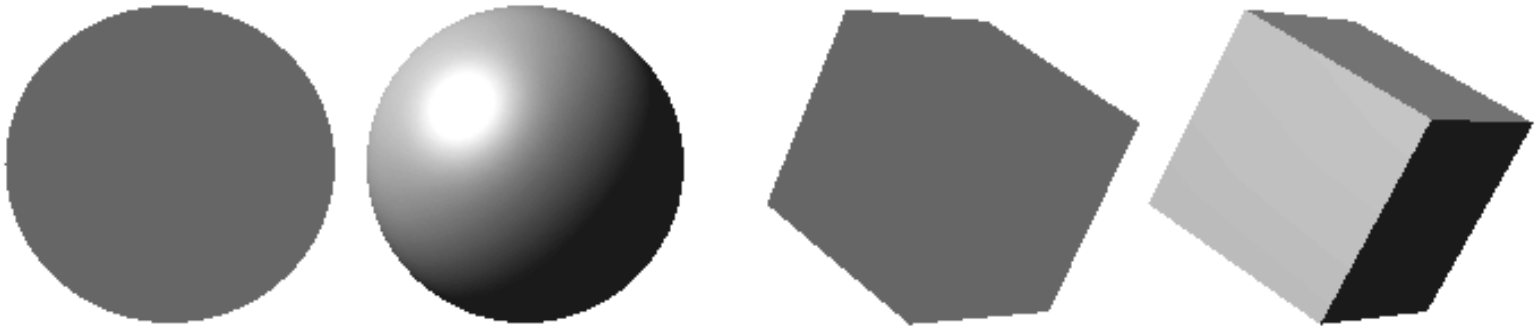
Beleuchtung/Schattierung



Objekte mit und ohne Beleuchtungseffekte

Schattierung/Shading: Farb- oder Helligkeitsmodifikation einer Oberfläche durch Beleuchtungseffekte

Beleuchtungsmodelle

Fotorealistische Darstellungen erfordern die Berücksichtigung von Beleuchtungseffekten (Reflexion, Schattierung, Schatten).

Bei den folgenden Betrachtungen sind alle Überlegungen einzeln auf die drei Farben des RGB-Modells anzuwenden, um Intensitätsberechnungen durchzuführen.

Aus der physikalischen Sicht müssten alle Intensitätsberechnungen für jede Wellenlänge separat durchgeführt werden.

Lichtquellen

- **Streulicht** oder **ambientes Licht** ist Licht, das nicht direkt von einer spezifischen Lichtquelle stammt oder aus einer Richtung kommt, sondern durch (Mehrfach-)Reflexion an (verschiedenen) Oberflächen entsteht.

- **Parallel einfallendes** oder **direktionales Licht** hat eine Farbe und eine Richtung, aus der es kommt. Die Lichtstrahlen verlaufen parallel. („unendlich“ weit entfernte Lichtquelle)

Lichtquellen

- Eine Lampe entspricht einer **punktförmigen Lichtquelle**.
  - Besitzt Farbe und Position in Raum.
  - Die Intensität des Lichts nimmt mit der Entfernung von der Lichtquelle ab. (**Dämpfung/Attenuation**)

    Quadratische Abnahme der Intensität mit der Entfernung

Lichtquellen

theoretisch: Intensität mit dem Faktor $1/d^2$ multiplizieren, wenn das Licht auf ein Objekt in der Entfernung d trifft.

drastische Effekte: $d \to 0 \Rightarrow$ Intensität $\to \infty$

Modifiziertes Modell

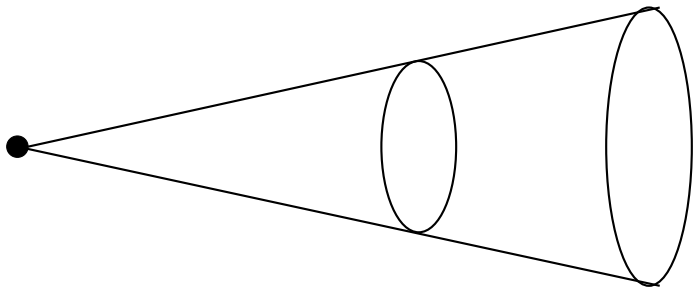
$$f_{\text{att}} = \min\left\{\frac{1}{c_1 + c_2 d + c_3 d^2}, 1\right\}$$

Linearer Term kann **atmosphärischen Dämpfung** modellieren.

Lichtquellen

- **Scheinwerfer** oder **Spotlights**: punktförmige Lichtquelle, deren Licht nur in einer Richtung einem Lichtkegel abgegeben wird.

  Ebenfalls quadratische Abnahme der Intensität mit der Entfernung

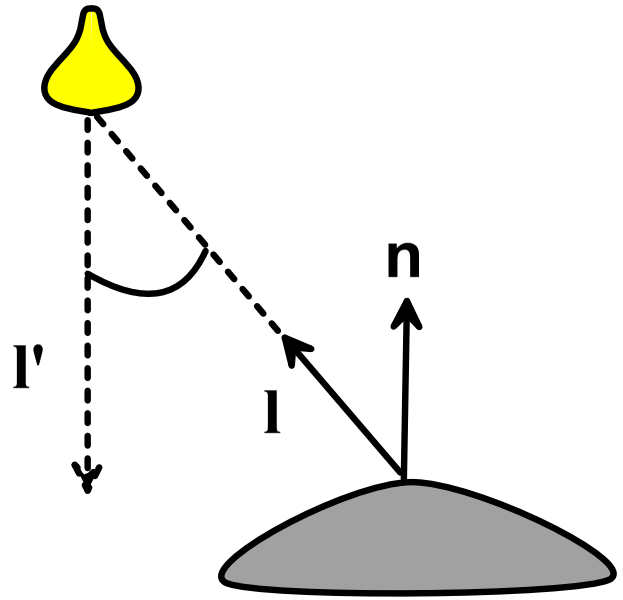


Lichtquellen

Abnahme der Intensität zum Rand des Lichtkegels:

Gesamtintensität:

$$I = I_S \cdot f_{\text{att}} \cdot (\cos\gamma)^p = I_S \cdot f_{\text{att}} \cdot \left(-\mathbf{l'}^{\top} \cdot \mathbf{l}\right)^p$$



Java 3D: Beleuchtung

Streulicht: Mittels

```
AmbientLight ambLight =
            new AmbientLight(ambColour);

ambLight.setInfluencingBounds(bounds);
```

wird Streulicht in der Farbe (`Color3f`) `ambColour` erzeugt.

Wie bei den `Interpolatoren` bei der Animation muss auch dem Licht (nicht nur dem ambienten Licht, sondern jeder Art von Licht) ein Wirkungsbereich zugeordnet werden.

Dies erfolgt mittels der Methode `setInfluencingBounds(bounds)`. Dabei ist `bounds` beispielsweise eine `BoundingSphere`.

Java 3D: Beleuchtung

direktionales Licht: Mittels

```
DirectionalLight dirLight =
  new DirectionalLight(lightColour,
                       lightDir);
```

wird Licht der Farbe `lightColour` einer unendlich weit entfernten Lichtquelle erzeugt.

Die Richtung, in die das Licht strahlt, wird durch den `Vector3f lightDir` festgelegt.

punktförmige Lichtquelle: Mittels

```
PointLight pLight =
        new PointLight(lightColour,
                       location,
                       attenuation);
```

wird eine punktförmige Lichtquelle mit der Lichtfarbe `lightColour` im Punkt (`Point3f`) `location` definiert.

`attenuation` ist ein Objekt der Klasse `Point3f`, dessen drei Werte die Koeffizienten des quadratischen Nennerpolynoms für die Lichtdämpfung angeben.

Java 3D: Beleuchtung

Spot Light: Eine punktförmige Lichtquelle strahlt in alle Richtungen gleichmäßig.

Die Klasse `SpotLight` erweitert die Klasse `PointLight`.

Einem `SpotLight` wird eine Richtung, in die es scheint, und ein Lichtkegel zugeordnet:

```
SpotLight spLight =
            new SpotLight(lightColour,
                          location,
                          attenuation,
                          direction,
                          angle,
                          concentration);
```

Java 3D: Beleuchtung

Die ersten drei Parameter haben dieselbe Bedeutung wie beim `PointLight`.

Der `Vector3f direction` gibt die Richtung an, in die das Licht strahlt.

Der `float`-Wert `angle` definiert den (halben) Öffnungswinkel des Lichtkegels.

Der `float`-Wert `concentration` zwischen 0 und 120 gibt an, wie stark die Intensität des Lichtes zum Rand des Kegels abfällt.

Java 3D: Beleuchtung

- Alle Lichtquellen sollten der `BranchGroup` `bgLight` mittels `addChild(...)` zugeordnet werden.
- `bgLight` muss mittels `addBranchGraph(...)` dem `SimpleUniverse` hinzugefügt werden.

Java 3D: Beleuchtung

bewegte Lichtquellen:

- ordne die Lichtquelle nicht direkt der `BranchGroup bgLight` zu
- erzeuge eine Transformationsgruppe für die Bewegung
- ordne die Lichtquelle der Transformationsgruppe zu
- ordne die Transformationsgruppe `bgLight` zu

(vgl. `MovingLight.java`)

Java 3D: Beleuchtung

- Lichtquellen sind selbst unsichtbar.
- Um eine Lampe zu modellieren und nicht nur ihr Licht zu modellieren, muss ein entsprechendes Objekt in der Szene erzeugt werden.
- Die zugehörige Lichtquelle kann auch `theScene` statt `bgLight` zugeordnet werden.

Selbstleuchtende Objekte

Objekt mit eigener Leuchtkraft:

Beleuchtungsgleichung: $I = k_i$

eigentlich:

$$I^{(\text{rot})} = k_i^{(\text{rot})} \qquad I^{(\text{grün})} = k_i^{(\text{grün})} \qquad I^{(\text{blau})} = k_i^{(\text{blau})}$$

Selbstleuchtende Objekte werden üblicherweise nicht als Lichtquellen angesehen, die andere Objekte erhellen.

Reflexion

Intensität in einem Pixel durch Beleuchtung mit einer Lichtquelle:

$$I = I_{\text{Lichtquelle}} \cdot f_{\text{Pixel}}$$

- $I_{\text{Lichtquelle}}$: Intensität des Lichts der betrachteten Lichtquelle
- f_{Pixel}: Faktor, der von verschiedenen Parametern abhängt: Farbe und Art der Oberfläche, Entfernung der Lichtquelle im Falle von Dämpfung und dem Winkel, in dem das Licht im dem betrachteten Pixel auf die Oberfläche trifft.

Reflexion

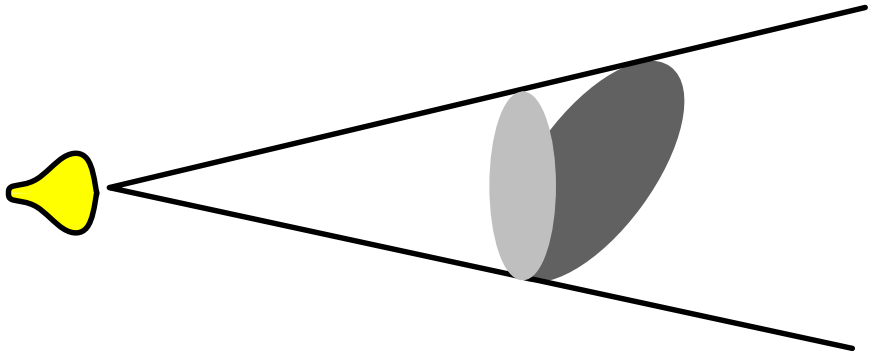
Beleuchtungsgleichung bei ambientem Licht:

$$I = k_a \cdot I_S$$

k_a: Reflexionskoeffizient der Oberfläche für Streulicht

Diffuse Reflexion

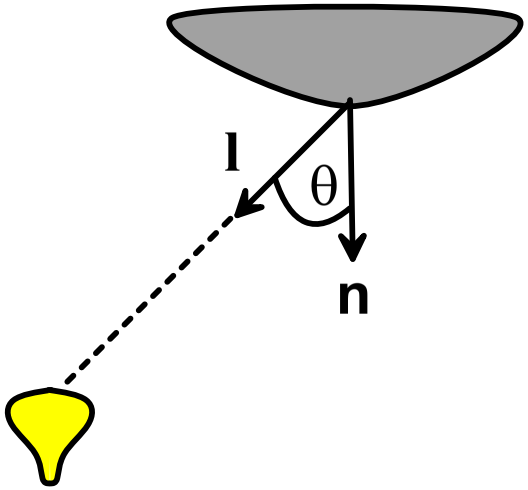
Lichtintensität in Abhängigkeit vom Auftreffwinkel



Lambertsches Reflexionsgesetz:

$$I = I_L \cdot k_d \cdot \cos\theta$$

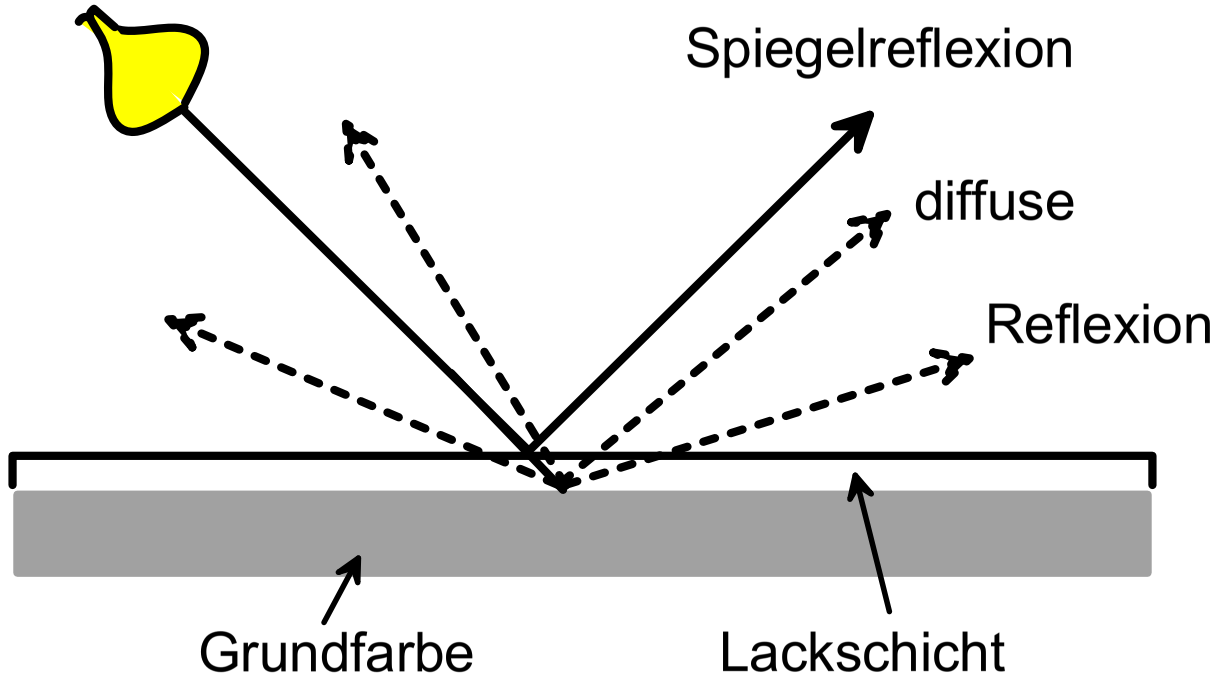
Diffuse Reflexion



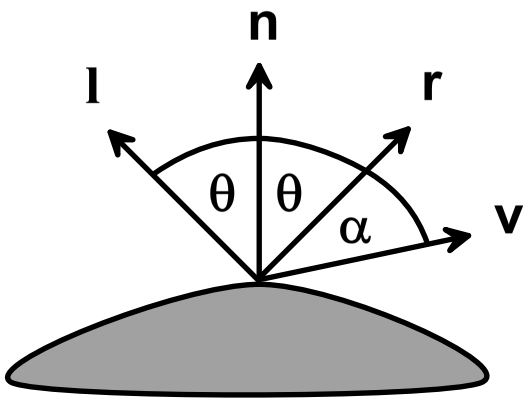
bei normierten Vektoren:

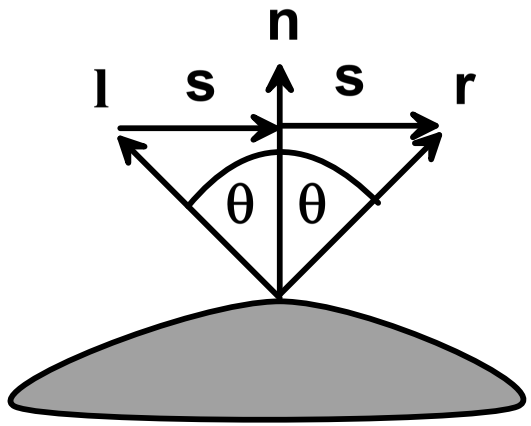
$$I = I_L \cdot k_d \cdot (\mathbf{n}^\top \cdot \mathbf{l})$$

Spiegelreflexion



Spiegelreflexion





$$\mathbf{r} = \mathbf{n} \cdot \cos(\theta) + \mathbf{s} \qquad \mathbf{s} = \mathbf{n} \cdot \cos(\theta) - \mathbf{l}$$

$$\mathbf{r} = 2 \cdot \mathbf{n} \cdot \cos(\theta) - \mathbf{l} \qquad \mathbf{r} = 2 \cdot \mathbf{n}(\mathbf{n}^\top \cdot \mathbf{l}) - \mathbf{l}$$

Vorausetzung: $0° \leq \theta < 90°$, d.h. $\mathbf{n}^\top \cdot \mathbf{l} > 0$

Spiegelreflexion

idealer Spiegel: Spiegelreflexion nur in Richtung von $\mathbf{r}$

nicht-ideal: Spiegelreflexion um $\mathbf{r}$ herum

Beleuchtungsmodell von Phong:

$$I = I_L \cdot W(\theta) \cdot (\cos(\alpha))^n$$

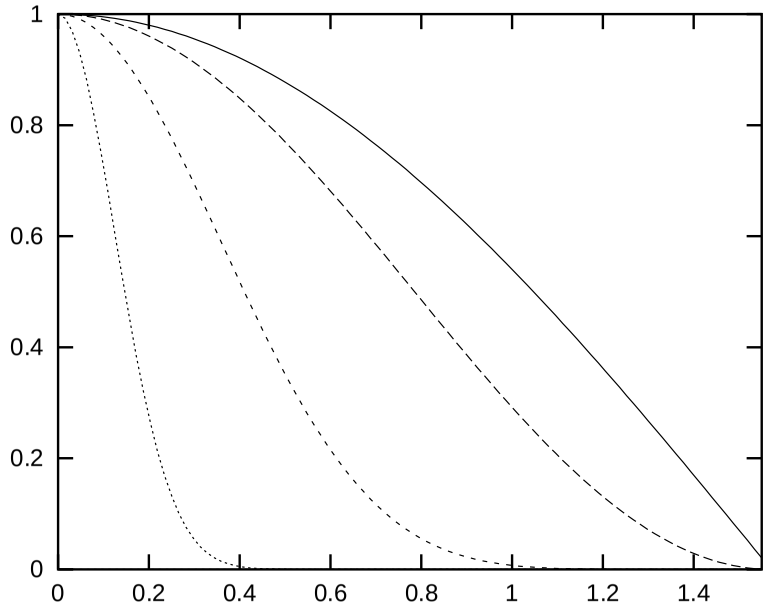
$0 \leq W(\theta) \leq 1$: Anteil des Lichts, der Spiegelreflexion beim Einfallswinkel θ unterliegt.

n: **Spiegelreflexionsexponent,**

$n \to \infty$: idealer Spiegel

Beleuchtungsmodell von Phong

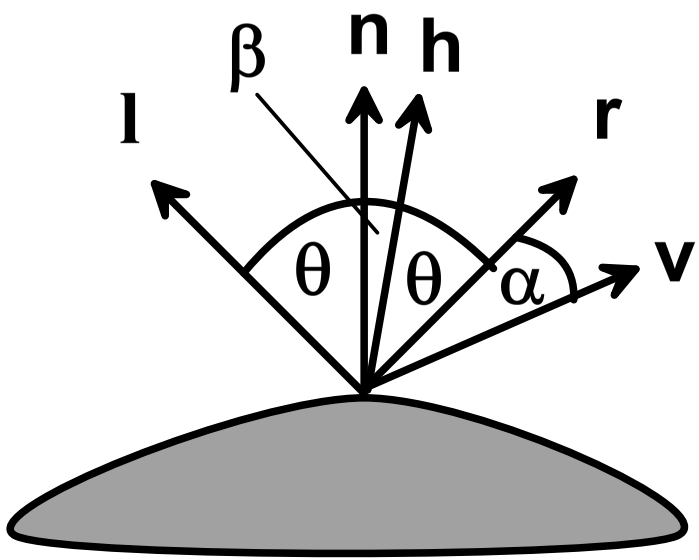
$$I = I_L \cdot W(\theta) \cdot (\mathbf{r}^\top \cdot \mathbf{v})^n$$



$$(\cos\alpha)^{64}, (\cos\alpha)^{8}, (\cos\alpha)^{2}, \cos\alpha$$

Mod. Phong-Beleuchtungsmodell

Alternatives Maß für die Stärke der Spiegelreflexion: Abweichung des Normalenvektors $\mathbf{n}$ von der Winkelhalbierenden $\mathbf{h}$



Mod. Phong-Beleuchtungsmodell

modifiziertes Phongsches Modell: Verwende anstelle $\cos\alpha$ den Term $\cos\beta$:

$$\cos\beta \;=\; \mathbf{n}^{\top}\cdot\mathbf{h}$$

mit

$$\mathbf{h} \;=\; \frac{\mathbf{l}+\mathbf{v}}{\|\,\mathbf{l}+\mathbf{v}\,\|}$$

Vorteil: $\mathbf{h}$ bleibt im Gegensatz zu $\mathbf{r}$ bei direktionalem Licht und unendlich weit entferntem Beobachter (Parallelprojektion) konstant.

Mehrere Lichtquellen

$$I = I_{\text{Eigenleuchten}} + I_{\text{Streulicht}} \cdot k_a + \sum_j I_j \cdot f_{\text{att}} \cdot g_{\text{Kegel}} \cdot \left(k_d \cdot (\mathbf{n}^\top \cdot \mathbf{l}_j) + k_{\text{sr}} \cdot (\mathbf{r}_j^\top \cdot \mathbf{v})^n \right)$$

Deferred Shading

großer Rechenaufwand beim z-Puffer-Algorithmus bei mehreren Lichtquellen

aufwendiges Rendering von Objekten, die später durch andere überschrieben (verdeckt) werden

Deferred Shading:

- Führe zunächst einen Durchlauf des z-Puffer-Algorithmus durch, in dem nur der z-Puffer berechnet wird.
- Im zweiten Durchlauf werden dann nur die Objekte gerendert, deren z-Koordinate im z-Puffer eingetragen ist.

Java 3D: Oberflächeneigenschaften

Jedem Objekt wird eine `Appearance` zugeordnet, die alle grundlegenden Informationen über die Erscheinung und Darstellung der Oberfläche enthält.

Ein wesentliches Attribut einer `Appearance` ist das `Material`.

Damit werden Farbe und Reflexionseigenschaften der Oberfläche des Objektes festgelegt.

Java 3D: Oberflächeneigenschaften

Konstruktor für `Material`:

```
Material ma =
            new Material(ambientColour,
                         emissiveColour,
                         diffuseColour,
                         specularColour,
                         shininessValue);
```

- `ambientColour` ist eine Farbe, die angibt, in welcher Farbe die Oberfläche Streulicht reflektiert.

Java 3D: Oberflächeneigenschaften

- `emissiveColour` ist eine Farbe, die angibt, in welcher Farbe die Oberfläche selbst leuchtet.

  Das Objekt erscheint in dieser Farbe auch bei Dunkelheit. Es leuchtet aber keine anderen Objekte an, fungiert also nicht als Lichtquelle für die restliche Szene.
- `diffuseColour` ist die Farbe der Oberfläche bei diffuser Reflexion.
- `specularColour` ist die Farbe der Oberfläche bei Spiegelreflexion.

Java 3D: Oberflächeneigenschaften

- `shininessValue` ist ein `float`-Wert zwischen 1 und 128, der angibt, wie glänzend die Oberfläche ist (je größer, desto glänzender).

  Mit dem `shininessValue` wird der Spiegelreflexionsexponent im Phongschen Beleuchtungsmodell bestimmt.

```
Appearance app = new Appearance();
```

erzeugt eine `Appearance`.

Java 3D: Oberflächeneigenschaften

Mit `app.setMaterial(ma);` werden der `Appearance` die Oberflächeneigenschaften zugewiesen, die im `Material ma` definiert wurden.

Jedem elementaren geometrischen Objekt (`Box, Sphere, Cylinder, Cone`) oder jedem `Shape3D` kann im Konstruktor oder mittels der Methode `setAppearance(app)` eine gewünschte `Appearance app` zugeordnet werden.

(vgl. `LightingExample.java` und `LightingExample2.java`)

Schattierung

Gekrümmte Oberflächen werden durch ebene Polygone angenähert.

Ebene Polygone haben denselben Normalenvektor in jedem Punkt.

Konstante Schattierung:

Es wird ein Punkt eines ebenen Polygons gewählt und für diesen die Intensität berechnet. Dieser Wert wird für die gesamte Polygonfläche verwendet.

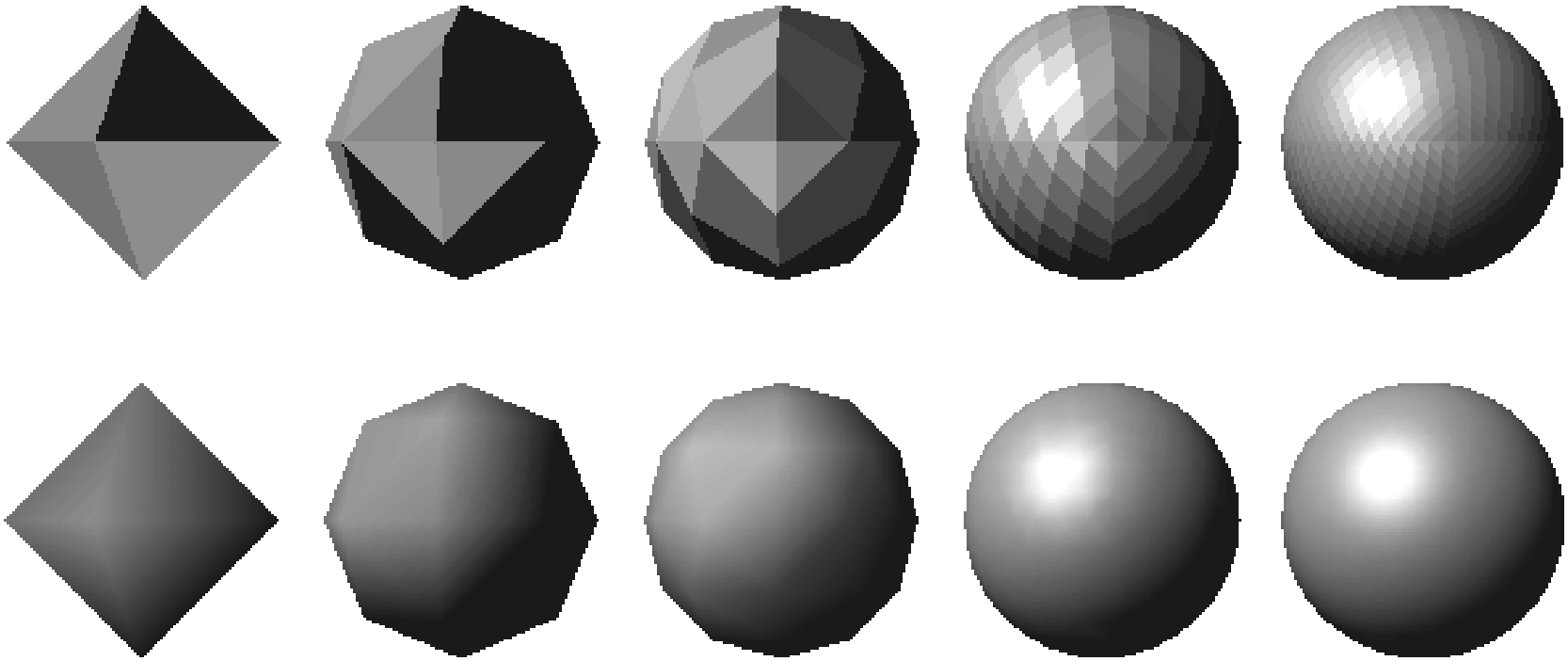
Konstante Schattierung

implizite Annahmen:

- Die Lichtquelle ist unendlich weit entfernt, so dass $\mathbf{n}^\top \cdot \mathbf{l}$ konstant ist.
- Der Betrachter ist unendlich weit entfernt, so dass $\mathbf{n}^\top \cdot \mathbf{v}$ konstant ist.
- Das Polygon repräsentiert die tatsächliche Oberfläche des Objektes und ist nicht nur eine Approximation einer gekrümmten Fläche.

Konstant vs. Gouraud

Konstante Schattierung führt leicht dazu, dass Oberflächen facettenhaft wirken.



Interpolierte Schattierung

Bei der **interpolierten Schattierung** wird die Intensität über einem Dreieck durch Interpolation ermittelt.

Bei der **Gouraud-Schattierung** müssen die Normalenvektoren an den Ecken des Polygongitters bekannt sein.

Entweder werden sie direkt aus der approximierten gekrümmten Fläche berechnet oder als Mittelwert der Normalenvektoren der an die Ecke angrenzenden Polygone.

Gouraud-Schattierung

Danach werden die Eckenintensitäten berechnet (je nach Beleuchtungsmodell)

Die Intensitäten an den Kanten werden mittels linearer Interpolation zwischen den Eckenintensitäten berechnet.

Die Intensitäten innerhalb des Polygons werden mittels Scan-Line-Verfahren linear zwischen den Intensitäten auf den entsprechenden Punkten auf den Kanten interpoliert.

Gouraud-Schattierung

Sind die drei Punkte (x_i, y_i, z_i) gegeben und als zugehörige Intensitäten I_i ($i = 1, 2, 3$) berechnet, bestimmt sich die interpolierte Intensität des Punktes

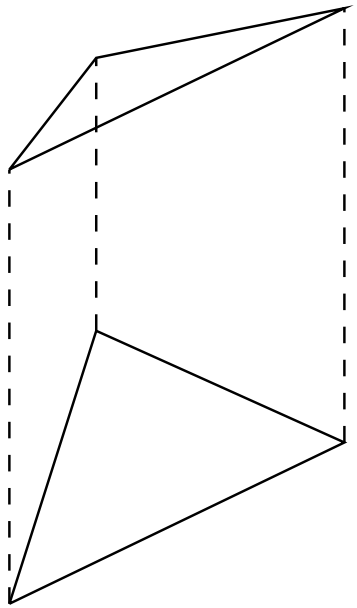
$$t_1 \cdot (x_1, y_1, z_1) + t_2 \cdot (x_2, y_2, z_2) + t_3 \cdot (x_3, y_3, z_3)$$

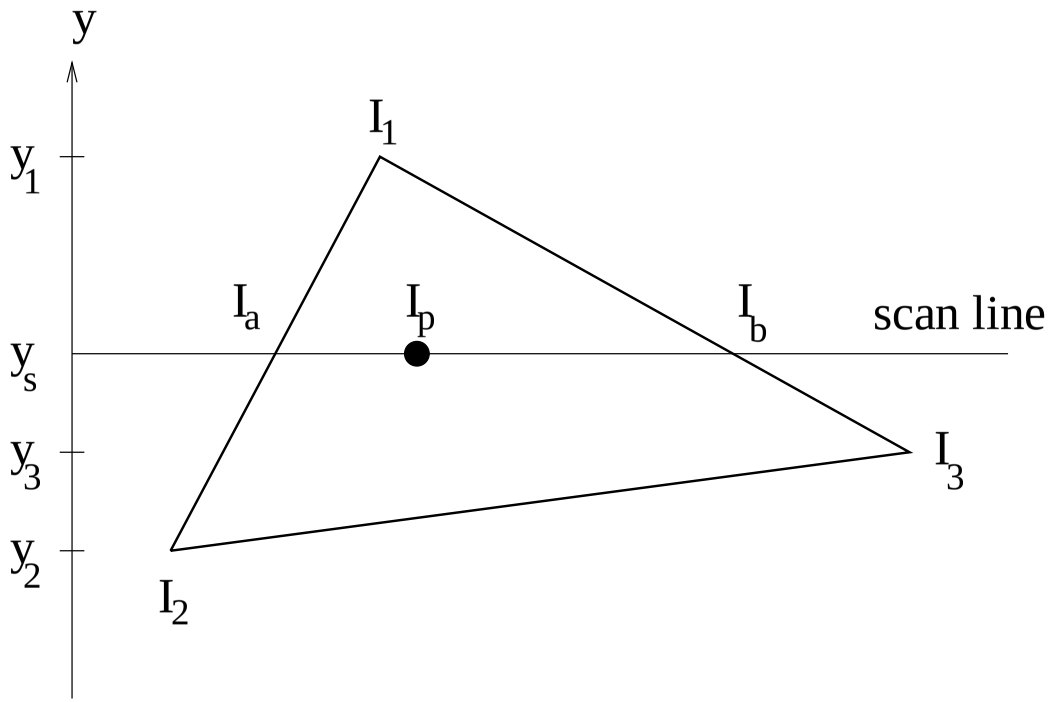
(mit $t_1 + t_2 + t_3 = 1$, $t_1, t_2, t_3 \geq 0$) mit der Formel

$$t_1 \cdot I_1 + t_2 \cdot I_2 + t_3 \cdot I_3$$

Die interpolierte Intensität ist nur ein Näherungswert für die tatsächliche Intensität.

Gouraud-Schattierung





Gouraud-Schattierung

$$I_a = I_1 - (I_1 - I_2)\frac{y_1 - y_s}{y_1 - y_2}$$

$$I_b = I_1 - (I_1 - I_3)\frac{y_1 - y_s}{y_1 - y_3}$$

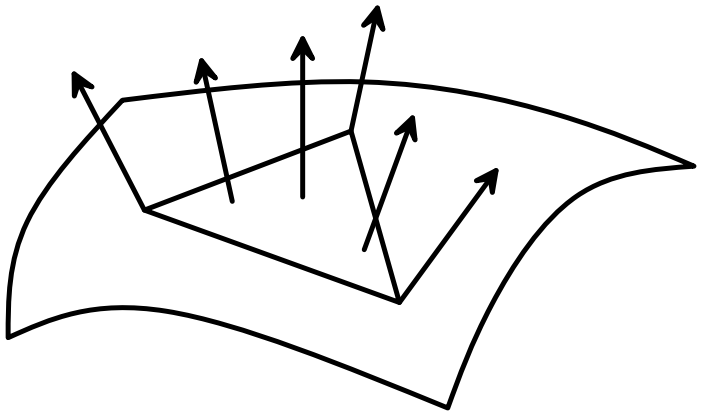
$$I_p = I_b - (I_b - I_a)\frac{x_b - x_p}{x_b - x_a}$$

Bei der Gouraud-Schattierung kann maximale Intensität immer nur in den Ecken angenommen werden.

Phong-Schattierung

Die **Phong-Schattierung** verfährt genauso wie die Gouraud-Schattierung, außer dass anstelle der Intensitäten die Normalenvektoren interpoliert werden.

Dadurch kann sich die maximale Intensität auch im Inneren des Dreiecks ergeben.



Java 3D: Schattierung

In Java 3D wird standardmäßig die Gouraud-Schattierung verwendet.

In Java 3D kann zwischen `SHADE_GOURAUD` und `SHADE_FLAT` gewählt werden.

Bei `SHADE_FLAT` wird eine Fläche (ein Dreieck) jeweils gleichmäßig mit einer Farbe eingefärbt.

Festlegen der gewünschten Schattierung über die `Appearance`:

Java 3D: Schattierung

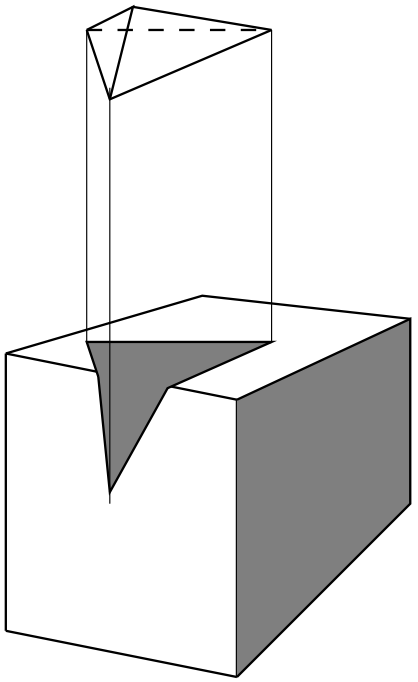
```
Appearance app = new Appearance();

ColoringAttributes ca = new
        ColoringAttributes(
          new Color3f(1.0f,1.0f,1.0f),
          ColoringAttributes.SHADE_FLAT);

app.setColoringAttributes(ca);
```

(vgl. `ShadingExample.java`)

Schatten

Sichtbare Oberflächen, die nicht von der Lichtquelle direkt bestrahlt werden, erhalten nur Streulicht und wirken daher schattiert.

Das gleiche gilt für sichtbare Oberflächen, die von einem anderen Objekt (teilweise) in Richtung der Lichtquelle abgedeckt werden.



Zweipass-z-Puffer-Algorithmus

1. Führe den gewöhnlichen z-Puffer-Algorithmus aus der Sicht der Lichtquelle durch.

2. Führe den z-Puffer-Algorithmus aus der Betrachtersicht mit folgender Modifikation durch:

Wenn ein Punkt bei dem herkömmlichen z-Puffer-Algorithmus in den Puffer eingetragen werden soll (als vorläufig sichtbar erkannt wurde), transformiere seine Betrachterkoordinaten (x, y, z) in die Lichtquellenkoordinaten (x', y', z').

Zweipass-z-Puffer-Algorithmus

Vergleiche den Wert z' mit den im (Lichtquellen-)z-Puffer für (x', y') eingetragenen Wert z_L.

Ist z_L näher an der Lichtquelle als z', liegt (x', y', z') bzw. (x, y, z) im Schatten und wird entsprechend gezeichnet.

Andernfalls liegt (x', y', z') nicht im Schatten und wird direkt von der Lichtquelle bestrahlt.

Transparenz

Bei durchscheinenden Flächen sind zwei Aspekte zu berücksichtigen:

- Wieviel Licht (welcher Farbe) lässt die Fläche durch?
- Wie stark ist die Brechung?

Konturen von Objekten hinter transparenten Flächen lassen sich erkennen.

Transluzente Flächen (wie Milchglas) lassen nur Licht durch, Konturen von dahinter liegenden Objekten sind nicht erkennbar.

Transparenz ohne Brechung

Fläche F_2 liege hinter der transparenten Fläche F_1.

Interpolierte/gefilterte Transparenz: Intensität in einem Pixel P:

$$I_P = (1 - k_{\text{transp}}) \cdot I_1 + k_{\text{transp}} \cdot I_2$$

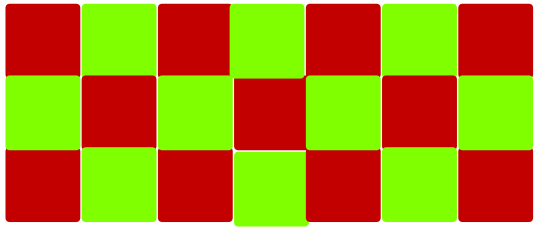
- I_1 ist die Intensität des Pixels, die sich allein aus der Betrachtung der Fläche F_1 ergäbe.
- I_2 ist die Intensität des Pixels, die sich allein aus der Betrachtung der Fläche F_2 ergäbe.
- $k_{\text{transp}} \in [0, 1]$ ist der **Transmissionskoeffizient.**
  $k_{\text{transp}} = 1$: F_1 ist völlig transparent (durchsichtig).
  $k_{\text{transp}} = 0$: F_1 ist überhaupt nicht transparent.

Transparenz ohne Brechung

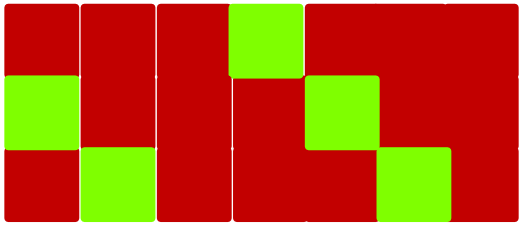
Screen-Door-Transparenz: Pixel werden alternierend mit der Farbe der vorderen und der dahinter liegenden Fläche gefärbt.

Je nachdem, wie transparent die vordere Fläche ist, wird die Mehrheit der Pixel mit Farbe der hinteren bzw. der vorderen Fläche eingefärbt.

vordere/hintere Fläche:



50% Transparenz



25% Transparenz

Java 3D: Transparenz

```
TransparencyAttributes ta =
    new TransparencyAttributes();

ta.setTransparencyMode(
   TransparencyAttributes.BLENDED);

ta.setTransparency(transpValue);

trApp.setTransparencyAttributes(ta);
```

`transpValue` ist ein `float`-Wert zwischen 0 und 1, der angibt, wie transparent (1 = völlig durchsichtig) die Oberfläche sein soll.

Java 3D: Transparenz

Für den `TransparencyMode` stehen zur Verfügung:

- `BLENDED`: interpolierte/gefilterte Transparenz
- `SCREEN_DOOR`: Screen-Door-Transparenz
- sowie: `NICEST, FASTEST, NONE`

(vgl. `TransparencyExample.java`)

Texturen

Textur: Bild, das man auf eine Oberfläche zur Modellierung von Details aufbringt

Bilder auf Oberflächen: z.B.

- bemalte Wand
- Landschaften als Hintergrund, die nicht explizit dreidimensional modelliert werden sollen
- Abbildung einer Weltkarte auf einen Globus

Muster/Strukturen/Maserungen: z.B.

- Holzmuster
- Reliefs oder Faltenwurf von Stoff

Texturen

Texturen werden z.T. anstelle expliziter dreidimensionaler Modellierung eingesetzt (z.B. Hintergrundlandschaft, Reliefs), wo eine dreidimensionale Modellierung unnötig oder zu aufwendig ist.

Bei Mustern, Maserungen oder Reliefs ist man i.A. an einer wiederholten Aufbringung der Textur interessiert, während Bilder meistens nur einzeln auf ein und dieselbe Oberfläche abgebildet werden.

Texturen

Oberflächendetailpolygone könnten grundsätzlich für die Texturmodellierung verwendet werden.

Oberflächendetailpolygone verfeinern die eigentlich Oberflächenpolygone, so dass Texturdetails einzeln auf den Oberflächendetailpolygonen aufgebracht werden könnten.

Oberflächendetailpolygone sind jedoch extrem aufwendig und eignen sich nur für einfach Strukturen wie Buchstaben.

Texturen

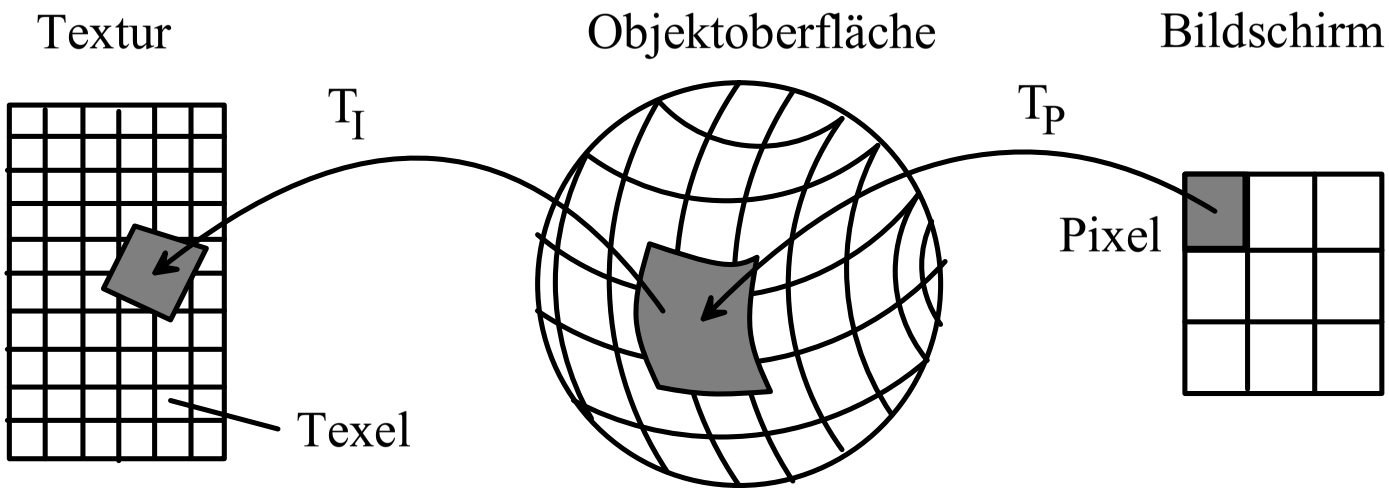


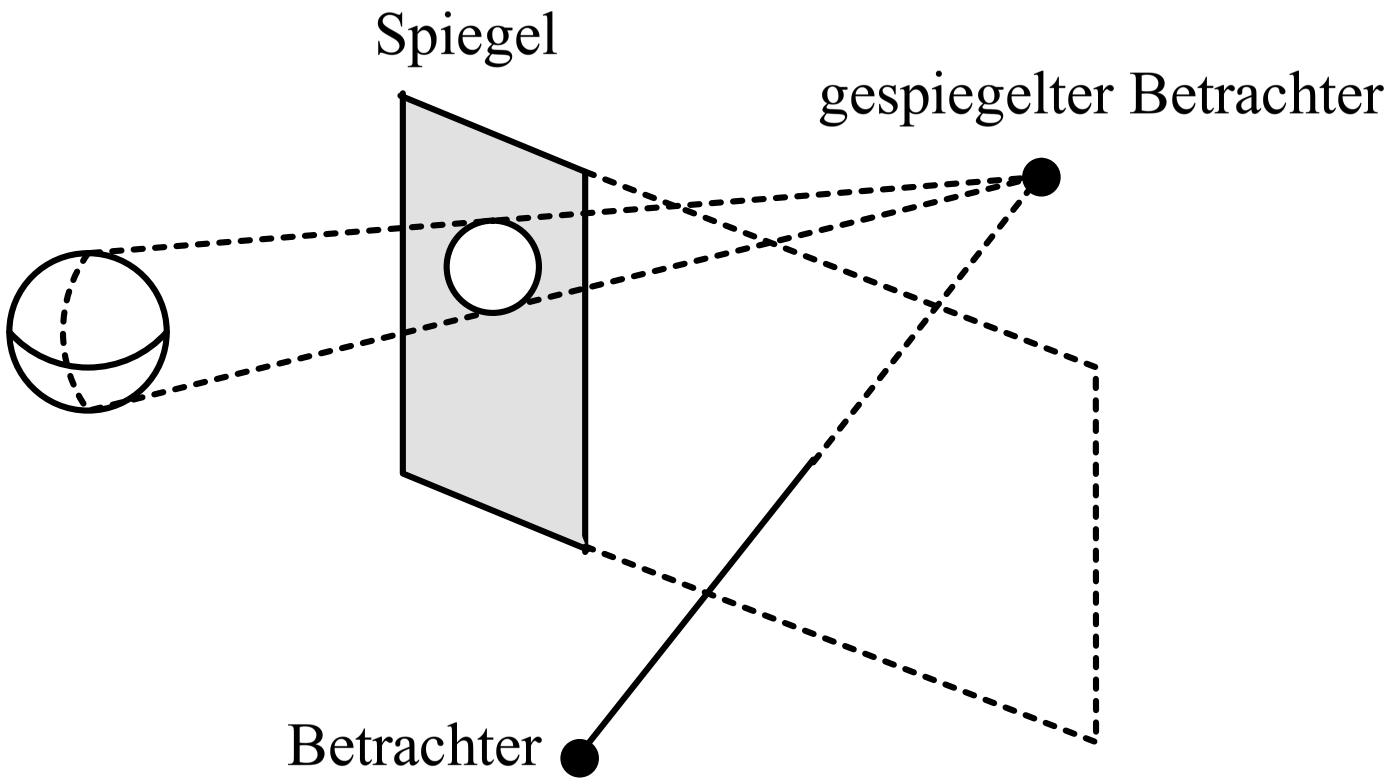
Abbildung einer Textur auf eine Oberfläche

Texturen

Darstellung spiegelnder Oberflächen mittels Environment-/Reflection-Mapping:

- Spiegelung des Betrachters an der Spiegelebene
- gespiegelter Betrachter wird als neues Projektionszentrum angesehen
- Projektion der Szene auf die Spiegeloberfläche als Texture

Texturen



Bump-Mapping

Oberfläche bleibt flach beim Aufbringen einer Textur.

Beim Bump-Mapping wird ähnlich anstelle einer gewöhnlichen Textur an jedem Texturpunkt ein Störwert $B(i, j)$ definiert, der den korrespondierenden Punkt auf der Oberfläche in Richtung des Normalenvektors verschiebt.

Ist die Fläche in parametrisierter Form gegeben und der zu modifizierende Punkt $P = P(x(s,t), y(s,t), z(s,t))$, ergibt sich der nicht-normalisierte Normalenvektor bei P aus dem Kreuzprodukt der partiellen Ableitungen nach s und t:

Bump-Mapping

$$\mathbf{n} = \frac{\partial P}{\partial s} \times \frac{\partial P}{\partial t}$$

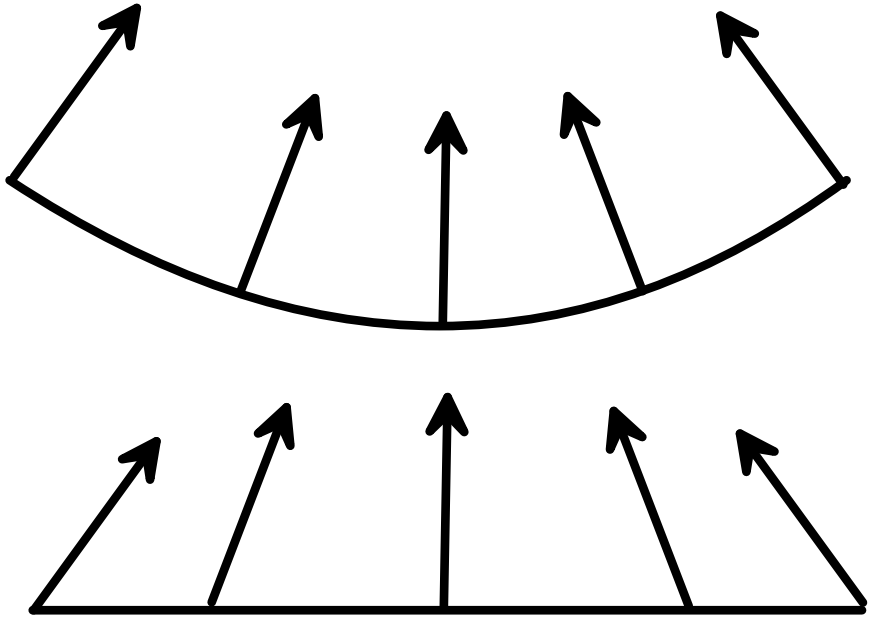
Ist $B(T(P)) = B(i,j)$ der entsprechende Bump-Wert, ist der verschobene Punkt auf der Oberfläche:

$$P' = P + B(T(P)) \cdot \frac{\mathbf{n}}{\| \mathbf{n} \|}$$

Gute Annäherung für den neuen Normalenvektor:

$$\mathbf{n}' = \frac{\mathbf{n} + \mathbf{d}}{\| \mathbf{n} + \mathbf{d} \|} \quad \text{mit} \quad \mathbf{d} = \frac{\frac{\partial B}{\partial u} \cdot \left(\mathbf{n} \times \frac{\partial P}{\partial t} \right) - \frac{\partial B}{\partial v} \cdot \left(\mathbf{n} \times \frac{\partial P}{\partial s} \right)}{\| \mathbf{n} \|}$$

Bump-Mapping



Erzeugung des Eindrucks einer Mulde durch modifizierte Normalenvektoren

Java 3D: Texturen

zunächst: Laden des Bildes und Umwandlung in eine `ImageComponent2D`.

Die Höhe und Breite der `ImageComponent2D` müssen Zweierpotenzen sein!

```
TextureLoader textureLoad =
   new TextureLoader("image.jpg",null);

ImageComponent2D textureIm =
   textureLoad.getScaledImage(128,128);
```

Java 3D: Texturen

Dann wird eine `Textur2D` erzeugt, die dann einer `Appearance` zugeordnet werden kann.

```
Texture2D myTexture =
    new Texture2D(
      Texture2D.BASE_LEVEL,Texture2D.RGB,
      textureIm.getWidth(),
      textureIm.getHeight());

myTexture.setImage(0,textureIm);

Appearance textureApp =
                  new Appearance();
```

Java 3D: Texturen

```
textureApp.setTexture(myTexture);

TextureAttributes textureAttr =
        new TextureAttributes();

textureAttr.setTextureMode(
      TextureAttributes.REPLACE);

textureApp.setTextureAttributes(
                    textureAttr);

textureApp.setMaterial(
                  new Material());
```

TextureExample.java

Prinzipiell kann genau angegeben, wie bzw. an welchen Stellen die Texture im Detail auf der Oberfläche angebracht werden soll.

`TexCoordGeneration` bietet eine einfache, aber nicht immer optimale Möglichkeit, Texturen automatisch auf Oberflächen anzubringen.

```
TexCoordGeneration tcg =
new TexCoordGeneration(
TexCoordGeneration.OBJECT_LINEAR,
TexCoordGeneration.TEXTURE_COORDINATE_2);

textureApp.setTexCoordGeneration(tcg);
```

Java 3D: Texturen

Texturen als Hintergrund in Java 3D

```
TextureLoader textureLoad =
 new TextureLoader("image.jpg",null);

Background bg =
 new Background(textureLoad.getImage());

bg.setApplicationBounds(bounds);
```

Java 3D: Texturen

`bounds` ist wie üblich z.B. eine `BoundingSphere`, die den Gültigkeitsbereich des Hintergrundes angibt.

Hinzufügen der Textur zur Szene:

```
theScene.addChild(bg);
```

Anstelle des Ladens eines Hintergrundbildes kann der Hintergrund auch durch eine Farbe definiert werden:

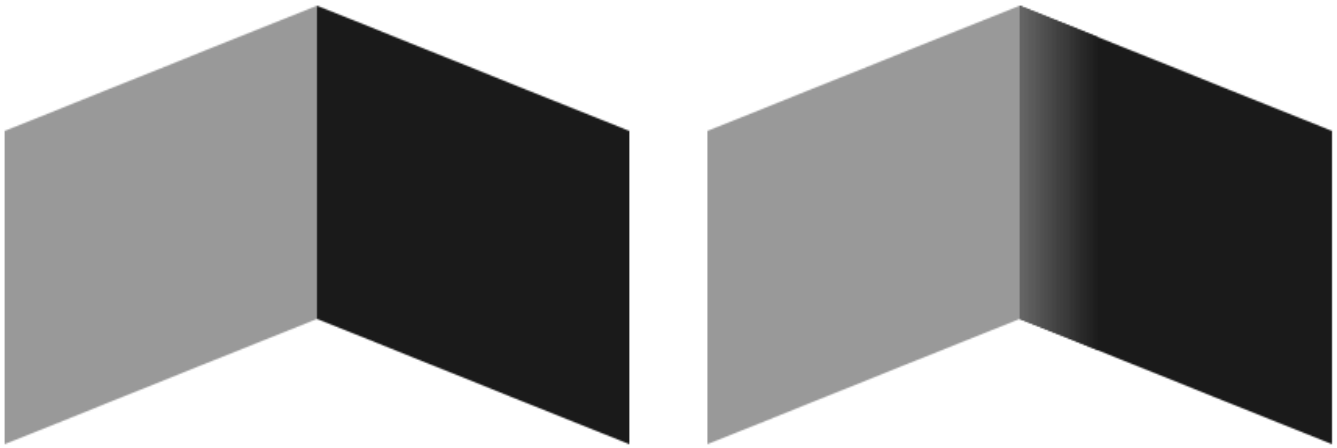
```
Background bg = new Background(colour);
```

(vgl. `BackgroundExample.java`)

Radiosity-Methoden

Bisher: Beleuchtung der Objekte nur durch Lichtquellen und allgemeines Streulicht, keine Interaktion zwischen den Objekten

Führt zu scharfen Kanten zwischen Flächen



Radiosity-Methoden

Rate der Energie, die eine Oberfläche abgibt: **Radiosity**

Die Radiosity ergibt sich aus der Summe des reflektierten, des (von hinten) durchgelassenen Lichts und der Eigenemittierung.

Radiosity-Modell: Jede Oberfläche wird als Lichtquelle angesehen.

Interaktion der Oberflächen unabhängig vom Standpunkt des Betrachters bei ausschließlich diffuser Reflexion

Die Radiosity-Gleichung

$$B_i = E_i + \varrho_i \cdot \sum_{j=1}^{n} B_j \cdot F_{ji} \cdot \frac{A_j}{A_i}$$

- B_i: Radiosity des Flächenstücks i gemessen in (Energie/Zeit)/Fläche
- E_i: Energie, mit der das Flächenstück i Licht aussendet (selbstleuchtend)
- ϱ_i: Reflektivität des Flächenstücks i (Anteil des reflektierten Lichtes, einheitenlos)

Die Radiosity-Gleichung

- F_{ji}: dimensionsloser Form- oder Konfigurationsfaktor, der den Anteil des vom Flächenstücks j abgestrahlten Lichts spezifiziert, der i trifft. (In F_{ji} wird die Form und die relative Orientierung der beiden Flächenstücke zueinander berücksichtigt.)
- A_i: Flächeninhalt des Flächenstücks i

Die Radiosity-Gleichung

In Umgebungen mit ausschließlich diffuser Reflexion gilt:

$$A_i F_{ij} = A_j F_{ji}$$

und damit

$$B_i = E_i + \varrho_i \cdot \sum_{j=1}^{n} B_j \cdot F_{ij}$$

d.h.

$$B_i - \varrho_i \cdot \sum_{j=1}^{n} B_j \cdot F_{ij} = E_i$$

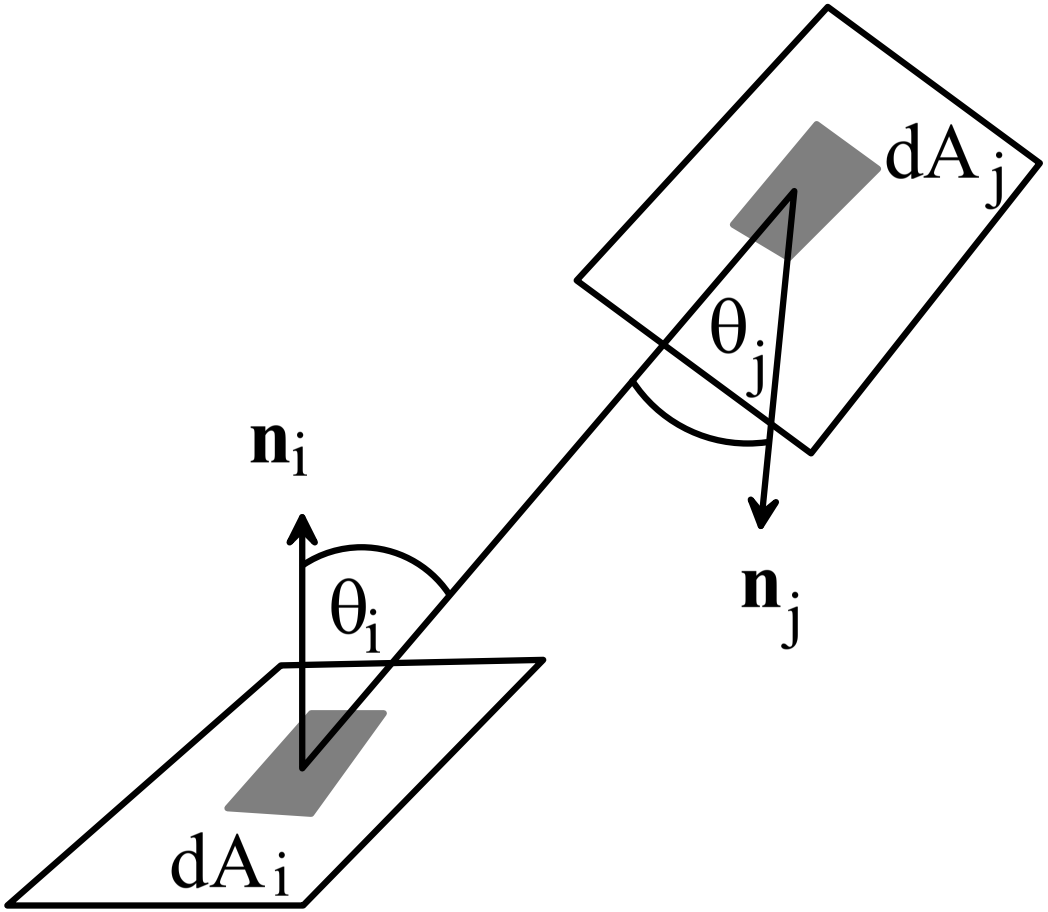
Die Radiosity-Gleichung

$$\begin{pmatrix} 1-\varrho_1 F_{1,1} & -\varrho_1 F_{1,2} & \dots & -\varrho_1 F_{1,n} \\ -\varrho_2 F_{2,1} & 1-\varrho_2 F_{2,2} & \dots & -\varrho_2 F_{2,n} \\ \vdots & \vdots & \vdots & \vdots \\ -\varrho_n F_{n,1} & -\varrho_n F_{n,2} & \dots & 1-\varrho_n F_{n,n} \end{pmatrix} \cdot \begin{pmatrix} B_1 \\ B_2 \\ \vdots \\ B_n \end{pmatrix} = \begin{pmatrix} E_1 \\ E_2 \\ \vdots \\ E_n \end{pmatrix}$$

Dieses lineare Gleichungssystem muss für jede Wellenlänge (oder zumindest für RGB) gelöst werden.

(ϱ_i und E_i hängen von der Wellenlänge ab.)

Radiosity-Methoden



Formfaktor von der (differentiellen) Fläche dA_i zur (differentiellen) Fläche dA_j:

$$dF_{d_i,d_j} = \frac{\cos(\theta_i) \cdot \cos(\theta_j)}{\pi \cdot r^2} \cdot H_{ij} \cdot dA_j$$

$$H_{ij} = \begin{cases} 1 & \text{falls } dA_j \text{ von } dA_i \text{ aus sichtbar ist} \\ 0 & \text{sonst} \end{cases}$$

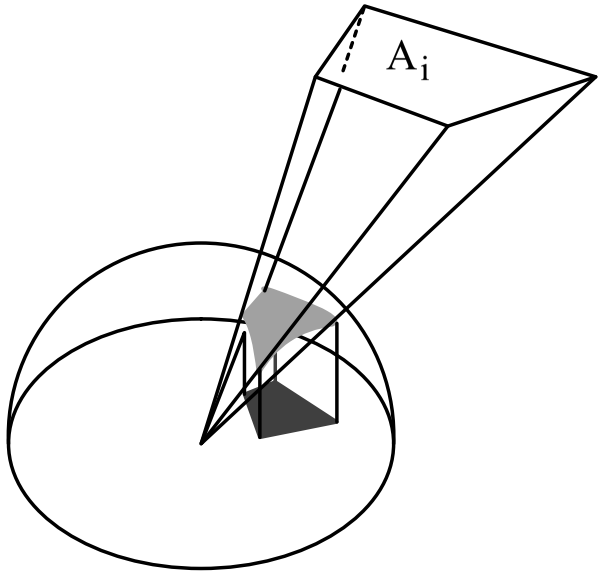
Formfaktor von der (differentiellen) Fläche dA_i zur Fläche A_j:

$$dF_{d_i,j} = \int_{A_j} \frac{\cos(\theta_i) \cdot \cos(\theta_j)}{\pi \cdot r^2} \cdot H_{ij}\, dA_j$$

Formfaktor von der Fläche A_i zur Fläche A_j:

$$F_{i,j} = \frac{1}{A_i} \int_{A_i} \int_{A_j} \frac{\cos(\theta_i) \cdot \cos(\theta_j)}{\pi \cdot r^2} \cdot H_{ij}\, dA_j\, dA_i$$

Radiosity-Methoden



Approximative Bestimmung des Formfaktors: Der Anteil der projizierten Fläche in dem Kreis entspricht dem Formfaktor.

Näherungslösung des Radiosity-Gleichungssystems: schrittweise Verfeinerung

iterative Berechnung der B_i

1. Setze alle $B_i = E_i$ und $\Delta B_i = E_i$
2. Wähle O_{i_0} mit dem größtem ΔB_{i_0}
3. Neuberechnung der B_i und der ΔB_{i_0}

$$B_i^{\text{(neu)}} = B_i^{\text{(alt)}} + \varrho_i \cdot F_{i_0 i} \cdot \Delta B_{i_0}$$

$$\Delta B_i^{\text{(neu)}} = \begin{cases} \Delta B_i^{\text{(alt)}} + \varrho_i \cdot F_{i_0 i} \cdot \Delta B_{i_0} & \text{falls } i \neq i_0 \\ 0 & \text{falls } i = i_0 \end{cases}$$

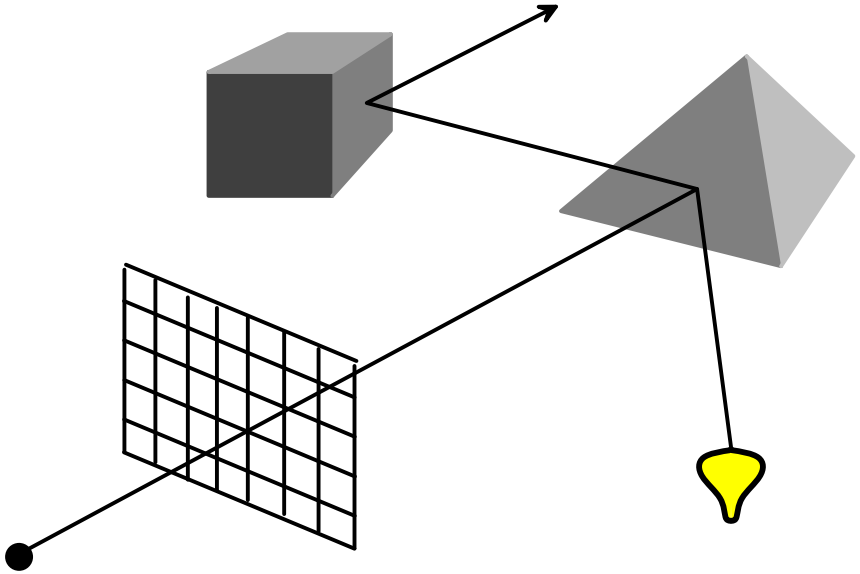
Radiosity-Methoden

Wiederhole Schritte 2. und 3., bis sich nur noch minimale Änderungen ergeben oder bis eine Maximalanzahl von Iterationsschritten erreicht wurde.

Vorteil der schrittweisen Verfeinerung: Kann jeder Zeit abgebrochen werden und verbessert Schattierung schrittweise.

Spiegelreflexion muss nachträglich hinzugefügt werden.

Ray-Tracing



Rekursives Ray-Tracing für realistischere Spiegelreflexion