Einführung in die Computergrafik

Für Modulklausur

Modellierung

- Mengen: Einzelne große Buchstaben (N, R)
- Skalare: Einzelne kleine Buchstaben (lateinisch, griechisch) (a, b, α, β)
- Punkte: Kleine fette Buchstaben unterstrichen ($\underline{a}$)
- Vektoren: Kleine Buchstaben mit Pfeil ($\vec{a}$)
- Farben (RGBA): Drei Punkte über Kleinbuchstaben ($\ddot{c}$)
- Normierte Vektoren: $\hat{a}$
- Matrizen: Einezlner großer Buchstabe
- Homogene Vektoren (Dimension um eins erweitert und letzte für letzte Dimension eins als Koordinate): Schlange ($\tilde{a}$)
- Homogene Matrizen: Schlange über einzelnem großen Buchstaben ($\tilde{A}$)
- Drahtgitter: Graphen von Kanten und Kurven
- Flächenmodell: Erweiterung durch Flächen
- Körpermodell: Mengenoperationen über Grundkörper (CSG) / Abzählen von Zellen (Octree)
- Topologie: Gibt es eine Verbindung zwischen zwei Punkten auf einem Objekt?
- Geometrisches Modellieren: Rechnergestütztes Entwerfen von geometrischen Objekten und ihren Lagebeziehungen; Alternativ: Einscannen mittels 3D-Kamera
- Einheitskreis: $\sin^2\alpha + \cos^2\alpha = 1$
- Skalarprodukt: $\vec{a} * \vec{b} = \langle \vec{u}, \vec{v} \rangle = u_x v_x + u_y v_y + u_z v_z$. Skalarprodukt entspricht $\langle \vec{u}, \vec{v} \rangle = |\vec{u}| * |\vec{v}| * \cos\alpha$
- Kreuzprodukt: $\vec{u} \times \vec{v}$: Ergebnis orthognal zu $\vec{u}$ und $\vec{v}$ ($\vec{u} \perp \vec{u} \times \vec{v} \perp \vec{v}$) - Gibt außerdem Flächeninhalt an, der von $\vec{u}$ und $\vec{v}$ aufgespannt wird
- Orthonormales Koordinatensystem zu einem normierten Vektor $\vec{z}$ aufstellen: $\hat{x} = \frac{\vec{v} \times \hat{z}}{||\vec{v} \times \hat{z}||}$ und $\hat{y} = \hat{z} \times \hat{x}$. $\vec{v}$ nicht parallel zu $\vec{z}$. $\vec{v} = \vec{0}$ und kleinste Komponente von z in v auf 1 setzen
- Winkelberechnung: $\arctan 2(\langle \vec{u}, \vec{v} \rangle, ||\vec{u} \times \vec{v}||)$
- Spatprodukt: $[\vec{u}, \vec{v}, \vec{w}] = \langle \vec{u}, \vec{v} \times \vec{w} \rangle = \det(\vec{u}, \vec{v}, \vec{w})$, Tetraedervolumen ein sechstel davon
- $\text{dist}(\underline{x}) = \langle \hat{n}, \underline{x} - \underline{r} \rangle$, wobei $\underline{x}$ der Punkt ist und $\underline{r}$ ein Punkt auf der Ebene
- Projektion eines Punktes auf den nächstgelegenen Punkt der Ebene (Lot): $\text{Proj}(\underline{x}) = \underline{x} - \text{dist}(\underline{x})\hat{n}$
- Baryzentrische Koordinaten: Punkt $\underline{p}$ im Dreieck $\underline{p_0}$, $\underline{p_1}$, $\underline{p_2}$ mit $\underline{p} = \sigma_0\underline{p_0}\sigma_1\underline{p_1}\sigma_2\underline{p_2}$ und $\sigma_0 + \sigma_1 + \sigma_2 = 1$. Im Fall $\underline{p} = \underline{p_i}$ gilt $\sigma_i = 1$ und $\sigma_{j \neq i} = 0$. Berechnung mit LGS:

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} x_0 & x_1 & x_2 \\ y_0 & y_1 & y_2 \\ z_0 & z_1 & z_2 \end{pmatrix} \begin{pmatrix} \sigma_0 \\ \sigma_1 \\ \sigma_2 \end{pmatrix}$$

  Positionstests: Auf Eckpunkten ist ein σ 1, auf Kanten ist eins 0. Ist ein σ kleiner als null oder größer
- Modellierungs Ansätze: Implizit ($f(x, y) = x^2 + y^2 - r^2$ gegeben, auf Formrand ist $f = 0$ (Formrand = Implizite Kurve), innen $f < 0$, außen $f > 0$), Parametrisch mit Parameter t und einem Vektor, der für jedes t einen Punkt zuordnet (Umkehrfunktion: Einbettung)
- Implizite Modellierung: Sehr viele Körperteile und Schleifen können definiert werden ohne sie explizit modellieren zu müssen, Abspalten und Vereinen von Schleifen bei Animation erfordert keine Beachtung, aber offene Flächen etc. können nicht modelliert werden
- Parametrische Modellierung: Intuitive lokale Änderung von Form möglich über "Kontrollpunktparadigma", Zusammensetzungen / Ausblendungen ermöglichen einfache Kombination, dabei kann es aber zu ungewollten scharfen Ecken / Kanten kommen
- Facette: Fläche eines Körpers, Attribute werden pro Facettenecke angegeben (Keil / corner / wedge genannt / vertex), z. B. Texturkoordinate, Farbe, Normale
- Umlaufsinn: Ecken werden in konsistenter Richtung durchlaufen (Finger in die Richtung: Daumen zeigt Normalenrichtung)
- Backface Culling: Nichtzeichnen von Flächen, auf die "von innen" geschaut wird
- Indizierte Speicherung von Modelldaten: Zu Beginn der Modellbeschreibung wird eine Liste mit allen Vertexpositionen und Normalen gespeichert und später nur noch der Index angegeben
- Normalenberechnung: Im Dreieck einfach Kreuzprodukt von zwei Kanten, bei Polygonen gegen den Uhrzeigersinn das Kreuzprodukt von je zwei Punkten nehmen (und das vom letzten Punkt mit dem Ersten) und Addieren
- Normalenschätzung für einen Knoten: Alle Normalenvektoren der umliegenden Flächen addiert und jeweils gewichtet (gleich, mit Keilwinkel oder Dreiecksfläche) - bei scharfen Kanten sollte für jede Ecke eigene Normale benutzt werden
- Triangulierung: Zerlegung eines Polygons in Dreiecke
- Einfaches Polygon: Eine Schleife (in jedem Punkt schneiden sich genau 2 Kanten), bei allgemeinen Polygonen schneiden sich Kanten ggf.
- Polygon ist konvex, wenn die Verbindung zweier beliebiger Punkte im Polygon liegt (hinreichend) und alle Innenwinkel kleiner als 180 Grad sind (hinreichend)
- Für die Winkelberechnung wird zusätzlich 3. Dimension eingeführt, die null ist (beim Kreuzprodukt dann als einziges nicht null ist), $||\vec{u} \times \vec{v}|| = |u_x v_y - u_y v_x$
- Polygonecke (Drei aufeinanderfolgende Knoten) zu mittlerem Knoten ist Ohr, wenn Innenwinkel kleiner 180 Grad sind (konvex) und kein Knoten im beschriebenen Dreieck liegt
- Ear-Cutting-Algorithmus: Alle Ohren abschneiden. In Pseudo-Code:

  1. Klassifiziere alle Knoten als Konkav / Konvex
  2. $n - 3$ mal wiederholen: Für alle konvexen Knoten prüfen, ob Ecke ein Ohr ist (nur gegen konkave Knoten testen) und ggf. Ohr abschneiden und benachbarte Knoten neu klassifizieren und innere Iteration abbrechen
  3. Ein Polygon mit n Knoten enthält $n - 2$ Dreiecke, daher müssen $n - 3$ Ear-Cuts gemacht werden. Worst-Case-Laufzeit: $O((n-3) * k * (n-k))$

- Texture Mapping: Zuordnen eines Bildes (Textur) zu einer Fläche, Bump Mapping: Änderung der Normale durch Bump Mapping. Texturkoordinaten im Wertebereich 0 bis 1. Texturkoordinaten werden baryzentrisch interpoliert.
- Benachbarte Dreiecke sollten auch im "Texturraum" benachbart sein
- Wavefront-Object-Format (*.obj): Zeilenbasiert, eine Zeile beginnt mit "v" (Vertexkoordinaten) / "vt" (Vertexindex) / "vn" (Normale) / "f" (Flächenbeschreibung). Flächenbeschreibung mit Tripel aus Positions- / Textur- und Normalenindex pro Knoten (z. B. "f 1/1/2 3/1/3 4/3/2". Texturkoordinate kann z. B. weggelassen werden
- Szenengraph (Hierachische Organisation der Objekte einer Szene): Besteht aus Gruppenknoten, Objektknoten, Transformationsknoten, Geometrieknoten, Kameraknoten, Materialknoten, Lichtknoten, Animationsknoten (vgl. Komposition bei SWT!)
- Graphen: Knotentiefe (Weg bis zur Wurzel inklusive Wurzel und Knoten)
- Implizite Modellierung: Repräsentierte Kurve / Fläche als Nullstellenmenge über $\mathbb{R}^2$ oder $\mathbb{R}^3$ (Einheitskugel: $f(x,y,z) = x^2 + y^2 + r^2 - 1 = 0$
- Gradient $\vec{\triangle} f$ sind die partiellen Ableitungen von f komponentenweise. An Flächenpunkten zeigt $\vec{\triangle} f$ in Normalenrichtung.
- Implizit repräsentierte Kurve / Fläche kann durch Marching Squares / Cubes mit einem Polygon angenähert werden. Schritte:

  1. Definiere Gitter aus Ausdehnung und Auflösung
  2. Pro Knoten: Werte Funktion aus und klassifiziere in innen außen
  3. Pro Kante, die innen mit außen verbindet, erzeuge Kantenpunkt
  4. Pro Zelle verbinde Kantenpunkte mit Liniensegmenten

  Wenn eine Kante innen mit außen verbindet, muss es auf der Kante eine Nullstelle der Funktion geben.
  Es gibt 16 mögliche Fälle (4 Symetrieklassen). Ist auf allen 4 Kanten einer Zelle ein Punkt, gibt es zwei Möglichkeiten, wie die Kantenpunkte verbunden werden können.
- Implizite Modellierung: Funktionen können z. B. auch vereinigt werden oder die Schnittmenge genommen werden (min und max)
- Constructive Solid Geometry (CSG): Verknüpfung durch Mengenoperationen (Schnittmenge, Differenz, Vereinigung etc.)
- Ableitung nach t: "Geschwindigkeitsvektor" spannt zusammen mit dem Kurvenpunkt die Tangente auf
- Parametrische Flächen werden über zwei Paramter u, v parametrisiert. Partiell abgeleitet nach u' und v' spannen sie die Tangentialebene auf
- Das Kreuzprodukt der Tangentenvektoren ergibt die Flächennormale von A

Rasterisierung

- Speicher $\leftrightarrow$ CPU $\rightarrow$ Matrizmultiplizierstufe $\rightarrow$ Clippingteilstufe $\rightarrow$ Vektorengenerator $\rightarrow$ CRT
- Vektordisplays: "Braunsche Röhre", Elektronenstrahl wird mittels Elektromagneten auf Bildschirmposition gelenkt und zeichnet so die zu zeichnenden Formen direkt nach. Rasterdisplays: Pixelmatrix. Um Formen zu zeichnen, werden diese vorher gerastert.
- Vektordisplays: geringer Speicherbedarf, schnelles Umschalten für Animation, begrenzte Komplexität möglich (Füllen ist aufwendig), Beispiele: Plotter
- Rasterdisplays: Hohe Parallelisierbarkeit, beliebige Komplexität, unterschiedliche Displaytechnologien (IPS, TFT), Abtastprobleme, Beispiele: Rasterdisplay, Tintenstrahldrucker
- Primitive zum Rasterisieren: Stilisierte Kurven, Polygone, Ellipsensegmente, Füllalgorithmen, Schriften, Verläufe, Muster
- Rastergrafik-APIs für Anwendungen: Java AWT, Java 2D, Qt, Gtk+
- Echtzeitgrafik: Szene wird tesseliert (in ebene Drei- und Vierecke zerlegt) und an die GPU gesendet. Diese transformiert die Knoten in Bildkoordinaten, zerlegt die Polygone in Fragmente (Pixel) und berechnet deren Farbe und Tiefe
- Rasterisieren von Linien: Rasteralgorithmus sollte keine doppelte Dicke haben, einfach zu implementieren sein, ganzzahlige Arithmetik verwenden (Numerische Ungenauigkeiten vermeiden) und inkrementell arbeiten
- DDA, Bresenham, Mittelpunkt
- Vor dem Rasterisieren wird jede Linie $\underline{p} \rightarrow \underline{q}$ auf den Standardfall zurückgeführt: $\underline{p}$ im 4. Quadranten (X negativ, Y positiv), $\underline{q}$ im 1. Quadranten (X positiv, Y positiv), wobei $\underline{p}_y < \underline{q}_y$ gilt
- DDA (Digital Differential Analyzer): Steigung m ist konstant, daher gilt $y_i = y_0 + m(x_i - x_0)$. Inkrementelle Ablaufweise / Pseudocode

  1. y mit y0 initialisieren
  2. m setzen: $m = (y_n - y_0)/(x_n - x_0)$

- Bresenham: Welcher Mittelpunkt eines Pixels liegt näher an der Linie? Mittelpunktalgorithmus: Auf welcher Seite liegt der Mittelpunkt der angrenzenden Pixel?
- Bei Linien von $\underline{p} - \underline{q} = \begin{pmatrix}\Delta x \\ \Delta y\end{pmatrix}$ gilt $\vec{n} = \begin{pmatrix}-\Delta y \\ \Delta x\end{pmatrix}$ (90 Grad-Winkel)
- Für die Berechnung des Abstands zu einer Linie: Abstand von $\underline{x}$ zur Linie ist $\frac{\langle \underline{x} - \underline{p}, \vec{n}\rangle}{||\vec{n}||}$
- Bresenham: $2\Delta y - \Delta x \leq 2d_i$ zur Entscheidung, ob man nach rechts oder nach rechts oben geht ($d_{i+1}^{\text{rechts}} = \langle \underline{x_i} + \begin{pmatrix}1 \\ 0\end{pmatrix} - \underline{p}, \vec{n}\rangle = d_i - \Delta y$ bzw. $d_{i+1}^{\text{rechtsoben}} = \cdots = d_i - \Delta y + \Delta x$
- Ablauf zur Rasterisierung: y auf y_0 initalisieren, d auf 0, Δx und Δy ausrechnen mit $x_n - x_0$ bzw. $y_n - y_0$. $\Delta d_{(+1,0)} = -\Delta y$ und $\Delta d_{(+1,+1)} = \Delta x - \Delta y$ setzen. $c_{\text{test}} = 2 * \Delta y - \Delta x$ setzen. Dann x von x_0 nach x_n iterieren und den Pixel an x, y setzen. Dann entscheiden, ob $c_{\text{test}} \leq 2d$ gilt. Tut dies, dann $d = d + \Delta d_{+1,0}$ setzen, ansonsten $y+ = 1$ und $d+ = \Delta d_{(+1,+1)}$ setzen
- Treppenstufenartefakte: Treten durch Unterabtastung des Pixelgitters auf. Behebung durch Antialiasing (Kantenglättung): Darstellung in höherer Auflösung (und anschließender Herunterrechnung) oder Filterung während des Zeichnens (nicht gut, weil erfordert Transparenz und

führt zu fehlerhaften Überlappungen o. ä.)
- Gekrümmte Linien: Inkrementelle Algorithmen (vgl. $x_2 + y_2 = r_2$
- Dicke Linien zeichnen durch widerholen einer Maske ("Pinsel"): Ineffizient oder durch Randberechnung und Füllen (kompliziert)
- Füllalgorithmen (für Dreiecke, Polygone, Kreise): Farbe, Muster, Farbverlauf, Struktur zu füllen. Dabei werden farblich *ähnliche* nahe Pixel gefüllt
- Entweder Füllen mit 4er-Nachbarschaft (kann zu unvollständigem Füllen führen, da diagonal nicht gefüllt wird) oder 8er-Nachbarschaft (kann zu Ausläufen führen)
- Rekursives Füllen: Funktion fill (x,y) färbt den Pixel an x, y (falls dieser nicht außerhalb des Bildbereichs liegt und eine ähnliche Farbe wie der) und ruft sich selbst für die umliegenden Pixel auf. Toll einfach zu implementieren, aber führt zu Stapelüberläufen und abhängig von Definition der Pixelnachbarschaft
- Pixellaufalgorithmus: Füllfunktion füllt eine Linie in einem Polygon auf und startet Rekursion für Pixel der darüber und darunterliegenden Zeile, an denen ein Pixellauf von rechts beginnt
- Graphisches Debuggen: Fehler treten erst nach sehr vielen Iterationen auf, daher muss ein "graphischer Debugger" geschaffen werden, der erlaubt, die Anzahl der Iterationen für einen Algorithmus vorzugeben. Vorher wird Zustand der Datenstrukturen gespeichert. Dann wird der Algorithmus mit verschiedenen Iterationszählern gestartet.
- Brute-Force-Algorithmus für das Füllen von Dreiecken: Rechteck um das Dreieck wird betrachtet, anschließend werden die baryzentrischen Koordinaten für jeden Pixel bestimmt und damit geprüft, ob der Pixel im Dreieck liegt (inkrementell)
- Sweep-Line-Algorithmus: Alle vom Dreieck geschnittenen Zeilen werden durchwandert (da Dreieck konvex ist, ist der Schnit ein Interval). Interval und Koordinaten werden von Zeile zu Zeile mitgeführt und alle inneren Pixel auf berechnete Farbe gesetzt. Bei Polygonen können dies entsprechend mehrere Intervalle sein ("Liste aktiver Intervalle wird mitgeführt"). Bei Dreiecken nebeneinander soll jeder Pixel nur einmal gezeichnet werden (keine Löcher, keine doppelten Pixel bei Transparenz).
  Daher zeichne Pixel, deren Zentrum im Dreieck liegt. Pixel auf Kanten: Zwei Richtungen werden festgelegt, wobei der Punkt der ersten Richtung zugerdnet wird, wenn die Kante nicht parallel zur ersten Richtung ist. Punkt auf Ecke: Wenn Kante parallel zur ersten Richtung die Ecke verlässt, weise Punkt dem rechts gelegenen Dreieck zu, ansonsten dem auf das die Richtung zeit
- Polygon: Kantenzug, der Eckpunkte verbindet. Einfache Polygone: konvex, konkav - nicht einfach: nicht überlappend, überlappend. Überlappende Polygone können dabei Einschlüsse haben

Transformationen

- Koordinatensystem mit Ursprung $\underline{0}$ spannt zusammen mit Vektoren $\hat{x}$, $\hat{y}$ auf. Die Koordinatenachsen bilden eine Basis, die orthonormal ist, wenn die Vektoren normiert sind und alle orthogonal zueinander stehen
- Lineare Abbildung (vgl. Einführung in die Mathematik 2): Ursprung bleibt erhalten. Es werden praktisch die Koordinatenachsen transformiert, wobei die erste Spalte von M angibt, mit was die X-Koordinate eines Vektors transformiert wird und die zweite Spalte entsprechend für Y-Koordinate. Kann durch Matrix dargestellt werden und durch Matrix-Vektor-Multiplikation:

$$\mathbb{R}^{2\times 2} : M = \begin{pmatrix} M_{11} & M_{12} \\ M_{21} & M_{22} \end{pmatrix} = (\hat{x}' \vec{y}') = \begin{pmatrix} \vec{f}(\hat{x}) & \vec{f}(\vec{y}) \end{pmatrix}$$

- Beispiel für Transformationsmatrix M mit Rotation um den Winkel α gegen den Uhrzeigersinn:

$$M = \begin{pmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{pmatrix}$$

- Identitätstransformation entsprechend:

$$I = I^{-1} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

- Punktspiegelung (= Rotation um 180 Grad in $\mathbb{R}^2$:

$$N = \begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix}$$

- Achsenspiegelung:

$$N_x = \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix}$$

- Projektionen: Eine Koordinate wird weggelassen (2D $\rightarrow$ 1D), z. B. $\begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}$, um auf die Y-Achse zu projizieren
- Skalierung um a, b (< 0: Verkleinerung, > 1: Vergrößerung) :

$$S(a, b) = \begin{pmatrix} a & 0 \\ 0 & b \end{pmatrix}$$

- Scherung in Y-Richtung:

$$H_y(s) = \begin{pmatrix} 1 & s \\ 0 & 1 \end{pmatrix}$$

- Diese Transformationen können nach $\mathbb{R}^3$ übertragen. Bei Rotation ist dann entsprechend die Rotation um die X-Achse, die Y-Achse oder die Z-Achse möglich. Dabei bleibt jeweils die Koordinate erhalten, um die rotiert wird. Die anderen werden so rotiert, wie in $\mathbb{R}^2$.
- Modelltransformation: Positionierung von Objekten in einer Szene (vgl. Szenengraph), dies sind die bisher eingeführten Transformationen
- Systemtransformation: Umrechnung der Koordinaten in ein anderes Koordinatensystem, z. B. zur Projektion von der 3D-"Objektebene" in die 2D-Bildebene des Betrachters. Systemtransformationen sind praktisch invertierte Modelltransformationen, da die Koordinatendarstellung eines transformierten Vektors im ursprünglichen Koordinatensystem berechnet wird.
  Die Systemtransformation ergibt sich durch Invertieren der Matrix mit den transformierten Basisvektoren in den Spalten
- Vektor in Komponentendarstellung des Koordinatensystems A: $\vec{v}_A$. Systemtransformation von Koordinatensystem B ins Koordinatensystem A: M_A^B

- Modelltransformationen können über Matrixmultiplikation verkettet werden. Dabei wird die zweite Transformationsmatrix von rechts an die erste multipliziert. Daher ist die verkettete Transformation gleich die Systemtransformation vom letzten Koordinatensystem ins Anfangskoordinatensystem.
- Affine Abbildungen erweitern lineare Abbildungen um Translationen / Verschiebungen, bei denen der Ursprung des Koordinatensystems verschoben wird
- Affine Transformationen werden durch homogene Vektoren mit Matrix-Vektor-Multiplikation beschrieben, dabei wird zunächst eine zusätzliche Dimension mit 1 festgelegt
  Die Transformationsmatrix wird zu:

$$\tilde{M} = \begin{pmatrix} M_{11} & M_{12} & t_x \\ M_{21} & M_{22} & t_y \\ 0 & 0 & 1 \end{pmatrix}$$

  Dabei ist die letzte Spalte das Bild des Ursprungs
- Lineare Transformationen und Translationen ergeben somit zusammen affine Transformationen.
- Affine Kombination: Linearkombination mit mehreren Vektoren, bei denen sich die Gewichte (Faktoren) der einzelnen Vektoren auf eins aufsummieren
- Affine Invarianz: Beim Zeichnen einer zu transformierenden Kurve ist es das Gleiche, erst die Kurvenpunkte zu transformieren und dann die Kurve zu berechnen oder erst die Kurvenpunkte zu berechnen und dann die Transformation auf die einzelnen Punkte anzuwenden
- Affine Transformationen erhalten **nicht** die Winkel zwischen Vektoren. Daher müssen transformierte Normalenvektoren nach der Transformation mit der inverstransponierten Transformationsmatrix multipliziert werden: $\vec{n}' = (M^{-1})^T \hat{n}$. Anschließend muss $\vec{n}'$ normiert werden.
- Klassische Projektionen: Ebene Projektion, Parallele Projektion, Perspektivisch (1-Punkt, 2-Punkt ,3-Punkt). Parallele Projektionen können rechtwinklig oder schiefwinklig sein. Koordinatensystem wird an Hauptrichtungen eines zu zeichnenden Objektes ausgerichtet.
- Parallelprojektion: Objekt wird in xz, yz, xy-Ebene o. ä. projiziert, wobei die Information entlang der Projektionsrichtung verloren geht (eine Koordinate wird also praktisch weggelassen)
- Projiziert man nicht entlang einer der Hauptrichtungen, ist das Verhältnis der Koordinatenachsen untereinander entscheidend: Bei Isometrischer Projektion ist das Verhältnis gleich, bei dimetrischer Projektion entsprechend das Verhältnis zweier Achsen, bei trimetrischer Projektion alle verschieden
- Perspektische Projektion: Strahlen auf die Bildebene des Betrachters, z_0 ist dabei der Abstand von der Bildebene zum Projektionszentrum. Diese können mit hogener Matrizenschreibweise repräsentiert werden. Dabei wird die homogene Transformationsmatrix $\tilde{M}$ zu

$$\begin{pmatrix} z_0 & 0 & 0 & 0 \\ 0 & z_0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & z_0 \end{pmatrix}$$

  z_0 funnktioniert also als gemeinsamer Nenner. $\underline{p}$ wird um $-z$ transformiert, damit die Bildebene erreicht wird.
- Rationale Zahlen können durch einen homogenen Vektor dargestellt werden, um Quotienten zu repräsentieren (Erste Komponente des Vektors ist dann Zähler, zweite Nenner). Alle homogenen Vektoren mit einer 0 in der zweiten Komponente repräsentieren ∞, aber entweder $-\infty$ oder $+\infty$. $(0, 0)$ ist ungültig.
- Rationale Funktionen können durch homogene Matrizen dargestellt werden:

$$f(q) = \frac{aq+b}{cq+d} \rightarrowtail \begin{pmatrix} a & b \\ c & d \end{pmatrix} \approx \lambda \begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

- Erweitert man rationale Funktionen auf 2 oder 3 Dimensionen, wird nur eine w-Komponente für einen gemeinsamen Nenner hinzugefügt

$$\tilde{M} = \begin{pmatrix} M_{11} & M_{12} & t_x \\ M_{21} & M_{22} & t_y \\ p_x & p_y & 1 \end{pmatrix}$$

- Die perspektivische Projektion auf die Bildebene kann als Verkettung einer umkehrbaren perspektivischen Abbildung gefolgt von einer Projektion entlang der z-Richtung interpretiert werden. Daher wird zunächst die Z-Koordinate weggelassen und dann mit $p_z = \frac{-1}{z_0}$ transformiert
- Perspektivische Transformation am Beispiel von $\tilde{P}_y$ mit $p_y = \frac{1}{2}$: Punkte auf der x-Achse bleiben an der selben Stelle und heißen Fixpunkte
- Punkte, die zwischen Auge und x-Achse liegen, werden nach außen und unten geschoben
- Strecken können auseinander gerissen werden, wenn ein Endpunkt vor und der andere hinter dem Auge liegt
- Um zu verhindern, dass Objekte von hinter dem Auge vor das Auge gebracht werden, muss vorher an der near-clipping Ebene geclippt werden
- Punkte auf der zur x-Achse parallelen Gerade durch den Augpunkt werden auf Punkte im unendlichen abgebildet
- Hessesche Normalform für Ebenen: $n_x x + n_y y + n_z z + d = 0$ ($\tilde{n}^T \tilde{x} = 0$, wobei $\tilde{n} = \begin{pmatrix} n_x \\ n_y \\ n_z \\ d \end{pmatrix}$ mit d als Abstand zum Ursprung) Die homogene Ebenendarstellung transformiert sich mit der Inverstransponierten.
- Nach perspektivischer Abbildung können sich eigentlich parallele Linien schneiden; der Schnittpunkt ist ein Fluchtpunkt. Jede Raumrichtung kann einen anderen Fluchtpunkt haben. Man spricht von 1-, 2- oder 3-Punktperspektive, wenn es für 1, 2 oder 3 Hauptrichtungen einen Hautpfluchtpunkt gibt (Hauptfluchtpunkt für y ist dann der Fluchtpunkt, in dem sich alle parallel zur y-Achse verlaufenden Geraden schneiden
- Der Fluchtpunkt in einer Richtung $\vec{v}$ kann mit Hilfe der Abbildungsmatrix $\tilde{f}_{\vec{v}} = \tilde{P}(\vec{p})\tilde{v} mit \tilde{v} = \begin{pmatrix} \vec{v} \\ 0 \end{pmatrix}$ bestimmt werden. Dafür multipliziert man die homogene Transformationsmatrix $\tilde{M}$, wobei p_x, p_y und p_z variabel sind, mit dem RIchtungsvektor $\vec{v}$.
- In den Spalten einer homogenen Matrix stehen die homogenen Fluchtpunkte der Hauptachsen

und das homogene Bild des Ursprungs
- Lineare Transformationen: Repräsentieren Vektoren, Ortsvektoren in 2×2 Matrix im $\mathbb{R}^2$ bzw. 3×3 Matrix im $\mathbb{R}^3$. Geht für Skalierung, Spiegelung, Projektion, Rotation, Scherung. Transformation mit $\vec{v}' = M\vec{v}$, Vorsicht: Normalenvektoren mit $\vec{n}' = (M^{-1})^T \hat{n}$
- Affine Transformation: Punkte und Vektoren, homogene Matrix (Dimension + 1), zusätzlich Translation unterstützt, Punkte werden mit $\begin{pmatrix} p' \\ 1 \end{pmatrix} = \tilde{M} \begin{pmatrix} p \\ 1 \end{pmatrix}$ transformiert, Vektoren mit $\begin{pmatrix} \vec{p'} \\ 0 \end{pmatrix} = \tilde{M} \begin{pmatrix} \vec{p} \\ 0 \end{pmatrix}$ transformiert
- Perspektivische Transformation: Punkte / Ebenen werden mit homogener Matrix transformiert, zusätzlich ist perspektivische Transformation mögliche. Transformation durch:

$$\begin{pmatrix} \vec{n}' \\ d' \end{pmatrix} = (\tilde{M}^{-1})^T \begin{pmatrix} \hat{n} \\ d \end{pmatrix}$$

- Bei Rotationstransformationen und Translationen werden generell alle Größen (Längen, Winkel, Flächen, Längenverhältnisse und affine Koordinaten, Reihenfolge von Punkten, Verhältnisse von Verhältnissen, Geraden, Ebenen, Geradensegmente) erhalten, bei Scherung Längen und Winkel nicht. Bei affiner Transformation werden Längen, Winkel und Flächen nicht unbedingt erhalten. Bei perspektivischer Transformation nur Verhältnisse von Verhältnissen sowie Geraden und Ebenen.
- OpenGL: Objektkoordinaten werden über Modelview-Matrix auf Aufkoordinaten gerechnet, dann über Projektionsmatrix auf Clipkoordinaten, dann über w-Clip auf normalisierte Devicekoordinaten und schließlich über Viewport-Transformation auf Fensterkoordinaten
- OpenGL-Pipeline zur Pixelberechnung: Rechteckigen Bildausschnitt definieren, Textur- / Shaderberechnung, Alphatest, Tiefentest, Blending
- OpenGL-Matrixstapel: Alle Transformationen werden mit homogenen Matrizen dargestellt, alle Punkte mit homogenem Vektor. Transformationsmatrizen: GL_PROJECTION für Transformation von Weltkoordinaten in Bildkoordinaten, GL_MODELVIEW für Transformation von Objektkoordinaten in Kamera- oder Weltkoordinaten, GL_TEXTURE für Transformation von Texturkoordinaten
- Aktuell gewählter Matrizenstapel kann mit glLoadIdentity, glPushMatrix, glPopMatrix etc. verändert werden
- Jeder Knoten wird dann mit Modelview- *und* Projektionsmatrix in Bildkoordinaten überführt
- Dies kann z. B. genutzt werden, um einen Würfelbaum zu zeichnen, bei dem das aktuelle Koordinatensystem rotiert, verschoben und skaliert wird

Grafikprogrammierung

- Grundlage: Ereignisverarbeitung - Abfragen von Eingabegeräten, Berechnen der Welt, Zeichnen der Szene $\rightarrowtail$ Volle Auslastung des Rechners ohne Pause. Daher am Besten zwischendurch mal schlafen.
- Grafik-APIs (OpenGL) erlauben die Programmierung der GPU (Konfiguration, Ansichtsdefinition, Beleuchtung, Materialparameter, Geometrie). Die GPU generiert dann daraus ein Pixelbild (Rasterisierung)
- GLU: Hilfsbibiliothek für Bildmanipulation, Triangulierung, NURBS etc. GLUT: Fenster, Ereignisverarbeitung, Menüs, Schriften
- GPU enthält Bildpuffer für Farbe und Tiefenpuffer mit Tiefe für jedes Fragment / Bereich
- Alpha-Kanal des Farbpuffers: Transparenzwert, Double Buffer: Zweiter Bildpuffer, in den gezeichnet wird und der am Ende mit dem Bildschirmpuffer getauscht wird, Tiefenpuffer: Tiefensortierung von Fragmenten mit Tiefentest, Stencil-Puffer: Schattenberechnung o. ä., Accum-Puffer: Antialiasing, Motion-Blur
- Globaler Zustand wird gespeichert (Kontext: Zeichenfarbe, Liniendicke...). Daher entfällt Verwaltungsarbeit, dieselbe Funktion kann mit unterschliedlichem Zustand für verschiedene Aufgaben verwendet werden, der Zustand muss den Zeichenmethoden nicht übergeben werden. Aber ist nicht parallelisierbar. Es kann außerdem zu unerwünschten Ergebnissen kommen, wenn der Zustand innerhalb einer Funktion geändert und nicht wieder freigegeben wird.
- Tripel Buffering: Double Buffering kann zu Tearing führen, weil die Grafikkarte während der Pufferfüllung das Bild abschickt. Mit Tripel Buffering wird der Puffer nur nach vollständigem Abstasten des aktuellen Bildpuffers durchgeführt.
- GPUs unterstützen nur Dreiecke (oder Vierecke über zwei Dreiecke), daher werden Polygone mit Facetten approximiert (ebene Flächen)
- Flat Shading: Beleuchtung nach Form
- Gouraud Shading: Beleuchtung mit Normalen
- Phong Shading: Beleuchtung pro Pixel
- OpenGL unterstützt Strecken (GL_POINTS), Lines, Triangles, Quads, Polygon
- Für jeden Vertex können mit glColor, glNormal und glTexCoords Vertexparameter festgelegt werden. Anschließend wird mit glVertex gezeichnet
- Reihenfolge: Gegen den Uhrzeigersinn
- Dreiecks- und Viereckstriefen sind am Effizientesten zum Zeichnen von z. B. Landschaften
- Smooth Shading: Interpolation der Farbe und Normale (vgl. baryzentrische Koordinaten). Flat Shading: Nur Wert vom letzten Vertex nehmen
- Direct Mode: Nicht indizierte Spezifikation, Senden der Daten für jeden Vertex direkt an die GPU (flexibel, langsam)
- Vertex Array: Indizierte Spezifikation mit einer Indizierung möglich
- Display Lists: Speicherung zusammenhängender, mehrfach genutzter Befehle in Display Listen - Schnell in der Ausführung, langsam in der Erstellung
- Lochkamera: Bild wird durch ein Loch auf eine Projektionsfläche projiziert (auf dem Kopf). Je größer das Loch, desto unschärfer. Computergrafik: Ideales unendlich kleines Loch, alles scharf
- Lochkameramodell wird in OpenGL verwendet. Augposition ist Position des Lochs bei der Kamera. Virtuelle Bildebene wird definiert, damit Bild nicht auf dem Kopf steht. Tiefenpuffer be-

grenzt durch Clipping dargestelltes Bild.
- Kameradefinition: Bildausschnitt im Fenster in Pixelkoordinaten, Sichtvolumen relativ zum Augkoordinatensystem (Öffnungswinkel, Verhätlnis Breite / Höhe, Clipping-Bereich des Tiefenpuffers). Positionierung der Kamera im Koordinatensystem der Szene (Augpunkt, Fokus, Obenrichtung)
- Culling / Clipping: Elimination von Objekten, die nicht auf den Bildschirm projizieren
- Clipping: Verwurf der nicht im Bildschirmbereich befindlichen Objekte
- Im 3D wird an Sichtpyramide geclipped
- Sutherland-Hodgman zum Clippen eines beliebigen Polygons mit einem anderen beliebigen Polygon
- Sutherland-Cohen zum Clippen einer Linie mit einem Rechteck: Bereichen um das Bildschirmrechteck wird eine Bitmaske zugeordnet (links z. B. 1000, rechts 0100, Mitte immer 0000). Den Endpunkten der Linie wird ein Bereich zugeordnet. Haben die beiden Endpunkte ein gemeinsames Bit (logisches Und), ist die Linie außerhalb des Bildschirms und braucht nicht gezeichnet zu werden. Ist sie innerhalb des Bildschirms, sind beide Bereiche 0000. Ansonsten werden die Schnittpunkte mit dem Rechteck berechnet und die Linien entsprechend gekürzt.
- Culling: Dem Beobachter abgewandte Seiten werden nich gezeichnet; Back-Face-Culling: Abgewandte Seiten werden wegen ihres Umlaufsinns eliminiert. Kann bei Objekten mit Rand / Löchern zu Problemen führen
- Transformation in OpenGL: Rotate, Translate, Scale
- Vertigo Effect / Dolly Zoom: Gleichzeitiges Wegbewegen und Hineinzoomen
- OpenGL-Lichtquellen: Richtungslichtquellen (Globale Richtung und farbige Intensität), Punktlichtquellen (Position, farbige Intensität und Intensitätsabfall mit Distanz), gerichteter Strahler (Punktlichtquelle plus Abstrahlrichtung, Bündelung und Öffnungswinkel)
- Ambientes Licht: Unabhängig der Richtung, simuliert Streulicht in der wirklichen Welt
- Diffuses Licht: Abhängig von Orientierung der Oberfläche zur Lichtquelle (Am stärksten bei senkrechter Einstrahlung)
- Spekulares Licht: Hängt zusätzlich von Beobachterposition ab (am hellsten bei spiegelnder Reflektion). Emuliert Glanzlichter, Fokusierung über shininess-Faktor
- In OpenGL bis zu acht Lichtquellen, die parametrisiert werden
- MipMapping: Vorhalten vorgerechneter Texturen in verschiedener Auflösung, damit die Grafikkarte nicht beim Zeichnen herunterrechnen muss
- Texturfilterung: Darstellung eines sich wiederholenden Musters kann in der Ferne zu Geisterfrequenzen und Treppenartefakten führen: Bildfilterung notwendig!
- Abtasttheorem: $f_a > 2f_{\text{orig}}$
- Lineare MipMap-Filterung hilft zur Vermeidung von Geisterfrequenzen

Kurven

- Monombasis: Polynom
- Berechnung von Binomialkoeffizienten über das pascalsche Dreieck: i ist die Spalte, n die Zeile (jeweils von null, Spalte von links)
- Bernsteinbasis: $B_i^g(t) = \binom{g}{i}(1-t)^{g-i}t^i$
- Polynome können in verschiedenen Basen dargestellt werden. Die Umrechnung des Koeffienztenvektors in eine andere Basis heißt Basistransformation
- Die Transformationsmatrix für den Koeffizientenvektor ergibt sich durch Transponieren und invertieren
- Horner-Schema: Spart Multiplikationsschritte (vorher: $2n-1$, nachher: n). Dafür wird das Polynom $a + bx + cx_2$ so gerechnet: $a + x(b + x(c + x))$
- Kontrollpunkte definieren den Kurvenverlauf, die Kontrollpunkte können dabei verschieden gewichtet werden
- Basis: Der Einfluss der verschiedenen Kontrollpunkte wird durch polynomiale Basisfunktion gesteuert
- Einfachste Kurve: Verbindung zwischen zwei Punkten
- Splines: Glatter Übergang aus mehreren Kurvensegmenten
- Bezier-Kurven:

$$\sum_{i=0}^{g} \underline{b}_i B_i^g(t)$$

  Dabei ist B die Bernsteinbasis
- Kontrollpunkte können auch als Kontrollpunktmatrix gegeben werden (zeilenweise die Basen, oben für $i = 0$)
- Endtangenteninterpolation: Tangente der Kurvenendpunkte entspricht der Tangente an den Endsegmenten des Kontrollpolygons
- Konvexe-Hüllen-Eigenschaft: Kurvenverlauf nur innerhalb der konvexen Hülle der Kontrollpunkte
- Matrixdarstellung: Kontrollpunkte werden als Matrix dargestellt. Für Bezierkurve:

$$\underline{c}(t) = \sum_{i=0}^{g} \underline{b}_i B_i^g(t) =$$

$$\begin{pmatrix} b_{0,x} & b1,x & b2,x & b3,x \\ b_{0,y} & b1,y & b2,y & b3,y \end{pmatrix} \begin{pmatrix} (1-t)^3 \\ 3(1-t)^2 t \\ 3(1-t)t^2 \\ t^3 \end{pmatrix}$$

  Kontrollpunkte werden also spaltenweise genommen.
- Beziér-Kurven dritten Grades bestehen aus 4 definierten Kontrollpunkten: Startpunkt $\underline{b_0}$, Endpunkt $\underline{b_3}$, Tangente des Startpunktes $\underline{b_1}$, Tangente des Endpunktes $\underline{b_2}$
- De-Casteljau: Festlegung eines Faktors t (z. B. 0.2). Dann Verbinden der Punkte, die bei den einzelnen Verbindungen der Kontrollpunkte bei 0.2 markiert werden. Der Punkte, der als letztes markiert wird, liegt dann auf der Beziérkurve.
- Lagrange-Kurve: Kurve geht durch alle Kontrollpunkte (Interpolationseigenschaft)

$$L_i^g(t) = \prod_{0=k\neq i}^{g} \frac{t-u_k}{u_i-u_k}$$

- Bei vielen Knoten kommt es bei Lagrange-Kurven zu Überschwingungen

- Affine Invarianz (s. o.): Die Basisfunktionen summieren sich für alle t zu 1
- Konvexe Hülleneigenschaft für Lagrange-Kurven nicht erfüllt
- Hermite Interpolation: Interpolation von Positions- und Tangenteninformation
- Mit wachsendem Grad folgen Bezierkurven immer weniger den Kontrollpunkten → Splines
- Je mehr Ableitungen an Nahtstellen gleich sind, desto besser
- Parametrisch stetig: Die ersten k Ableitungen sind nach dem Parameter identisch; Geometrisch stetig: Die ersten k Ableitungen sind nach der Kurvenlänge identisch.
- Geschlossene Splines: Splines bilden einen geschlossenen Polygonzug, offene entsprechend nicht
- B-Splines werden mit der rekursiven Konstruktionsformel nach Cox und de Boor definiert
- Bei einem uniformen Spline sind die Abstände zwischen zwei Knoten immer gleich
- Basis-Splines: i: Laufindex, n: Segmentanzahl, g: Grad der Kurvensegmente, $k = g - 1$: Stetigkeit der Kurvensegmente, $K = n + g$ oder n: Anzahl Kontrollpunkte, $m = K + g + 1$: Anzahl Einträge im Knotenvektor
- Rekursion nach Cox De Boor bei offenen Splinebasen: $N_i^0(t) = 1$, falls $u_i \leq t \leq u_{i+1}$ und $u_i < u_{i+1}$
- Wählt man aufeinander folgende Knoten gleich, so bekommt der entstandene Knoten eine Multiplizität μ. Bei $\mu = g + 1$ bekommt der Spline einen Sprung. Dies ist nur an den Endpunkten bei offenen Splines erlaubt.
- $$\sum_{i=0}^{K-1} \underline{d_i} N_i^g(t)$$
- Splines liegen in der Kombination der konvexen Hüllen, wenn je drei Kontrollpunkte verbunden werden
- Die Basisfunktion N_i^g bzw. der De-Boor-Punkt d_i beeinflusst den Spline im Parameterbereich $[u_i, u_{i+g+1})$
- Alle Kurven bis auf Hermite affin invariant
- Bernstein-Basis für Bezier-Kurven positiv. Approximierend, Konvexhülleneigenschaft, Endtangenteninterpolation
- Lagrange-Basis: Kontrollpunktinterpolation (Kontrollpunkte liegen auf der Kurve), unabhängig
- Hermite-Basis: Kontrolltangenteninterpolation, lokaler Kontrolltangenteneinfluss, C^1-stetig
- Natürliche Basis (Splines): Positiv, lokale Definition, Endtangenteninterpolation, lokaler Kontrollpunkteinfluss, Konvexhülleneigenschaft, C^{g-1}-stetig

Beleuchtung

- Licht besteht aus Überlagerung von Photonen (Wellenmodell)
- Sichtbares Licht: ca. 400 bis 800 nm Wellenlänge
- Wellenlänge, Frequenz, Energie (definieren Farbe), Polarisation (linear, zirkulär)
- Laser: Licht einer Wellenlänge, bei anderen Lichtquellen verschiedene Farben
- Leistungsspektrum: Ähnlich zu Histogramm bei Bildern, Darstellung welcher Anteil der Lichtquelle welche Wellenlänge hat
- Die Farben von Gegenständen sind relativ zur Beleuchtung (rot angestrahlt: sieht rot aus (oh))
- Emission: Lichtquelle strahlt Licht ab, Transport: Im Vakuum bewegen sich Photonen auf geraden Bahnen, Streuung: in transparenten Materialien (Glas) prallen Photonen an Atomen ab, Reflektion: Reflektion an Oberflächen, Brechung: z. B. Glas lässt Photonen durch (aber in anderem Winkel), Absorption: in transparenten Medien und an Oberflächen werden Photonen absorbiert, Detektion: im Auge oder Photosensor werden Photonen gezählt
- Strahlungsgleichgewicht innerhalb einer Szenerie, da genauso viele Photonen "reinkommen" wie "rausgehen"
- Lichtleistung in einem Punkt p in Richtung ω gibt Photonen an, die gerichtete Lichtleistung nennt man auch die Strahldichte (Menschliche Wahrnehmung: Leuchtdichte)
- Beide Kenngrößen sind im Vakuum entlang von Strahlen konstant (da entsprechend keine Photonen verloren gehen)
- Bei Beleuchtungssimulation wird Streuung häufig weggelassen (nur Berechnung von Emission, Reflektionen, Absorptionen, Detektion)
- Light Tracing: Verfolgen von Photonenpfaden ausgehend von der Lichtquelle, Visibility Tracing: Verfolgen von Photonenpfaden in umgekehrter Richtung
- Menschliches Auge hat 3 Farbrezeptoren (R, G, B), vgl. Zapfen bei EMI, Notation: $\ddot{\vec{L}} = (R, G, B)$, bei Simulation lohnt sich aber Betrachtung von mehr Farben (verschiedenen Wellenlängen), da sich diese bei Streuung und Reflektion anders verhalten
- LEDs, Energiesparlampen, Glühbirnen, organische LEDs als Flächenlichtquellen
- Lichtleistung: $\frac{\text{Wahrgenommene Lichtleistung}}{\text{Zugeführte Leistung}}$. Grünes Licht (555 nm) wird am stärksten wahrgenommen
- Richtungslichtquellen: Einfachste Lichtquelle, Lichtstrahlen fallen angenährt parallel ein, Richtung ω als Richtung **zur** Lichtquelle und Lichtausstrahlung M (Leistung pro Fläche)
- Punktlichtquelle: Sendet Licht in alle Richtungen mit gleicher Stärke aus (isotrop), definiert durch Position, Lichtstärke I (in Candela) als Leistung pro Richtung
- Weit entfernte Punktlichtquelle verhält sich wie Richtungslichtquelle
- Goniometrische Lichtquelle: Lichtstärke abhängig von Abstrahlrichtung
- Lichtstrahler (Spot): Richtungsabhängigkeit, die Lichtkegel erzeugt:

  $$\ddot{\vec{I}}(\phi, o) = \ddot{\vec{I}}_0 * (coso)^n$$

  Dies gilt nur, falls $|o| < o_{\max}$. $o_{\max}$ ist der Öffnungswinkel. Strahler sind im OpenGL-Standard.
- Flächenlichtquellen: Ausgedehnte Lichtquellen, wird durch Oberfläche und Abstrahlleistung pro Richtung und Fläche definiert. Ist oft isotrop (d. h. an allen Oberflächenpunkten in alle Richtungen gleich hell). Schwer zu simulieren, wird daher durch viele Punktquellen approximiert. Erzeugt weiche Schatten.
- Lokale Beleuchtungsmodelle beschreiben Reflektions- und Absorptionsverhalten von Licht

an Oberflächen

- Globale Beleuchtungsmodelle simulieren die Lichtausbreitung über mehrere Reflektionen etc.
- Echtzeitgrafik (OpenGL etc.) benutzt lokale Beleuchtungsmodelle und nährt andere Effekte nur an
- BRDF (Bidirektionale Reflektionsverteilungsfunktion): Richtung ω_{in} zur Lichtquelle, ω_{out}, einfallende Strahldichte L_{in}, mehrere Parameter r_α wie Reflektionsfarbe, Rauigkeit. Strahldichte L_{out}, das zum Beobachter reflektiert wird.
- Physikalisch richtig ist bei einer Kugel Integration der BRDF über eine Halbkugel
- Bei Punkt- und Richtungslichtquellen wird dann das einfallende Licht auf ein Objekt die Summe aller sichtbaren Lichtquellen
- Bei Betrachtung nur einer Lichtquelle ist die BRDF ein lokales Beleuchtungsmodell
- Diffuse Reflektion: Anteil hängt nicht von Beobachter ab. Wird in alle Richtungen gleich reflektiert.
- Gerichtet diffus (spekular): Am stärksten an der an der Normale gespiegelten Einfallsrichtung des Lichtes
- Spiegelnd: Nur ein Betrag, wenn der Betrachter genau aus der Einfallsrichtung des Lichts guckt
- In allen lokalen Beleuchtungsmodellen wird der diffuse Anteil gleich modelliert. Die BRDF wird als konstant angenommen und auf eine konstante Farbe gesetzt.
- $\otimes$ ist die komponentenweise Multiplikation zweier Vektoren
- Diffuse Reflektion kann berechnet werden mit: $\overset{\cdots}{L}_{\text{out}} = \overset{\cdot\cdot}{r}_{\text{diffuse}} \otimes \overset{\cdots}{L}_{\text{in}} \cos_+ \alpha$. Dies entspricht $\overset{\cdots}{L}_{\text{out}} = \overset{\cdot\cdot}{r}_{\text{diffuse}} \otimes \overset{\cdots}{L}_{\text{in}} \langle \hat{\omega_{\text{in}}}, \hat{n} \rangle_+$.
- Phong- und Blinn-Phong-Modell zur Lichtberechnung (Blinn-Phong ist Standard in OpenGL).
- Glattheit wird durch Exponenten m definiert (shininess). Je größer m desto fokussierter die Highlights.

$$\overset{\cdots}{L}_{\text{out (diff)}} = \overset{\cdot\cdot}{r}_{\text{diffuse}} \otimes \overset{\cdots}{L}_{\text{in}} \langle \hat{\omega_{\text{in}}}, \hat{n} \rangle_+$$

Dazu kommt der spekulare Anteil, falls $\langle \hat{\omega}_{\text{in}}, \hat{n} \rangle \geq 0$ gilt:

Phong:

$$\overset{\cdots}{r}_{\text{specular}} \otimes \overset{\cdots}{L}_{\text{in}} * \langle \hat{\omega}_{\text{out}}, \hat{\omega}_{\text{refl}} \rangle^m$$

Blinn-Phong:

$$\overset{\cdots}{r}_{\text{specular}} \otimes \overset{\cdots}{L}_{\text{in}} * \langle \frac{\hat{w}_{\text{in}} + \hat{w}_{\text{out}}}{||\hat{w}_{\text{in}} + \hat{w}_{\text{out}}||}, \hat{n} \rangle^m$$

- Liegen der Normalenvektor, die einfallende Lichtrichtung und das ausgehende Licht in einer Ebene, liefern Blinn und Blinn-Phong ungefähr das selbe Ergebnis
- Blinn-Phong-Modell ist realistischer (Abweichung Blinn vs Blinn-Phong steigt, je weiter die drei Vektoren nicht in einer Ebene liegen)
- Brechung nach dem Gesetz nach Snellnius: Brechungsindizes zweier benachbarter Materialien (n_1, n_2) werden genommen. Je höher der Brechungsindex, desto langsamer breiten sich Photonen aus. Beim Übergang in ein Medium mit kleinerem Brechungsindex gibt es einen Grenzwinkel, ab dem das gesamte Licht reflektiert wird.

$$n_1 \sin\theta_1 = n_2 \sin\theta_2$$

- Ambientes Licht approximiert Streulicht und wird einfach auf den diffusen und spekularen Anteil addiert
- Ambient Occlusion: Für jeden Oberflächenpunkt wird Winkel bestimmt, aus dem die Oberfläche ungehindert beleuchtet werden kann. Dadurch kann die ambiente Beleuchtung realistischer approximiert werden.
- Subsurface Scattering: Durchscheinende Materialien verursachen starke Streuung und Absorption. Approximation dieser Effekte nur nahe der Oberfläche mit einem diffusen Transmissionsmodell
- Kaustiken: Gekrümmte transparente Objekte können Licht bündeln und Kaustiken (fokussiertes Licht) erzeugen (Lupe / Glas)
- Radiosity: Alle Lichtquellen und Oberflächen werden als diffus angenommen und sind somit unabhängig vom Betrachter. Das Bild wird also vor der Darstellung der Szene berechnet und unterstützt diffuse Reflektion, Streuung, weiche Schatten und Farbbluten, aber keine gerichtete diffuse Reflektion (Spekulare Reflektion), Spiegelung, Brechung, Kaustiken
- Raytracing: Verfolgen von Lichtstrahlen vom Auge in die Szene. Unterstützt Spiegelung und Brechung und kann durch Abwandlung von Simulationsparametern und mehrfachem Verfolgen von Lichtstrahlen Effekte wie Motion Blur, Tiefenunschärfe und weiche Schatten erzeugen. Unterstützt diffuse Reflektionen und Kaustiken, ist aber noch nicht wirklich echtzeitfähig
- Bidirektionales Pathtracing: Zufällige Generierung von Lichtstrahlen vom und zum Betrachter mit Verknüpfung dieser Strahlen. Unterstützt alle Effekte (Streuung in transparenten Medien meist nicht unterstützt), ist aber nicht echtzeitfähig.

Raytracing

- Für jeden Pixel der Kamera wird ein Lichtstrahl in die Szene geschickt (Primärstrahlen)
- Schnittpunkte der Primärstrahlen werden berechnet, gibt es keinen, wird die Hintergrundfarbe angenommen. Anschließend wird für alle Lichtquellen, deren Licht den Schnittpunkt erreicht, das reflektierte Licht addiert. Kann Licht an der Oberfläche brechen, wird ein sekundärer Lichtstrahl ausgesandt, dessen Farbwert dann auf den des Primärstrahls aufaddiert wird.
- Supersampling: Um Treppenartefakte zu vermeiden (Aliasing), kann das Bild in höherer Auflösung gerendert werden und wird dann gefiltert. Oder Pixel werden mehrfach abgetastet: Gitter / Zufall / Jitter (zufällig pro Gitterzelle)
- Adaptives Sampling: Anfangen mit großem Grundraster, an den Ecken der Gitterzellen Strahlen aussenden. Dort, wo Farbwerte an den Seiten sich deutlich unterscheiden, wird unterteilt.
- Sekundärstrahlen werden bei einfallendem Licht $\vec{v}$ nach $\vec{v} - 2\langle \hat{n}, \vec{v} \rangle \hat{n}$ berechnet
- Durch numerische Ungenauigkeiten kommt es bei Sekundärstrahlen zum erneuten Schnitt mit der Oberfläche kommen, von der der Strahl

eigentlich reflektiert wurde. Dafür kann ein globales ϵ gewählt werden, das als Schwellenwert dient

- Bei jedem Schnitt eines Strahls mit einer Oberfläche wird vom Schnittpunkt aus ein Strahl zu jeder Lichtquelle geschickt und geprüft, ob ein anderes Objekt dazwischen liegt (d. h., ob das Licht dieser Lichtquelle betrachtet werden muss). Daher werden zwei Schnittfunktionen implementiert: Schnittprüfung mit beliebigem Schnittpunkt und Schnittprüfung mit Rückgabe des Schnittpunktes.
- Materialparameter (Diffuse Reflektion, Spekulare Reflektion) werden erweitert durch skalaren Refraktionskoeffizienten und einen Brechungsindex
- Schattenfühler können statt allgemeiner Existenz eines blockierenden Objektes auch zurückgeben, wieviel Licht durchgelassen wird. Dafür wird die Länge des Durchgangs durch das Objekt mit dem Absorptionsfaktor des Objekts multipliziert. Zusätzlich wird zweimal der Transmissionskoeffizient multipliziert (Nur Approximation, da Brechung des Strahls nicht berücksichtigt wird)
- $\#_{\text{rays}} = O(w * h * s * (1 + m) * 2^{d+1})$, d: Maximale Rekursionstiefe, s: Supersampling-Faktor, n: Anzahl Objekte in der Szene, m: Anzahl Lichtquellen
- Anzahl Berechnungen: $\#_{\text{rays}}$ multipliziert mit der Anzahl der Objekte in der Szene
- Maximale Rekursionstiefe (Sekundärstrahlen) wird limitiert und Beschleunigerstrukturen verwendet, dadurch $\#_{\text{comp}} = O(w * h * s * m * \log n)$
- Distribution Raytracing: Beim Supersampling werden einzelne Abtastpunkte verwendet, um weiche Schatten, Motion Blur etc. zu erzeugen. Dazu werden die entsprechenden Parameter (Position auf der Flächenlichtquelle, Zeitpunkt, Position auf der Blendenöffnung) gewählt und mitgerechnet
- Motion Blur: Für jeden Strahl wird zufällig ein Zeitpunkt in dem gewählten Zeitinterval ausgewählt, Bewegte Objekte müssen pro Strahl neu positioniert werden!
- Tiefenunschärfe: Abstand zum Fokus der virtuellen Linse kleiner als eigentliche Distanz
- Diffuse Spiegelung: Lichstrahlen werden an glatter Oberfläche reflektiert, entsteht verwaschene Reflektion
- Hesssche Normalform:

$$\langle \hat{n}, \underline{x} \rangle = d$$

- Strahl-Pritimitv-Berechnung: Parametrische Form des Strahls $\underline{p} + t\vec{v}$ wird in implizite Darstellung des Primitivs eingesetzt und berechnet den Strahlparameter. Beispiel Ebene:

$$\langle \hat{n}, \underline{p} + t_* \vec{v} \rangle = d$$

- Keinen, einen oder viele Schnittpunkte (Bei Ebene keinen oder einen). Der mit kleinstem t (kleinstem Abstand) wird herausgesucht und die Normale am Schnittpunkt berechnet.
- Beim Schnitt von Strahl mit Kugel geht man genauso vor, ebenso
- Quader sind wichtig für Hierachie von Hüllvolumen. Idee: Aufspalten in einzelne Ebenen (Ebenenstreifen)
- Achsenparallele Ebenenstreifen: Fallunterscheidung für Parallelität
- Schnittberechnung: Bei Schnitt eines Strahls mit einem Körper schneidet der Strahl den Körper über einem Interval
- Achsenparallele Quader: Berechne Schnittinterval $T_{x/y/z}$ mit den drei Ebenenstreifen, $T_Q = T_X \cap T_Y \cap T_Z$
- Normale ergibt sich entlang einer einer Koordinatenachse
- Schnittberechnung mit Dreieck: Schnitt mit Ebene testen und dann baryzentrische Koordinaten testen
- Unterteile Welt in regelmäßiges Voxelgitter zur Beschleunigung und markiere Voxel, in denen Objekte enthalten sind
- Traversierung: Voxel entlang des Strahls verfolgen
- Gitterauflösung ist dabei entscheidend für Qualität des Ergebnisses, sehr kleine Objekte kommen nicht gut. Mögliche Verbesserung durch Schachtelung des Gitters.
- Mailbox-Technik: Wenn Objekte von mehreren umgebundenen Volumen referenziert werden können (Gitter), jeder Strahl kriegt eine eindeutige ID und jedes Objekt eine "Mailbox" mit letzter Schnittberechnung (ID, Schnittinfo). Strahle können so mehr oder weniger gecached werden.
- Hüllenvolumen: Bounding Volumes (Quader, die das Objekt einschließen) werden zu jedem Objekt erstellt und der Schnitt damit zunächst geprüft (schneller als Schnittprüfung mit dem komplexen Objekt).
- Hüllvolumen-Hierarchie: Umschließe die Objekte in einer Hierarchie von sich überlappenden Hüllvolumen (BVH)
- Kompromiss zwischen einfachem Hüllenvolumen (Quader, Kugel etc) mit viel Trefferzahlen und enganliegender Hülle (konvexe Hülle) mit hohen Schnittkosten
- k-Drops ergeben guten Kompromiss (6-DOP: Würfel, 14-DOP: Fußballähnliches Gebilde aus Rechtecken)
- Aufbau Hüllvolumen: Umschließe alle Objekte der Szene mit axis aligned Bounding Box (AABB) der Szene, spalte Box entlang der Koordinatenrichtung und verteile Objekte möglichst gleichmäßig. Dann Kindboxen so berechnen, dass sie die Objekte ganz umschließen (dadurch überlagern sich Boxen). Für nächste Koordinatenrichtung wiederholen, bis Anzahl der Objekte in den Kindboxen klein genug ist.
- Traversierung: Baum aufbauen (s. o.), $t_m = \infty$ setzen und Baum dann so sortieren dass zuerst AABB geprüft wird, die als erstes getroffen wird. Beim Erreichen von Blättern am Baum (Objekte) wird Schnitt berechnet und t_m aktualisiert
- KD-Baum: Berechne für jeden Körper AABB und damit AABB der ganzen Szene. AABB der Szene rekursiv an einer achsenparallelen Ebene schneiden (Mögliche Optimierung der Splits: Position der Schnittebene, eine von den Achsen). Optimiere nach Kosten für den Schnitt mit einem zufälligen Strahl.
- Pro Knoten: Split entscheiden (Ob, Wo). Kosten beachten: Primitve mit Schnittkosten N, Kosten für Traversierung eines inneren Knotens t_t, Wahrscheinlichkeit, dass Strahl, der den Knoten schneidet, auch einen Kindknoten

schneidet: p_a.

- KD-Baum Optimierung mit SAH (Surface Area Heuristic). Wahrscheinlichkeit, dass Strahl, der Knoten schneidet, auch Kindknoten schneidet, wird mit $\frac{A(\text{Kindknoten})}{A(\text{Knoten})}$ bestimmt.