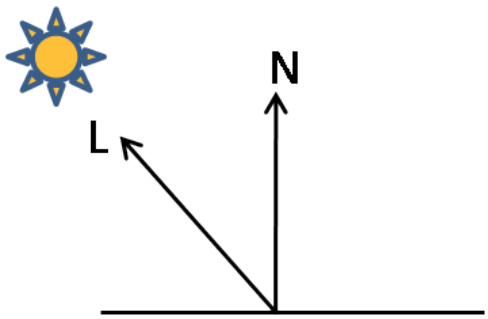
Wie Sie aus der Vorlesung wissen, besteht von einem Objekt abgestrahltes Licht hauptsächlich aus diffus und spekulär/spiegelnd reflektiertem Licht.

Für diese Aufgabe nehmen wir an, dass der Teapot ein perfekter Lambert'scher Strahler ist. Das heisst: Punkte auf der Oberfläche absorbieren Licht und geben dieses in alle Richtungen zu gleichen Teilen ab. Der Anteil des abgestrahlten Lichtes wird durch den Reflektions-Koeffizienten (Albedo) bestimmt. Die Menge des absorbierten und danach abgestrahlten Lichtes hängt von dem Winkel zwischen dem einfallenden Licht und der Oberflächennormalen ab. In dieser Aufgabe nehmen wir an, dass der Teapot das Licht nicht spekulär reflektiert. In Formeln sieht dies so aus:

$$I_{Lambert} = I_{Light} \cdot \langle \hat{\vec{L}}, \hat{\vec{N}} \rangle \cdot albedo$$

Dabei sind:

- I_{Light}: `RGBA lightSource.color`
- *albedo*: `double MATERIAL_ALBEDO`
- $\langle ., . \rangle$ ist ein Skalarprodukt
- $\vec{N}, \vec{L}$: siehe Abbildung
- $\hat{\vec{V}}$: ein normalisierter Vektor



Wir nehmen an, dass die Lichtquelle eine Punkt-Lichtquelle ist. $\vec{L}$ muss also aus den Positionen der Lichtquelle und des betrachteten Punktes berechnet werden.

Implementieren Sie die Methode `colorizePixel(..)` der Klasse `LambertMeshRenderer`, welche die Pixel als perfekte Lambert'sche Strahler einfärbt.

Hinweis 1: Nur Flächen, welche zur Lichtquelle hin zeigen, strahlen auch Licht dieser Quelle aus.

Hinweis 2: Die Normalen-Informationen der Vertices können aus dem Array `tni` entnommen werden, analog zu `tvi` in Aufgabenblatt 3.

Benötigte Dateien: `renderer.LambertMeshRenderer.java`

Mögliche Tests: Lambert Renderer

Unit-Tests: `ex4.LambertTest.java`

Aufgabe 2 – Phong'sches Reflexionsmodell

4 Punkte

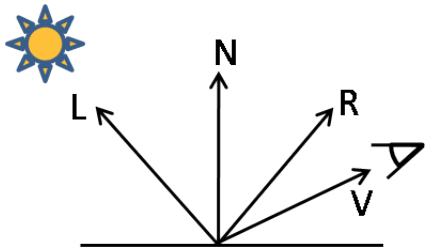
Wenn Sie bei dem LambertRenderer-Test den Teapot von hinten betrachten, stellen Sie fest, dass er im schwarzen Hintergrund verschwindet. In der realen Welt sind Oberflächen, welche nicht direkt von einer Lichtquelle beleuchtet werden meistens nie ganz schwarz. Dies, weil sie von Licht beleuchtet werden, welches von anderen Oberflächen reflektiert wurde. Das Phong'sche Beleuchtungsmodell approximiert dieses Phänomen, indem alle Oberflächen von einer *ambienten* Lichtquelle gleich stark beleuchtet werden. Das Phong-Modell simuliert auch spekulär reflektiertes Licht, indem es den Winkelunterschied zwischen dem reflektierten Strahl und dem Auge mit einbezieht.

Bei uns bestehen Phong'sche Lichtquellen aus deren Position, der Farbe und einer "ambienten Lichtquelle".

In Formeln sieht dies folgendermassen aus:

$$I_{Phong} = r_A \cdot I_A + r_D \cdot I_C \cdot \langle \hat{\vec{L}}, \hat{\vec{N}} \rangle + r_S \cdot I_C \cdot \langle \hat{\vec{R}}, \hat{\vec{V}} \rangle^m$$

- I_A: `RGBA lightSource.ambient`
- I_C: `RGBA lightSource.color`
- r_A: `RGBA material.ambient`
- r_D: `RGBA material.diffuseReflectance`
- r_S: `RGBA material.specularReflectance`
- m: `double material.shininess`
- $\langle ., . \rangle$ ist ein Skalarprodukt
- $\hat{\vec{V}}$: ein normalisierter Vektor
- $\vec{N}, \vec{L}, \vec{R}, \vec{V}$: siehe Abbildung



Implementieren Sie die Methode `colorize(..)` der Klasse `PhongMeshRenderer`, welche Meshes nach dem beschriebenen Phong-Model einfärbt.

Hinweis 1: Welche Werte der Skalarprodukte sind sinnvoll und welche sollten besser nicht verwendet werden?

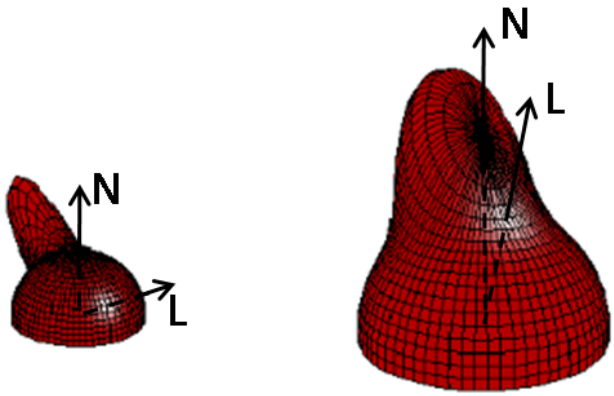
Benötigte Dateien: `renderer.PhongMeshRenderer.java`

Mögliche Tests: Phong Reflectance Model

Unit-Tests: `ex4.PhongTest.java`

Aufgabe 3 – Bidirectional Reflectance Distribution Function (BRDF) und Lambert'sche Strahler ⋆ **3 Punkte**

Wir distanzieren uns etwas vom Phong'schen Reflexionsmodell und betrachten Oberflächen von Materialien etwas abstrakter. Anstelle verschiedener Verhaltensweisen einer Oberfläche für (ambiente), diffuse und spekuläre Reflektion, versuchen wir das Verhalten der Obefläche mit einer Funktion zu beschreiben. Diese Funktion heisst Bidirectioanl Reflectance Distribution Function (BRDF). Die BRDF ist eine Eigenschaft der Oberfläche und nimmt als Parameter die Richtung zur einfallenden Lichtquelle, die Normalenrichtung und die Richtung zur Kamera entgegen. Das Resultat der BRDF ist die radiance/Abstrahlung unter den gegebenen Parametern.



BRDF einer Oberfläche bei fixierter Normalen- und Lichtrichtung. Der Funktionswert represäntiert die Abstrahlung in verschiedene (Blick-) Richtungen.

(a) BRDF des Lambert'schen Strahlers **(1 Punkt)**

Implementiern Sie die Funktion `getRadiance(..)` der Klasse `LabertBrdf`. Der Rückgabewert dieser Funktion wird in der bereits implementierten Methode der Klasse `Brdf` mit dem $\langle \hat{\vec{L}}, \hat{\vec{N}} \rangle$ Produkt gewichtet und elementweise mit der Farbe der Lichtquelle multipliziert.

Hinweis 1: Die BRDF eines Lambert'schen Strahlers ist konstant (albedo).

Benötigte Dateien: `reflectance.LambertBrdf.java`

(b) Reflectance Mesh Renderer **(2 Punkte)**

Implementieren Sie die methode `shade(..)` des `ReflectanceMeshRenderer`, welche ein Mesh mit mehreren Lichtquellen und mehreren BRDF's zeichnen kann. Rufen sie dazu für jede Lichtquelle und jede BRDF des Materials `getRadiance(..)` auf und kombinieren Sie die Rückgabewerte.

Hinweis 1: Berechnungen von Richtungen werden im Objektraum vorgenommen.

Benötigte Dateien: `renderer.ReflectanceMeshRenderer.java`

Mögliche Tests: Reflectance Renderer: Lambert, Reflectance Renderer: Lambert with multiple lightSources

Unit-Tests: `ex4.ReflectanceTest.java`

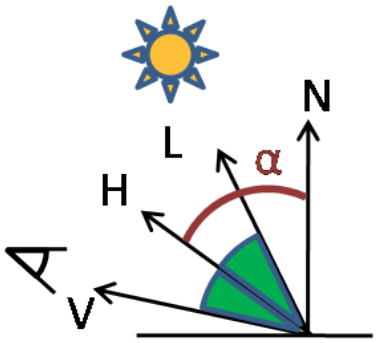
Aufgabe 4 – Cook-Torrance **3 Punkte**

Das Cook-Torrance Beleuchtungsmodell versucht die spekuläre Reflektion realistischer zu modellieren. Das lokale Beleuchtungsmodell modelliert Oberflächen durch viele kleine *Facetten*, welche unterschiedliche Normalenrichtungen haben. Dadurch werden Brechungsverhältnisse, Rauheit und Selbstabschattung berücksichtigt.

Das Cook-Torrance Beleuchtungsmodell verwendet dazu den sogenannten *Half Angle Vector* H:

$$H = \frac{L+V}{|L+V|}$$

α beschreibt den Winkel zwischen H und der Normalen N.



Das Cook-Torrance Modell besteht hauptsächlich aus drei Teilen:

1. Rauheit Es wird angenommen, dass die Oberfläche aus V-förmigen Vertiefungen besteht. Der proportionale Flächenanteil in Richtung α ist gegeben durch den Parameter m, welcher die Rauheit bestimmt:

$$D = \frac{1}{4\,m^2\cos^4(\alpha)}\exp\left(-\frac{1-\cos^2(\alpha)}{\cos^2(\alpha)\,m^2}\right) = \frac{1}{4\,m^2\langle N,H\rangle^4}\exp\left(-\frac{1-\langle N,H\rangle^2}{\langle N,H\rangle^2\,m^2}\right)$$

2. Geometrische Attenuation (Abschwächung) Die Selbstabschattung wird in drei Fälle unterteilt, um den reflektierenden Anteil G zu bestimmen.

$$\begin{aligned} G_1 &= 1 \\ G_2 &= \frac{2\,|\langle N,H\rangle||\langle N,V\rangle|}{|\langle V,H\rangle|} \\ G_3 &= \frac{2\,|\langle N,H\rangle||\langle N,L\rangle|}{|\langle V,H\rangle|} \\ G &= min\{G_1,G_2,G_3\} \end{aligned}$$

3. Fresnel Dieser Term approximiert den Effekt, dass die Reflektion zunimmt, je weiter die Oberfläche sich von der Blickrichtung wegdreht. (Dieser Effekt kann einfach selbst getestet werden, indem Sie zum Beispiel ein Blatt Papier so zwischen Auge und Lichtquelle halten, dass Sie beinahe nur noch den Rand des Papiers sehen.)

Dies approximieren wir durch:

$$F = r_0 + (1 - r_0)(1 - |\langle H, V \rangle|)^5$$

Die Cook-Torrance BRDF setzt sich schliesslich zusammen zu:

$$I_{out} = albedo \cdot \frac{F \cdot D \cdot G}{\langle V, N \rangle}$$

Implementieren Sie das beschriebene Cook-Torrance Modell in der Methode `getRadiance(..)` der Klasse `CookTorrance`.

Hinweis 1: Das Skalarprodukt zweier Vektoren der Länge 1 entspricht dem Cosinus des eingeschlossenen Winkels.

Benötigte Dateien: `reflectance.CookTorrance.java`

Mögliche Tests: Cook-Torrance: Lambert + Cook-Torrance

Unit-Tests: `ex4.CookTorranceTest.java`

Aufgabe 5 – Oren-Nayar **3 Punkte**

Das Oren-Nayar Reflektionsmodell versucht, im Gegensatz zu Cook-Torrance, die diffuse Reflektion realistischer zu modellieren.

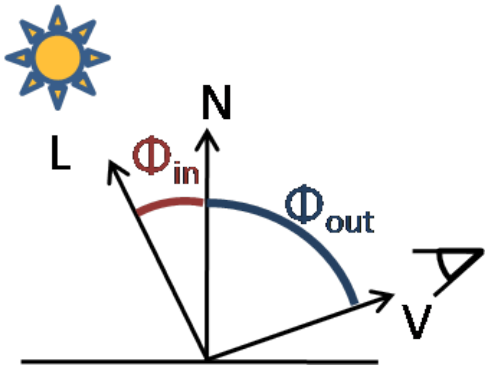
Das Modell bedient sich auch der Vorstellung, dass Oberflächen durch V-förmige Vertiefungen sogenannte Facetten haben. Die Neigungen dieser Vertiefungen werden als Gauss-Verteilt angenommen und haben eine Varianz von σ^2. Im Oren-Nayar Modell ist jede dieser Facetten ein Lambert'scher Strahler.

Durch physikalische Überlegungen gelangt man zur BRDF des Oren-Nayar Reflektionsmodell:

$$I_{out} = albedo \cdot \left[A + \left(B \cdot max\big(0, \cos(\phi_{in} - \phi_{out})\big) \cdot \sin\alpha \cdot \tan\beta \right) \right]$$

Wobei:

$$A = 1 - 0.5 \frac{\sigma^2}{\sigma^2 + 0.33}$$
$$B = 0.45 \frac{\sigma^2}{\sigma^2 + 0.09}$$
$$\alpha = max(\phi_{in}, \phi_{out})$$
$$\beta = min(\phi_{in}, \phi_{out})$$



Implementieren Sie das eben beschriebene Modell für die diffuse Reflektion in der Methode `getRadiance(..)` der Klasse `OrenNayar`.

Hinweis 1: Das Skalarprodukt zweier Vektoren der Länge 1 entspricht dem Cosinus des eingeschlossenen Winkels.

Hinweis 2: Es ist nicht nötig, einen Winkel explizit zu berechnen. Sie können aus dem Cosinus direkt den Sinus berechnen.

Hinweis 3: Der Tangens kann mit $\frac{\sin\beta}{\cos\beta+0.0001}$ berechnet werden, um Divisionen durch null zu vermeiden.

Benötigte Dateien: `reflectance.OrenNayar.java`

Mögliche Tests: OrenNayar

Unit-Tests: `ex4.OrenNayarTest.java`

Aufgabe 6 – Schatten

4 Punkte

Es gibt verschiedene Algorithmen zur Berechnung von Schatten. Wir schlagen vor, dass Sie Shadow Maps verwenden. Wenn Sie einen anderen Algorithmus implementieren wollen, können Sie dies gerne tun (z.B. Shadow Volumes).

Die Grundüberlegung der Shadow Maps: Rendert man die Szene von der Lichtquelle aus, sind Flächen, welche aus der Perspektive der Lichtquelle nicht sichtbar sind, im Schatten.

Um die Viewmatrix aus der Sicht der Lichtquelle zu erzeugen, gehen Sie wie folgt vor.

$$M = \left(\begin{array}{c|c} \mathbf{Q} & \mathbf{d} \\ \hline \mathbf{0}^T & 1 \end{array}\right)$$

Wobei $\mathbf{Q}$ eine 3×3 Matrix ist, welche das Koordinatensystem aus Richtung der Lichtquelle aufspannt:

$$Q = \begin{pmatrix} \vec{x} \\ \vec{y} \\ \vec{z} \end{pmatrix}$$

Wenn die Position der Lichtquelle durch L gegeben ist, können wir das Koordinatensystem aus der Richtung der Lichtquelle wie folgt generieren:

$$\vec{z} = \frac{L}{|L|}$$
$$\vec{y} = \frac{\vec{z} \times \vec{x_0}}{|\vec{z} \times \vec{x_0}|} \mid \vec{x_0} = (1,0,0)^T$$
$$\vec{x} = \frac{\vec{y} \times \vec{z}}{|\vec{y} \times \vec{z}|}$$

(a) View Matrix (2 Punkte)

Implementieren Sie die Methode `getViewMatrixOfLightSource(..)` der Klasse `PinholeProjection`, welche die eben beschriebene ViewMatrix erzeugt.

Hinweis 1: In unserem Setting befindet sich die Kamera im Ursprung. Um nun die Viewmatrix aus Sicht der Lichtquelle zu erhalten müssen wir also eine Rotation ausführen (Matrix Q) und anschliessend noch eine Translation (Vektor d) an die Position der Lichtquelle durchführen. Hierbei muss der Vektor d natürlich im neuen rotierten Koordinatensystem ausgedrückt werden.

Benötigte Dateien: `projection.PinholeProjection.java`

Mögliche Tests: Phong: Direction of LightSource, Reflectance: Direction of LightSource

Unit-Tests: `ex4.ShadowViewMatrixTest.java`

(b) Shadow Map (1 Punkt)

Implementieren Sie die Methode `generateShadowMap(..)` der Klasse `Occlusion`, die die Viewmatrix aus der Richtung der Lichtquelle in `shadowProjection` und das damit erzeugte Korrespondenzfeld in `shadowMap` speichert.

Benötigte Dateien: `occlusion.Occlusion.java`

Mögliche Tests: Phong: ShadowMap, Reflectance: ShadowMap

(c) Shadows (1 Punkt)

Ergänzen sie die Methode `inShadow(..)` der Klasse `Occlusion`, welche einen double zwischen 0 und 1 zurückgibt. Der Rückgabewert soll 0 sein, falls der zu prüfende Punkt von der Lichtquelle nicht beleuchtet wird, also im Schatten liegt. Es soll 1 zurückgegeben werden, falls der Punkt von der Lichtquelle "gesehen" wird.

Ändern Sie ihren `PhongMeshRenderer` so, dass die Berechnung der Farbe berücksichtigt, ob ein Schatten gezeichnet werden soll oder nicht. Falls Schatten gerendert werden sollen (`shadows` wahr ist), rufen Sie die Methode `inShadow(..)` der Member-Variable `shadowSystem` auf und verwenden den Rückgabewert entsprechend. Verändern Sie nun ebenfalls die Klasse `ReflectanceMeshRenderer` um Schatten rendern zu können.

Hinweis 1: Wir blicken in die negative z-Richtung.

Hinweis 2: Beziehen sie die Member-Variable `shadows` im Renderer mit ein. Greifen Sie nur auf das shadowSystem zu, falls Schatten aktiviert sind und shadowSystem initialisiert ist.

Hinweis 3: Seien Sie vorsichtig, wo sie die ShadowMap generieren lassen. Für jede Lichtquelle wird bestenfalls nur einmal eine ShadowMap erzeugt.

Hinweis 4: Beziehen Sie im Code die Membervariable `shadowBias` mit ein. Diese kann durch den Graphischen Test verändert werden.

Benötigte Dateien: `occlusion.Occlusion.java`, `renderer.PhongMeshRenderer.java`, `renderer.ReflectanceMeshRenderer.java`

Mögliche Tests: Phong: Shadow Renderer, Reflectance: Shadow Renderer

Aufgabe 7 – Weiche Schatten — 3 Punkte

Betrachten Sie die von Ihnen gerechneten Schatten aus der vorigen Aufgabe. Dabei wird Ihnen auffallen, dass der Übergang zwischen Schatten und beleuchteten Bereichen unnatürlich hart ist. Überlegen Sie sich, weshalb dies in der Natur nicht geschieht.

Ziel dieser Aufgabe ist es, weiche Schatten zu berechnen.

Eine Methode dafür ist Percentage Closer Filtering (PCF, siehe Vorlesung): Nachdem man die Objektkoordinaten auf die Shadow Map projiziert hat, testet man den Z-Buffer auch für die umgebenden Pixel (z.B. 3×3). Diese `boolean` Werte können dann miteinander verrechnet werden. Wenn Sie wollen, können Sie auch eine andere Methode implementieren, um Soft Shadows zu rendern (z.B. Light Jittering).

Benötigte Dateien: `occlusion.Occlusion.java`

Mögliche Tests: Soft Shadow: Phong Renderer, Soft Shadow: Reflectance Renderer, Soft Shadow: Cook-Torrance + OrenNayar

Aufgabe 8 – Theoriefragen — 6 Punkte

Sie haben in den Programmieraufgaben auf diesem Blatt den 'shadow mapping' Ansatz für das Berechnen von Schatten angewendet. Dabei kam der sogenante 'schadow bias' zum Einsatz. Erklären Sie wie 'shadow mapping' Verfahren funktionieren und warum es diesen 'bias' Term braucht. Wie wird dieser Term gewählt? Rendern Sie ein paar Beispiele mit Ihrer eigenen Implementierung.