JOHANNES KEPLER UNIVERSITÄT LINZ | JKU

Grundlagen der Computergraphik

364.012 VO Computergraphik
Sommersemester 2010

Eigene Skriptausarbeitung

Institute of Computer Graphics

Inhaltsverzeichnis

1.Transformations and Projections.....3

2.Depth-Handling and Rasterization.....6

3.Local Shading and Illumination.....12

4.Texturemapping.....16

5.Global Illumination.....19

5.1.Raytracing.....19

5.2.Radiosity.....22

6.Graphics Pipelines and Scene Graphs.....27

7.Human Vision and Basic Displays.....30

8.Curves and Surfaces.....34

9.Basics of Animation.....37

Name	**Kapitel**
Barbara Wakolbinger	2 – 3 – 5.2
Johanna Strasser	1 – 6 – 8 – 9
Markus Jäger	4 – 5.1 – 7

Sämtliche Abbildungen entstammen den vorgetragenen Vorlesungsfolien von Dr. Oliver Bimber, Institute of Computergraphics, JKU Linz

Wir garantieren für keine Richtigkeit der Ausarbeitung - Fehler inbegriffen!

1. Transformations and Projections

2D- und 3D-Oberflächen werden normalerweise durch ein Netz von Dreiecken dargestellt. Diese Netze werden durch Ecken (vertices) und ihren Verbindungen/Kanten (edges) definiert.
Ecken und Dreiecke besitzen verschiedene Eigenschaften: Position, Normalvektor, Farbe, Texturkoordinaten, etc.

Transformationen in 2D

Skalieren: s_x und s_y sind die Skalierfaktoren $\quad scale(s_x, s_y)=\begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix}$

Verzerren: Vertikale und horizontale Kanten bleiben parallel $\quad shear_x(s)=\begin{bmatrix} 1 & s \\ 0 & 1 \end{bmatrix} shear_y(s)=\begin{bmatrix} 1 & 0 \\ s & 1 \end{bmatrix}$

Drehen: Objekt wird um den Ursprung gedreht $\quad rotate(\phi)=\begin{bmatrix} \cos(\phi) & -\sin(\phi) \\ \sin(\phi) & \cos(\phi) \end{bmatrix}$

Reflektieren: um die x- oder y-Achse $\quad reflect_x=\begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} reflect_y=\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$

Um Veränderungen zu kombinieren, werden die Matrizen multipliziert. Wichtig ist die Reihenfolge (Assoziativität ist wichtig)! Jede komponierte Matrix kann via „Singular Value Decomposition“ (SVD) in ein Produkt von rotieren*skalieren*rotieren zerlegt werden.

Verschiebung in 2D

$$\begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} x \\ y \\ 1 \end{bmatrix}=\begin{bmatrix} x+t_x \\ y+t_y \\ 1 \end{bmatrix}$$

Um eine Verschiebung zu erreichen, ist eine Addition zur x- und/oder y-Achse notwendig. Aufgrund den Rechenregeln von Matrizen ist dies nur mit einer weiteren Dimension möglich!
Auch die Matrizen für die vorhergehenden Veränderungen müssen um eine Dimension erweitert werden!
Die sogenannte *„Rigid-body transforms“* (Transformation starrer Körper) enthält nur Rotation und Verschiebung des Objekts.

Viewport Transformation

Auch Fenstertransformation genannt.
Model zum Ursprung verschieben - Skalieren zu neuer Dimension - Zur neuen Position verschieben. Ist im Prinzip eine einfache Zusammensetzung von Skalierung und Verschiebung.

Basis-Transformationen in 3D

alle Formeln auf den Folien (Transformations and Projections, Folie 10).

$$scale(s_x, s_y, s_z)=\begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix} shear(s_x, s_y, s_z)=\begin{bmatrix} 1 & s_y & s_z \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} rotate_x(\phi)=\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\phi) & -\sin(\phi) \\ 0 & \sin(\phi) & \cos(\phi) \end{bmatrix} usw...$$

Für die Verschiebung wird hier eine 4x4 Matrix benötigt.

Normalvektor-Transformation

Vektoren sind Eigenschaften von Ecken und Dreiecken und beschreiben Richtungen, nicht Positionen.
Sie können nicht mit unserem Modell verschoben werden, weil sie vielleicht nach der Transformation nicht rechtwinkelig zu ihrer Ebene verbleiben. Zur korrekten Transformation von Richtungsvektoren, wie dem Normalvektor, benötigen wir die transponierte Inverse von M: $N=(M^{-1})^T$

Richtungsvektor-Transformation

Im Gegensatz zu Positionsvektoren soll die Verschiebung von Richtungsvektoren verhindert werden. Hierfür wird die 4. Koordinate 1 oder 0 gesetzt. Für die Verschiebung des Positionsvektors auf 1, für den Richtungsvektor auf 0 setzen.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & t_x \\ a_{21} & a_{22} & a_{23} & t_y \\ a_{31} & a_{32} & a_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} x \\ y \\ z \\ \boxed{1} \end{bmatrix} \text{position vector}$$

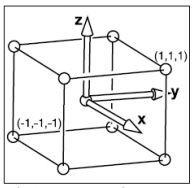
$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & t_x \\ a_{21} & a_{22} & a_{23} & t_y \\ a_{31} & a_{32} & a_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} x \\ y \\ z \\ \boxed{0} \end{bmatrix} \text{direction vector}$$

Diese 4. Koordinate nennen wir auch homogene Koordinate.

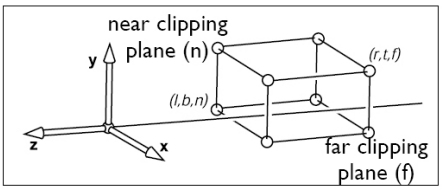
Inverse Transformation:
Die Inverse jeder Transformationsmatrix kann numerisch berechnet werden.
Matrixkompositionen können invertiert werden durch das Invertieren ihrer einzelnen Komponenten (umgekehrte Reihenfolge der Multiplikation). $M = M_1 M_2 \dots M_n \rightarrow M^{-1} = M_n^{-1} \dots M_2^{-1} M_1^{-1}$

Projektionen
Die transformierten 3D-Modelle müssen auf einen 2D-Bildschirm. Folgende Techniken erklären die 3D-zu-2D-Transformation.

Orthographische Projektion
Auch bekannt als „Parallel Projektion“.
Angenommen die Kamera blickt entlang der -z-Achse: Nun werden die Ecken im benutzerdefinierten Volumen in eine normalisierte Ansicht transformiert (kanonisches Ansichtsvolumen mit Grenzen: (-1, -1, -1) und (1, 1, 1))
Die z-Komponente wird nun ignoriert. Die x- und y-Komponenten werden in den Pixelraum transformiert (0,0 – n_x, n_y)

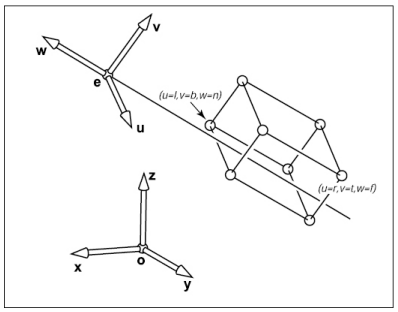


Canonical viewing volume with given boundaries (-1,-1,-1 and 1,1,1)



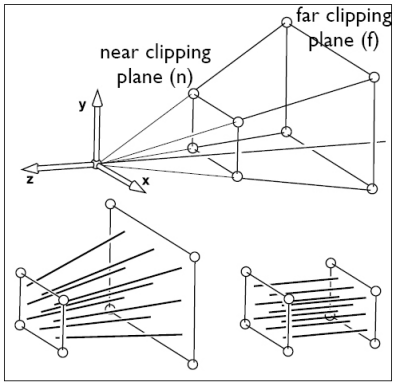
Orthographic viewing volume with defined boundaries (l,b,n and r,t,f)

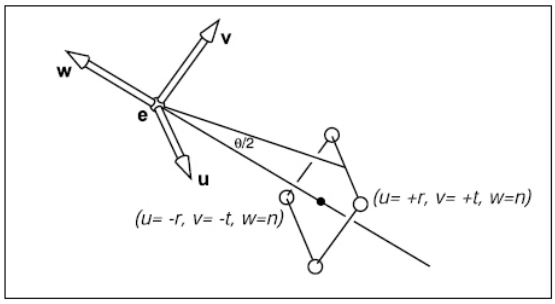
View Transformation
Eine zusätzliche Transformation ist notwendig, wenn die Kamera nicht entlang der -z-Achse schaut: Das Kamerakoordinatensystem (E) muss auf das Weltkoordinatensystem (O) abgebildet werden.
Die Kamera wird definiert durch: ihre Position (e), look-at – Vektor (g) und view-up – Vektor (t). Mit den Kamerakoordinaten (e als Ausgangspunkt) kann die „rigid-body“ - Transformation berechnet werden.



Die Transformations - Pipeline bildet also die 3D-Koordinaten (v) auf Pixel (p) ab: $p_s = M_{vp} M_o M_v M_s v_{obj}$

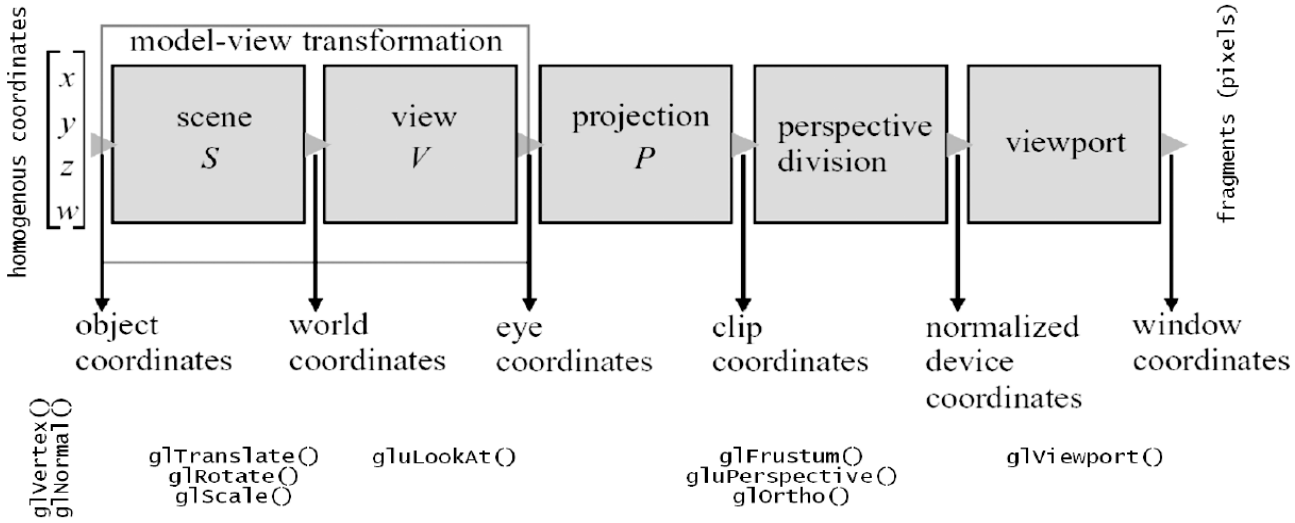
Perspektivische Projektion
Hierbei ist die Größe der Objekte am Bildschirm proportional zur ihrer Entfernung. Ecken werden zum Kamerazentrum projiziert und erscheinen wo sich die Sichtlinie mit der Betrachtungsebene schneidet.
Die perspektivische Transformation bildet jede Linie durch den Ursprung (Kamerazentrum) auf eine Linie parallel zur z-Achse ab, ohne den Punkt zu verschieben.
Eine perspektivische Projektion wird erreicht, wenn nach der perspektivischen Transformation eine orthographische Transformation durchgeführt wird.
Nach der Transformation müssen die Koordinaten skaliert werden. Mittels der Multiplikation von der „perspective transform matrix“ und den Koordinaten des Objekts kann die homogene Koordinate (4. Koordinate) einen beliebigen Wert erreichen. Dividiert man nun die 3 errechneten Koordinaten durch die homogene Koordinate, nennt man das Homogenisierung bzw. „Perspective Division“.
Für den orthographischen Fall wird die z-Komponente nicht genutzt, sie wird genutzt für Entfernung von versteckten Oberflächen, da es die Tiefenreihenfolge bewahrt.
Die Projektionsmatrix ergibt sich aus der perspektivischen Transformation gefolgt von einer orthographischen Projektion. Daraus ergibt sich folgende Änderung in der Transformationspipeline:
$p_s = M_{vp}(M_{pp} M_v M_s v_{obj})/h$





Für eine symmetrische (auf der Achse) perspektivische Projektion können die Projektionsparameter vereinfacht werden und ein Feld des Blickwinkels definiert werden, bei der Annahme eines quadratischen „pixel footprints“.

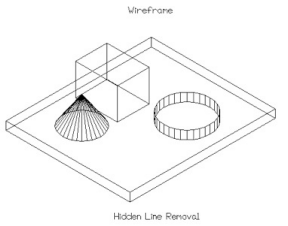
OpenGL's Transformation Pipeline



<u>Beispielfragen</u>:

- Wo liegt der Unterschied bei der Transformation von Vertices und Normalen?
  Detailfrage: Wie werden Normale richtig transformiert? Wie sieht die Matrix für Normale aus?
- Wie kann man effizient die Transformationsmatrix und deren Inverse berechnen?
- Wie sieht die Transformationspipeline aus, wie funktioniert sie?
- Erklären Sie (im Detail) die Komponenten der Transformationspipeline in OpenGL.
- Erklären Sie (in aller Kürze), was der Unterschied bei der Transformation von 3D Punkten und Normalen Vektoren ist.
- Erläutern Sie (in aller Kürze), warum die Transformationspipeline auf 4x4 Matrizen basiert und nicht auf 3x3 Matrizen.

2. Depth-Handling and Rasterization



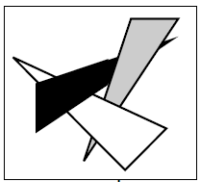
Hidden Surface Removal

- Drahtgitter-Modell (Wire-Frame Model): Dreiecke, die ohne Berücksichtigung der Reihenfolge ihrer Tiefe (Depth) gerendert werden, wodurch keine Oberflächen verborgen werden!

- Oberflächen-Modell (Surface Model): Dreiecke, gerendert unter der Berücksichtigung der Tiefen-Reihenfolge, wodurch Oberflächen im Vordergrund solche im Hintergrund verdecken

Bestimmung der Tiefen-Reihenfolge anhand der Tiefe relativ zur Kameraposition

Painter's Algorithm



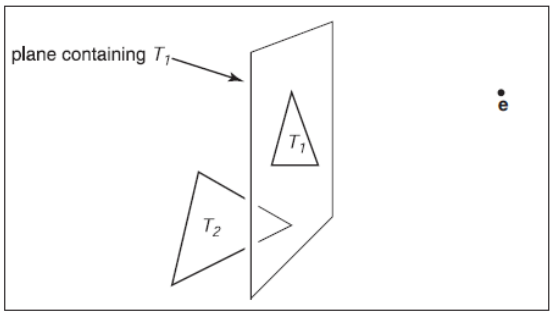
Voraussetzung ist, dass die Tiefen-Ordnung bekannt ist. Erst müssen die Oberflächen vorsortiert werden (anhand ihrer Tiefen), anschließend Rendern von Hinten nach Vorne.
Problem**:** wenn die relative Reihenfolge nicht klar definiert ist, also eine globale Back-To-Front Sortierung nicht möglich ist (Zyklen)

BSP-Trees (Binary Space Partitioning Trees)

Voraussetzung ist, dass sich keine Oberflächen kreuzen (die Ebenen (Planes) kreuzen sich nicht). Schaffung der Voraussetzung mittels eines Pre-Processing-Schrittes: Bei Kreuzung einer Ebene wird das betroffene Dreieck in mehrere zerschnitten, somit befinden sich die neuen Dreiecke jeweils eindeutig auf einer Seite der vorher durchkreuzten Ebene.
Die 3 Knoten eines Dreiecks befinden sich immer auf einer Ebene (eine Ebene ist stets durch 3 Punkte definiert!).

Implizite Ebenen-Gleichung: $f(p) = ((b-a) \times (c-a)) \cdot (p-a) = 0$
a, b, c ... Punkte auf der Ebene
sagt aus: Der Kosinus (Punktprodukt) aus Ebenen-Normale (Kreuzprodukt) und Vektor auf der Ebene (p-a) muss 0 sein (für Punkte auf der Ebene).

f(p) = 0 Punkt ist auf Ebene
$f(p^+)>0$ Punkt ist auf einer Seite der Ebene
(solche Punkte gekennzeichnet als p^+)
$f(p^-)<0$ Punkt ist auf der anderen Seite der Ebene
(solche Punkte gekennzeichnet als p^-)

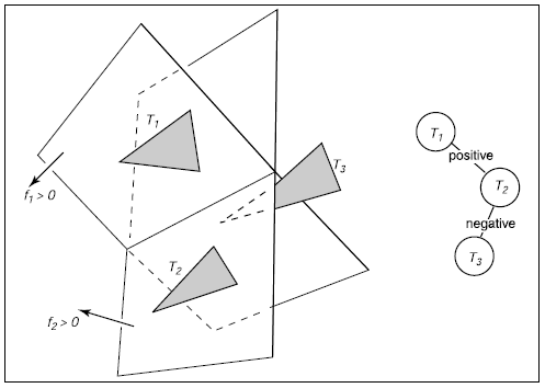


Die Dreiecke T1 und T2 haben je eine Ebenen-Gleichung (f1 und f2). Annahme: alle Punkte von T2 befinden sich auf der Seite f1(p)<0 von T1.

Ergebnis: korrekte Reihenfolge für jeden beliebigen Eye Point:
wenn f1(e)<0 T1, T2 (e auf selber Seite wie T2, somit T1 weiter hinten, zuerst zu rendern)
sonst T2, T1

Verallgemeinerung mittels binärer Baumstruktur und Rekursion:

1. Auswahl eines Referenz-Dreiecks T1 als Root
2. Aufbau der Baumstruktur durch Gruppieren aller anderen Dreiecke in solche auf der einen und solche auf der anderen Seite von T1



Erkenntnis: welche Seite welche (Vordergrund, Hintergrund) ist, hängt bei der Berechnung der Ebenen-Gleichung von der Ordnung der Knoten eines Dreiecks ab (andere Eckpunkt-Beschriftung ändert Ergebnis der Matrizenmultiplikationen (Kreuz- und Punktprodukt))
Lösung: Vergleich mit allgemeinem Referenzpunkt – zB e

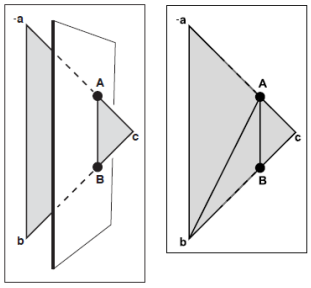
Algorithmus für alle Referenz-Dreiecke ist rekursiv zu wiederholen:

1. Rendern aller Dreiecke auf der entgegengesetzten Seite von e (möglicherweise verdeckt)
2. Referenz-Dreieck rendern
3. Rendern aller Dreiecke auf der gleichen Seite wie e (Vordergrund)

```
function render(bsptree tree, point e)
if (tree.empty) then
    return
if (f_tree.root(e)<0) then
    render(tree.plus,e)
    draw(tree.triangle)
    render(tree.minus, e)
else
    render(tree.minus,e)
    draw(tree.triangle)
    render(tree.plus,e)
```

Es ist immer ZUERST die ANDERE Seite des (Sub)trees (plus, minus), auf der sich e NICHT befindet, rekursiv abzuarbeiten (render). D.h. befindet sich e auf der minus-Seite (<0) in Bezug auf die Ebenen-Gleichung des aktuellen Referenz-Dreiecks $f_{tree.root}$ des (Sub)trees dann zuerst tree.plus sonst umgekehrt.
Folglich wird in jedem Durchlauf entschieden, ob zuerst der rechte oder linke Subtree des Referenz-Dreiecks gerendert werden soll.

Vorteil: funktioniert für jeden Blickpunkt (Viewpoint) e, deshalb kann für statische Szenen (d.h. nur Blickpunkt-Änderung, keine Bewegungen zB von Armen etc.) der BSP-Baum im Vorhinein berechnet werden und dann bei bewegter Kamera während der Laufzeit unterschiedlich traversiert werden → sehr gut für statische Kulissen in Spielen!



Problem: Ein Dreieck könnte die Ebene zur Teilung in die Gruppen (Dividing Plane) eines anderen kreuzen – häufig in 3D-Szenen! D.h. ein Knoten des Dreiecks ist auf einer anderen Seite als die beiden anderen.
Lösung: Finden der 2 Schnittpunkte (A, B) und Zerschneiden des Dreiecks in 3.
Ergebnis: Jedes der 3 Teil-Dreiecke ist nun klar auf einer Seite des Referenz-Dreiecks.

Berechnung der Schnittpunkte A und B
Vorgehen:

1. Parametrisieren der Linien ac und bc:
   Parametrische Linien-Gleichung: $l(t) = a + t(c - a)$
   t ... als eine Art Skalierung

   Alternative implizite Ebenen-Gleichung: $f(p) = ap_x + bp_y + cp_z + d = np + d = 0$
   n ... Ebenen-Normale
   d ... Entfernung (Distance) zum Ursprung

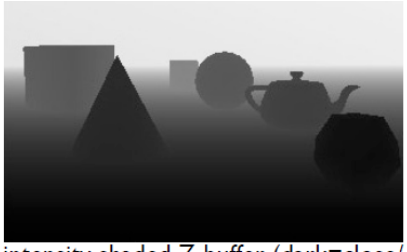
   Einsetzen von l(t) in f(p) ergibt: $n \cdot (a + t(c - a)) + d = 0$

2. Berechnung des Linien-Schnitts (bei Skalierung t) mit der impliziten Ebenen-Gleichung durch Auflösen von f(l(t)) nach t:

$$t = -\frac{n \cdot a + d}{n \cdot (c - a)} \qquad A = a + t_A(c - a) \qquad B = b + t_B(c - b)$$

Nachteil von BSP-Trees: sind nicht effizient für dynamische Szenen, aber gut für statische!

Z-Buffer (Depth-Buffer)
ist die verbreitetste Technik für Depth-Handling von dynamischen Szenen. Der Z-Buffer-Algorithmus ist extrem einfach und normalerweise hardware-unterstützt.

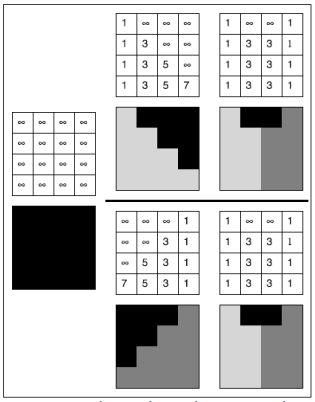

intensity shaded Z-buffer (dark=close/ small values, bright=far/large values)

Zu Beginn erhält jedes Pixel einen großen Wert bzw. Unendlichkeit. Danach wird der Tiefen-Wert und der Pixelinhalt ersetzt, sobald ein Dreieck bei einem bestimmten Pixel einen niedrigeren Tiefen-Wert besitzt.
Ergebnis: In jedem Pixel des Z-Buffers befindet sich die Tiefe des nähesten Dreiecks, dass bis dato gerastert wurde.

Das Bild zeigt, dass man zum selben Ergebnis kommt, wenn 2 Dreiecke in unterschiedlicher Reihenfolge mithilfe von Z-Buffering gerastert werden, da sowieso nur niedrigere Z-Werte ersetzt werden.

Bemerkungen zum Z-Buffer

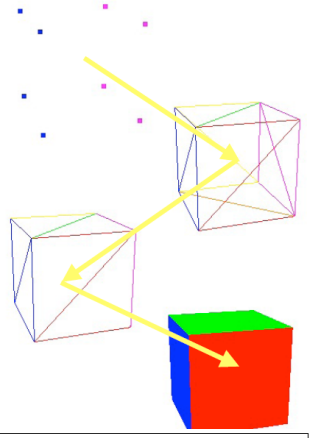
- Raster-Bilder werden in unterschiedlichen Buffern abgelegt: Frame-Buffer für Farbe, Z-Buffer für Tiefe und weitere.
- Derzeit hat der Z-Buffer auf Grafikkarten bis zu 32 bit (historisch 8 bit)
- Die Genauigkeit des Z-Buffers spielt eine dominante Rolle beim Rendering.
- Z-Buffer Fighting tritt auf, wenn der Unterschied zwischen 2 Z-Werten unter die Auflösungs-Genauigkeit des Z-Buffers fällt. Eine Entscheidung, was gerendert werden soll ist nicht möglich und die Szene flimmert!
- Die gesamte Z-Buffer-Auflösung wird zwischen Near und Far Clipping Plane der virtuellen Kamera verwendet. Eine sinnvolle Wahl ist deshalb essentiell und kann dynamisch, abhängig vom begrenzenden Volumen (Bounding Volume) der Szene erfolgen.
- Tatsächlich werden nicht die Z-Komponenten der Szenen-Geometrie direkt verwendet, sondern die Z-Werte nach der Projektion.



two rastered triangles and corresponding values in Z-buffer

Rasterisierung

Berechnung/Zeichnen von Linien zwischen Pixelkoordinaten der 2D-Projektion und Füllen von Pixeln der sichtbaren Dreiecksflächen, begrenzt durch diese Linien



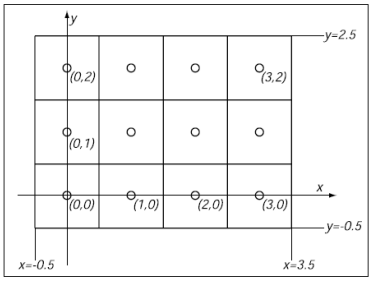
Rasterdisplays

derzeit State-Of-The-Art – Zukünftig könnten Displays zur Grafik-Darstellung holografisch sein (d.h. mit 3D-Effekt).

Aufbau von Rasterdisplays

- diskretes Gitter aus Pixeln
- für manche Displays bestehen die Pixel aus noch kleineren Elementen (zB RGB Sub-Pixel bei LCD/CRT Displays (Cathode Ray Tube – Röhrenbildschirme) – nicht bei DLP Displays (Digital Light Processor – für HDTV und digitales Kino)!)

Ausgangssituation: Bei der Projektion von Punkten einer Szene ist es nicht notwendigerweise so, dass diese exakt die diskreten Pixelpositionen mappen.
Ziel: Andere Punkte auf Linien und Oberflächen sollen angezeigt werden (auch relevant für Z-Values).
Mögliche Lösung: Diskretisieren und projizieren dieser Punkte aus dem 3D-Raum – ineffizient!

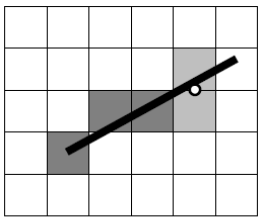


Zeichnen von Linien

bessere Lösung als Diskretion (danach Projektion) der Punkte aus dem 3D-Raum!
Idee: 3D Vertices nicht auf das nächst nähere Pixel runden, sondern gleich projizieren und im Raster die Pixel dazwischen füllen

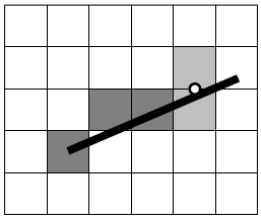
<u>Midpoint-Algorithmus (Mittelwert-Algorithmus)</u>

Meist verbreitetster Weg um Linien zu zeichnen, zeichnet dünnste, lückenlose Linie (Alternative: Bresenham Algorithmus)



Beispiel: m=0..1 (d.h. Linien mit Anstieg von 0-45°; gleiche Konstrukte für andere Steigungen!)
Vorgehen:

- Annahme: $x_0<x_1$ – andernfalls vertauschen
- Dadurch ergibt sich bei der Rasterisierung von links nach rechts, dass das nächste Pixel entweder auf gleicher Höhe oder um 1 höher zu zeichnen ist!
- Pro Spalte gibt es genau 1 Pixel (damit lückenlos), pro Zeile eventuell mehrere (m=0..1, 0-45°).



(Steigung: $m = \frac{y_1 - y_0}{x_1 - x_0}$)

– Berechnung des nächsten Pixels:
  Kandidaten: (x+1, y) oder (x+1, y+1)
  Mittelwert: (x+1, y+0.5)
  Entscheidung: Verläuft die Linie über dem Mittelwert, ist das obere Pixel, sonst das untere zu wählen. Für diese Berechnung mit der impliziten Linien-Gleichung:

$$f(x,y) = (y_0 - y_1)x + (x_1 - x_0)y + x_1 y_0 = 0$$

  (x_0, y_0) ... Startpunkt
  (x_1, y_1) ... Endpunkt
  Alternativ: Berechnung mit der parametrischen Linien-Gleichung (mit Parameter t)

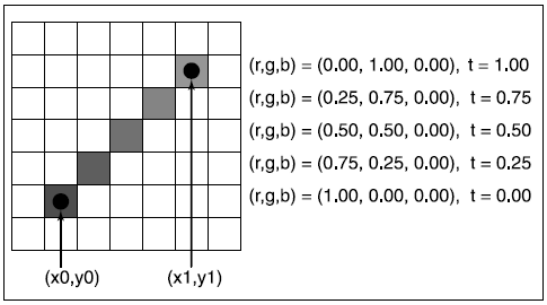
$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x_0 + t(x_1 - x_0) \\ y_0 + t(y_1 - y_0) \end{bmatrix}$$

  Beispiel: Steigung von m=-1..1, -45° bis +45° (ähnlich für andere Steigungsbereiche)
  Vorgehen:
  – Berechnung von t für alle x von x_0 bis x_1, Berechnung und Runden von y
  – t reicht von 0 (beim Start) bis 1 (am Ende). Einsetzen in die parametrische Linien-Gleichung ergibt die betroffenen Pixel zwischen den Linienendpunkten
  – Farbberechnungen: Sind die Farbwerte der Linienendpunkte gegeben, können mittels t für die dazwischenliegenden Pixel die Farben interpoliert werden:

$$t = \frac{x - x_0}{x_1 - x_0}$$

$$c_t = (1-t)c_0 + tc_1$$

  c_x ... RGB-Vektor von Pixel x



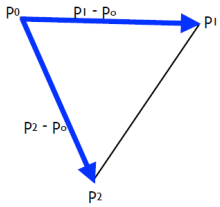
Rasterisierung von Dreiecken
Interpolation erfolgt in baryzentrischen Koordinaten (vorzeichenbehaftete, skalierte Entfernungen von den Linien eines Dreiecks)

Baryzentrische Koordinaten sind proportional zu den Flächen der Sub-Dreiecke, die sie aufspannen:

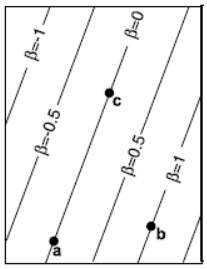
$$p = p_0 + \beta(p_1 - p_0) + \gamma(p_2 - p_0)$$

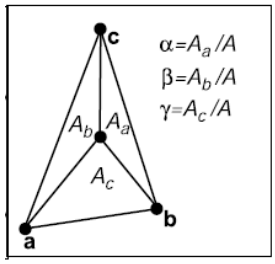
$$p = (1 - \beta - \gamma)p_0 + \beta p_1 + \gamma p_2$$

$$p = \alpha p_0 + \beta p_1 + \gamma p_2, \alpha = (1 - \beta - \gamma)$$



Baryzentrische Koordinaten ermöglichen die Interpolation von:
  – X,Y Bildschirm-Koordinaten (der Projektion)
  – Z-Values
  – RGB-Farben
  – Normalvektoren

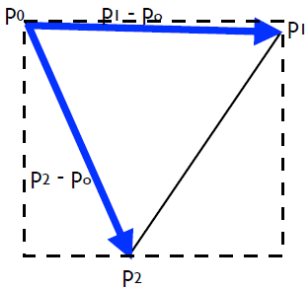




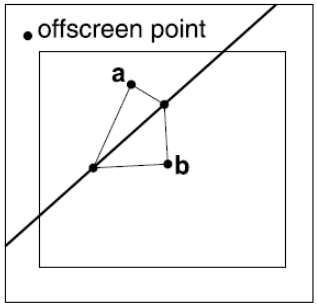
Welche Dreiecke auf dem Bildschirm sind zu rastern?
ALLE auf dem Screen mittels Brute Force (ineffizient)

```
for all x
  for all y
    compute coordinates
    if barycentric coordinates in interval 0..1 (for a triangle)
      interpolate parameters using barycentric coordinates
      draw pixel with interpolated parameters
    endif
  endfor
endfor
```

Lösung: Berechnung der Bounding Box (viereckige Begrenzung um das Dreieck) und Rastern von einzig dem Dreieck (Pixel im Dreieck – siehe oben) in der Bounding Box statt des gesamten Screens.



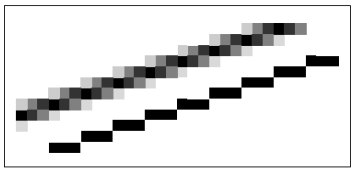
Alternative Lösung: Berechnung der gemeinsamen Bounding Box aller Dreiecke auf dem Screen und Rastern aller Pixel der Box, wobei für jedes Pixel zu testen ist, ob es innerhalb oder außerhalb eines Dreiecks der Box liegt.
Problem: Benachbarte Dreiecke teilen sich Kanten. Eine Kante soll aber genau einem Dreieck zugeordnet sein!
Lösung: Zuordnung der Eckpixel zu genau einem Dreieck

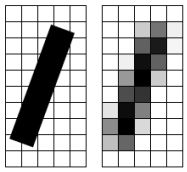


- Festlegung eines fixen "Offscreen Points"
- Feststellung auf welcher Seite der (verlängerten) Kantenlinie sich der Offscreen Point befindet
- Zuweisung der Ecke zu dem Dreieck, dessen dritter Eckpunkt auf derselben Seite wie der Offscreen Point liegt

Problem: Kantenlinie könnte durch Offscreen Point verlaufen
Lösung: im Falle des Falles Verwendung eines 2. Offscreen Points

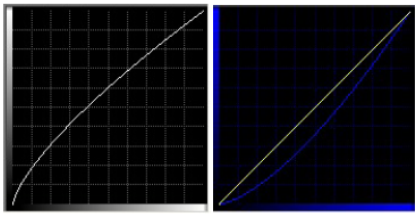
Aliasing (Treppeneffekt)
Linien erscheinen beim Rastern zackig. Lösung: Anti-Aliasing (Kantenglättung) zB durch einfache Anwendung eines so genannten Box Filters oder Faltung, oder fortgeschritteneren Techniken – hardware-unterstützt durch die Grafikkarte





Rendern (Übersetzen) und Anzeigen von Farben
Farbraum (Color Space) = Parametrisierungsschema für Farben. Am weitesten verbreitet in CG ist der RGB-Farbraum, da er von den meisten gerasterten Displays verwendet wird. Red, Green und Blue sind hier die die 3 Primärfarben, die für additives Farbmischen verwendet werden. Herkömmliche Grafikkarten erlauben 256 (8bit) Farbwerte für jeden Farbkanal, folglich sind 2^{24} Farben adressierbar. Die Darstellung von RGB-Farben ist Display-abhängig!

Monitor Gamma
Ausgangssituation: Hinsichtlich Farbe ist das Verhalten von Monitoren nicht linear (auch nicht das der menschlichen Wahrnehmung). Diese Tatsache macht Farbkalibrierung zu einer wichtigen und schwierigen Aufgabe. Lösung: Annäherung durch die Gamma-Funktion (heutzutage im Grafikkartentreiber anpassbar für alle RGB-Kanäle)



$$d = m \cdot a^{\gamma}$$

d ... angezeigte Intensität (weißer Graph rechts)
m ... maximale Intensität
a ... gewünschte Intensität (weißer Graph links)
blauer Graph = Gamma-Funktion

Herausfinden der Gamma-Werte der Funktion:

- Anzeige eines Schachbretts und eines grauen Bereichs
- Aus ausreichender Entfernung betrachtet erscheint das Schachbrett als einheitliche graue Fläche. Der Grauwert der einheitlichen Fläche (entspricht d) ist solange zu variieren, bis das Schachbrett (entspricht a) im selben Grauton erscheint. Dies entspricht einem Grauwert von 50% (da schwarze Felder 0%, weiße 100% sind).
- Aus der obigen Formel kann nun γ berechnet werden: $\gamma = \frac{ln(0.5)}{ln(a)}$

Gamma-Korrektur: Anzeige von a als $a^{(1/\gamma)}$

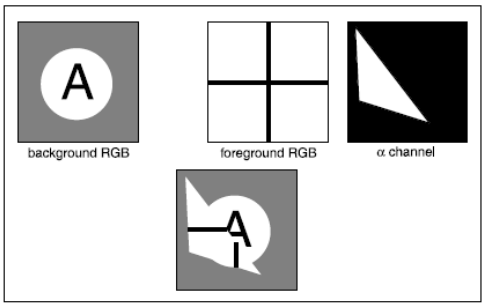
Alpha Blending
Die herkömmliche Rendering Pipeline unterstützt neben den RGB-Kanälen zusätzlich den Alpha-Kanal. Er enthält ein normalisiertes Gewicht (Alpha) pro Pixel.

Verwendung: zum Zusammenmischen (Blending) von unterschiedlichen, gerasterten Bereichen zur Simulation von einfachen Transparenz-Effekten (zB halbtransparenter Vordergrund (aktuelle Farbe) über undurchsichtigem Hintergrund (bereits im Grafikspeicher abgelegt)).
vgl. http://de.wikipedia.org/wiki/Blending_%28Grafik%29

Vorgehen:

– Der Alpha-Wert gewichtet während des Rasterns die Farbe des Vordergrunds c_f und des Hintergrunds c_b und addiert sie. Da mit c der Farbwert eines Pixels bestimmt wird (unter Berücksichtigung der sich überlappenden Schichten, ist die Rendering-Reihenfolge egal, allerdings muss man das Gewicht beachten, welche Schicht den Vordergrund und welche den Hintergrund darstellt (für korrektes Einsetzen von c_f und c_b in die Formel).
– Sollte Alpha 0 oder 1 sein, wird einfach der Bereich zugeschnitten statt gemischt.

$$c = \alpha c_f + (1 - \alpha) c_b$$



Bemerkung am Rande: Es gibt einige unterschiedliche Blending-Funktionen! Sie legen fest, wie die Farben von Vordergrund und Hintergrund gemischt werden. Gebräuchlich sind lineare, exponentielle und logarithmische.

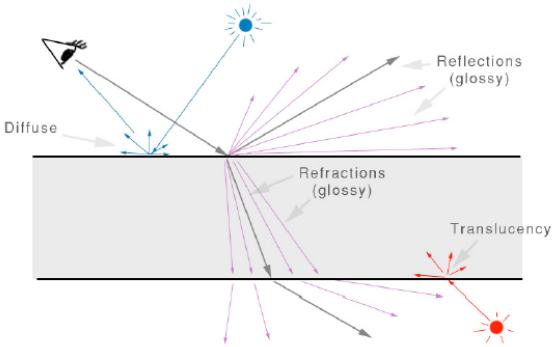
Beispiel Fog-Blending:
Das Bild wird so verändert, dass die Farbe aller Punkte entsprechend ihrer Entfernung vom Betrachter im dreidimensionalen Raum mit einem festen Farbton gemischt wird. Auf diese Weise sollen weit entfernte Objekte in einem künstlichen Nebel verschwinden. Zur Realisierung des Effektes bedient man sich der Entfernungsinformation der Pixel, die im Z-Buffer (siehe Kapitel Depth-Handling) gespeichert ist. Dieser Wert, zusammen mit einem Blendfaktor (Alpha), einer Farbe und einer einfachen Funktion bestimmen den resultierenden Farbwert.

Beispielfragen zum Kapitel:

- Was ist Alpha-Blending? Ist die Reihenfolge egal? Was versteht man unter Blending-Funktionen?
- Diskutieren Sie (im Detail) wie Verdeckung von Dreiecken effizient für statische Szenen und bewegliche Kamera berechnet werden können. (Algorithmus)
- Erklären Sie (in aller Kürze), wie der Tiefentest für dynamische Szenen auf gängiger Grafikhardware realisiert wird.
- Erklären Sie (in aller Kürze), wie Dreiecke gerastert werden. Gehen Sie dabei auch auf die baryzentrische Koordinate ein.

3. Local Shading and Illumination

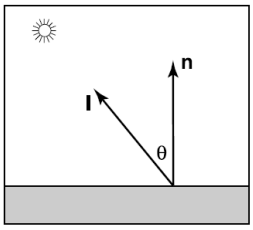
Shading = Surface Shading = Simulation von Lichtquellen (übersetzbar mit "Schattierung")
Illumination = Beleuchtung – gleichgesetzt mit "Shading"
diffuse = zerstreut, refractions = Brechungen
(Grafik: nach der Streuung werden die einzelnen gestreuten (violetten) Strahlen ebenfalls gebrochen!)
translucency = Durchscheinen/Durchsichtigkeit



Es existieren unterschiedliche Beleuchtungs-Modelle (Lighting/Illumination Models):
Lokale/direkte Beleuchtung: die Beleuchtung jedes Oberflächenpunkts erfolgt ausschließlich aufgrund des direkten Einflusses von Lichtquellen
Globale/indirekte Beleuchtung: zusätzlich fließen indirekte Effekte wie zB Reflexionen zwischen Oberflächen (Surface Inter-reflections) ein

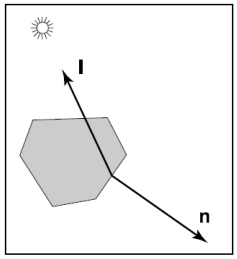
Lambertian Shading Model (nach Johann Heinrich Lambert)
Anwendung bei Objekten mit matter Oberfläche
Prinzipien:

1) Shading (c) eines Oberflächenpunkts ist proportional zum Kosinus des Winkels zwischen Oberflächennormale (n) und Richtung der Lichtquelle (l). Einschränkung: Diese Annahme ist allerdings nur gültig, wenn die Lichtquelle relativ weit weg von der Oberfläche ist. Optimal wäre eine unendlich weite Entfernung, da nur hier alle Lichtstrahlen dieselbe Richtung haben.



$$c \propto cos(\theta) = |n \cdot l|$$

2) Oberflächen, die der Lichtquelle abgewandt sind sollen nicht beleuchtet werden. Es reicht also zu prüfen, ob der Winkel +/- 90° bzw. das Punktprodukt (Skalarprodukt) von l und n positiv ist.
Bsp. rechts: nl ist negativ, deshalb gilt für die errechnete Farbe c=0



3) Lichtmodulation ist linear:
Die Farbe des Oberflächenmaterials (c_r) und der Lichtquelle (c_l) fließen durch eine zusätzliche Multiplikation mit ein (zB mit RGB-Werten durchzuführen).

$$c \propto c_r c_l \, |n \cdot l|$$

4) Mehrere (N) Lichtquellen (i) werden durch Summierung (sowohl von Lichtstärke als auch -farbe) berücksichtigt.
(Bemerkung: Evtl. ist eine Amplitudenbegrenzung (Clipping) notwendig, wenn zu große Helligkeitsunterschiede auftreten, die aber nicht abgespeichert werden können, weil es sich nicht um ein HDR (High Dynamic Range) Bild handelt).
vgl. http://de.wikipedia.org/wiki/High_Dynamic_Range_Image

$$c \propto c_r \sum_i^N c_{l_i} \, |n \cdot l_i|$$

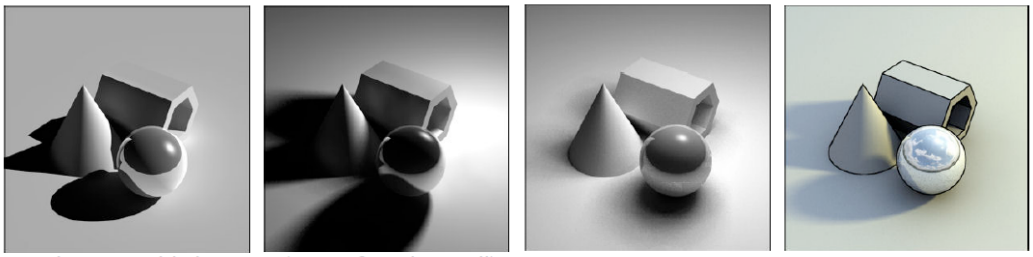
Bemerkungen zum Lambertian Model

$$c \propto \frac{cos(\theta)}{r^2} = \frac{|n \cdot l|}{r^2}$$

- Shading auf rauen/streuenden (diffusen) Oberflächen hängt in der Realität von mehr Faktoren als nur dem Winkel zwischen der Normalen und der Lichtrichtung ab.
- Die Lichtintensität fällt quadratisch proportional zur Distanz r zwischen Lichtquelle und Oberfläche ab.
- Die GLEICHE Lichtmenge, die mit einem festen/beschränkten Winkel ausgesandt wird, breitet sich auf einem größeren Teil der Oberfläche aus, wenn diese weiter entfernt ist. Somit ist die Lichtmenge PRO EINHEIT der Oberfläche WENIGER als bei geringerer Entfernung! (Square-Distance Attenuation/Dämpfung)
- Die quadratisch abnehmende Lichtmenge wird in CG selten verwendet, da der rasche Shading Abfall als unschön angesehen wird.

Arten von Lichtquellen

- Gerichtetes (directional) Licht: Die Lichtquelle ist unendlich weit entfernt und wird nur anhand der Richtung beschrieben.
- Point light: eine Lichtquelle mit Position (kugelförmige Ausbreitung des Lichts im Raum)
- Spot light: Point light mit auf einen Kegel/Winkel begrenzte Lichtausbreitung, ein Spot light mit einem Winkel von 360° wäre also ein Point light! *vgl.* http://board.gulli.com/thread/248710-3d-glossar-3d-fachbegriffe-/
- Weitere: Kugel- (spherical), Flächen- (area) und Umgebungs-/Umwelt- (environment) Licht



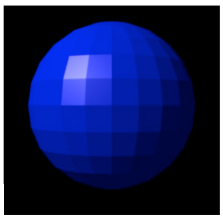
directional light | (not infinitely small) spherical light | area light | environment light

$\cdot\, l_i|)$

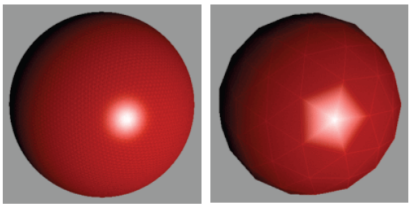
Ambiente Komponente/Umgebungslicht
Prinzip: Damit beim Lambertian Shading die der Lichtquelle abgewandten Seiten nicht schwarz (c=0) erscheinen, wird ein ambienter Term c_a hinzugefügt, dabei handelt es sich um eine konstante Farbe, die addiert wird.
Sinn von c_a ist die Annäherung des konstanten Umgebungslichts gebildet aus gegenseitigen Reflektionen (Inter-Reflections), Streuungen, etc. Die durchschnittliche Farbe aller Oberflächen bietet einen guten Wert!

Wie sind nun die hergeleiteten Formeln anzuwenden?

Flat Shading
Prinzip: Berechnung der Oberflächenfarbe c für jedes Dreieck mithilfe seiner Normalen, danach Anwendung auf jedes Pixel während der Rasterisierung.
Problem: Für eine möglichst glatte Oberfläche werden unzählige Dreiecke benötigt aber je höher die geometrische Auflösung, umso schlechter die Performance!

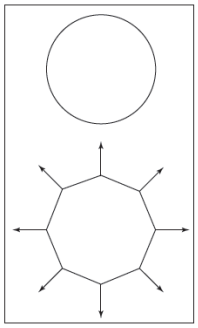


Gouraud Shading (nach Henri Gouraud)
Prinzip: Statt der Oberflächennormale eines Dreiecks ist der Durchschnitt der Oberflächennormalen, die so genannte Vertex-Normale für die Berechnung von c heranzuziehen. (Baryzentrische) Vertex-Farben und andere Parameter wie beispielsweise. Tiefenwerte (z) werden interpoliert (d.h. anhand einer Funktion berechnet). Problem: Das Ergebnis ist zwar besser als bei Flat Shading, jedoch immer noch sehr Performance-lastig bei Erhöhung der geometrischen Auflösung!

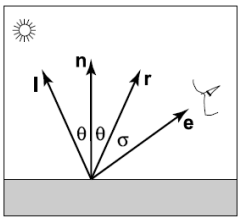


Gouraud shading with high geometry resolution | Gouraud shading with low geometry resolution

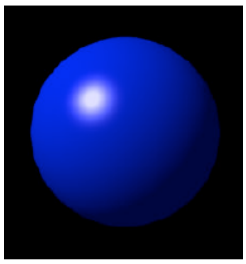
vertex normals are averaged surface normals



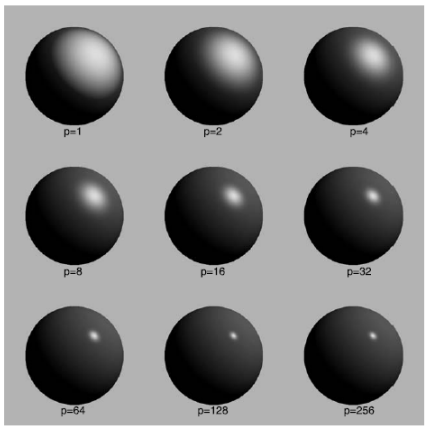
Phong Shading (nach Bui Tuong Phong)
Anwendung: Statt weitschweifigen (diffusen) Reflexionen sollen spiegelnde Glanzlichter (specular highlights) erzeugt werden.
Prinzip: Die Annäherung spiegelnder Glanzlichter auf der Oberfläche erfolgt durch Berücksichtigung von sowohl Richtungsvektor zum Beobachter/Viewer (e) als auch der natürlichen Richtung der Reflektion (r).
Die Oberflächenfarbe c ist hell, wenn e=r und fällt stufenweise für größer werdende σ.
ACHTUNG: Sollte r*e negativ sein, muss dies gesondert behandelt werden!



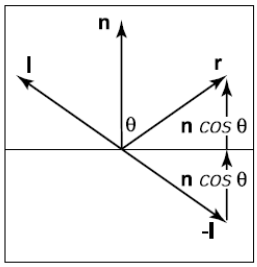
$$c \propto c_r c_l (r \cdot e) = c_r c_l (cos(\sigma))$$



Problem: Die spiegelnden Glanzlichter erscheinen unrealistisch flächendeckend.
Lösung: Einführung des Phong Exponents p: $c \propto c_r c_l (r \cdot e)^p$

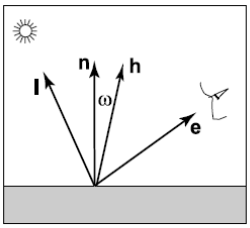


Raue Oberflächen: kleines p, ergibt große Glanzlichter
Glatte Oberflächen: großes p, ergibt kleine Glanzlichter
ACHTUNG: r*e darf hier nicht negativ sein!



Berechnung von r aus Lichtquellen-Richtungsvektor l und Oberflächen-Normalvektor n.
Diese Annäherung ist schnell, da negative Werte nicht geprüft werden müssen.

$$r = -l + 2(l \cdot n)n$$



Alternative: Berechnung des Halb-Vektors (half-vector) h zwischen Lichtquellen-Richtungsvektor l und Viewer-Richtungsvektor e

$$c \propto c_r c_l (h \cdot n)^p = c_r c_l cos(\omega)^p$$

Glanzlichter treten auf, wenn h nahe n ist!
Vorteil: h*n ist immer positiv!
(Nachteil: sqrt und div sind zu berechnen? keine Ahnung wo das sein soll?!)

$$h = \frac{e + l}{\|e + l\|}$$

Danach: Phong Interpolations

- **Phong Vertex Interpolation**
  Berechnung der Oberflächenfarbe c für Vertices und Interpolation während des Rasterns.
  Problem: möglicherweise Artefakte, falls die geometrische Auflösung zu niedrig ist.
  Formel für interpoliertes c mit baryzentrischen Koordinaten:
  (d.h. hier sind die anderen Vertices c0, c1 und c2 bekannt)
  $$c = \alpha c_0 + \beta c_1 + \gamma c_2$$

- Lösung: **Phong Normal Interpolation**
  (Standard für die meisten CG-Systeme!):
  Statt Farben zu interpolieren, werden Normalvektoren während der Rasterization interpoliert:
  $$n = \alpha n_0 + \beta n_1 + \gamma n_2$$
  Auf jedes gerasterte Pixel wird dann ein Shading Model (und zwar Phong Shading) angewandt.

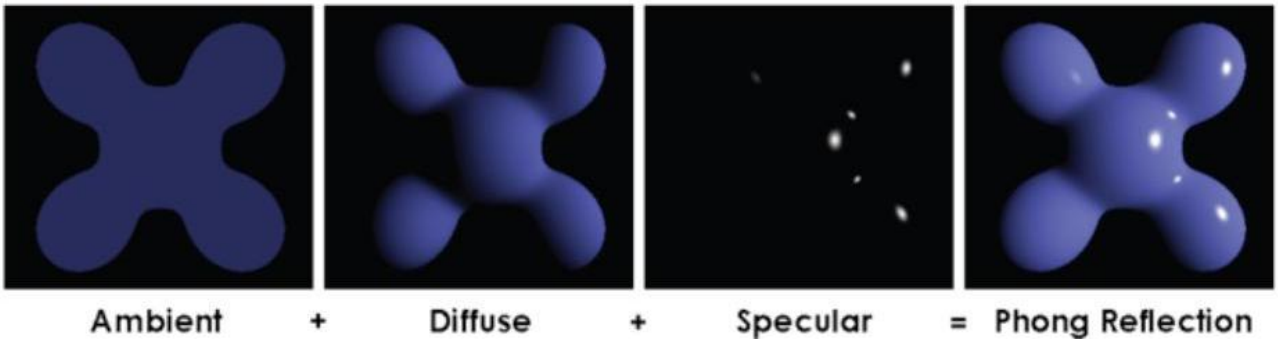
Fazit – Local-Shading bei üblichen CG-Systemen

- Die ambiente (c_a), diffuse (c_d) und spekulare (c_s, p) Komponente sind normalerweise als Materialparameter der Oberfläche, gemeinsam mit deren Reflexion c_r definiert.
- Mehrere (N) Lichtquellen (i) mit individuellen Farben (c_l)
- Lineare Lichtmodulation

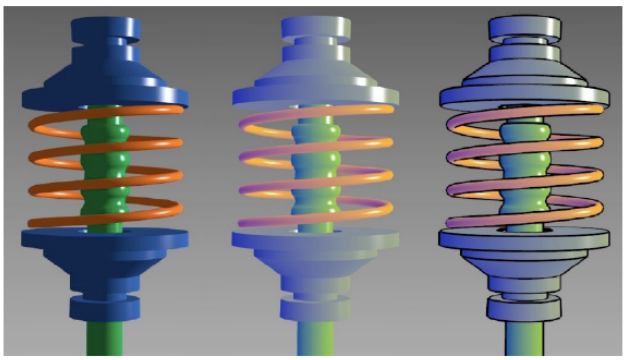
surface reflectance, light color

$$c = c_r(c_a + \sum_i^N c_{l_i}(c_d |n \cdot l_i| + c_s(h_i \cdot n)^p))$$

ambient, diffuse, specular



Non-Photorealistic Rendering (NPR)
Techniken zur Visualisierung mit weniger Stördaten und somit einfacher lesbar zB Shading Techniken zur Simulation von Zeichnungen, (Car)toons, Gemälden oder Skizzen/Entwürfen



Realisierungen von typischen NPR-Anwendungen

<u>Silhouetten (Umrisse) hervorheben</u>
gegeben: ein 3D-Dreiecks-Modell

Definition einer Silhouetten-Kante: Kante, die 2 Dreiecke gemeinsam haben, wobei eines Richtung Beobachter, das andere entgegen gerichtet ist

Vorgehen:
Betrachten der beiden Normalen n_0 und n_1 und den Vektor von der Kante zum Beobachter-Auge

$$(e \cdot n_0)(e \cdot n_1) \leq 0$$

ODER
Prüfen der beiden impliziten Ebenen-Gleichungen

$$f_0(e) f_1(e) \leq 0$$

<u>Alle sichtbaren Kanten hervorheben</u>
Festlegung: Sichtbare Kanten befinden sich dort, wo tatsächlich ein Eck in der Geometrie ist, d.h. gemeint sind all jene Kanten an den Grenzen zwischen Vordergrundflächen (Vordergrund-Hintergrund-Kante wäre eine Silhouette)
Vorgehen: Prüfen des Unterschieds zwischen den Normalen zweier Dreiecke, ist dieser Unterschied größer als ein bestimmter Schwellenwert, handelt es sich um eine Kante, die hervorgehoben werden soll

$$n_0 \cdot n_1 \leq threshold$$

<u>Cold-To-Warm Shading</u> (Beispiel für Color-coded Shading) hinzufügen
Definition Cold-To-Warm Shading: Oberflächen, die in eine Richtung schauen, werden kühler beleuchtet (hier: violett, dunkelblau), Oberflächen, die in die andere Richtung schauen, werden wärmer beleuchtet (hier: orange, hellgrün)

Vorgehen:
1) Festlegen einer Richtung für warmes Licht (l)
2) Anwenden des Kosinus (berechnen von n*l):
   zur Modulation zwischen warmer (c_w) und kalter (c_c) Farbe
3) Mit errechnetem Wert ein Mischverhältnis aufstellen:

$$k_w = \frac{1 + n \cdot l}{2}$$

$$c = k_w c_w + (1 - k_w) c_c$$

Beispiel:
Wahl der reinen, warmen Farbe:

$$n \cdot l = 1 \rightarrow c = c_w$$

Wahl einer aus warm und kalt gemischten Farbe:

$$n \cdot l = 0 \rightarrow c = \frac{1}{2} c_w + \frac{1}{2} c_c$$

Beispielfragen zum Kapitel:
- Erklären Sie (in aller Kürze), den Unterschied zwischen Gouraud Shading und Phong Shading.

4. Texturemapping

Non-Uniform Surface Reflectance
Wir wissen bereits, wie 3D Bilder beleuchtet werden, was aber ist mit Oberflächen, die sehr feine Details haben? Einerseits könnte man mittels einer extrem hohen Auflösung, bei der jeder Knoten bzw. jedes Dreieck eine andere Reflektion hat, arbeiten, dies würde aber einen Rendering-Overkill gleichkommen. Besser ist, man verwendet niedrig aufgelöste geometrische Objekte, auf deren Oberflächen man die benötigten Details „aufmalt“.

2D Texture Mapping
Hier handelt es sich um die häufigste Form des Mappings in der Computergrafik. Texturdetails („Texels“) sind in einer 2D Grafik gespeichert und werden auf eine 3D Oberfläche aufgemalt – aber wie? Setzen wir einen normalisierten (0..1) 2D Texturraum in u,v Koordinaten voraus. Die Reflektion der Textur ist dann eine Funktion von R(u,v). Die Zuweisung einer Reflektion an eine 2D Oberfläche ist dann einfach

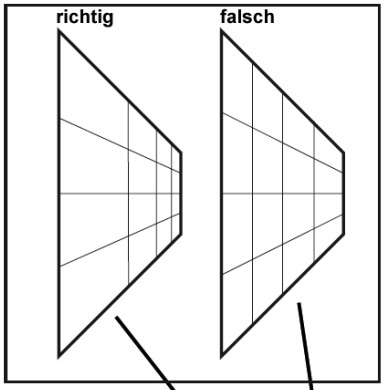
Beispiel: 2D Bild auf eine Kugel abbilden:
Sphäre: $x = x_c + R * \cos(\varphi)\sin(\theta),\ y = y_c + R * \sin(\varphi)\sin(\theta),\ z = z_c + R * \cos(\theta)$ mit $x_c, y_c, z_c = center,\ R = radius$

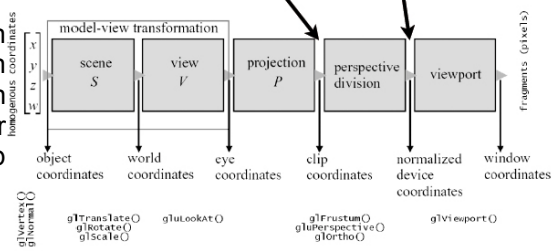
Sphärenkoordinaten $\theta = \arccos(\frac{z - z_c}{R})$ und $\varphi = \arctan(\frac{y - y_y}{x - x_c})$ (Längen- und Breitenkoordinaten)

Zuweisung der Texturkoordinaten: $u = \frac{\varphi}{2\pi}$ und $v = \frac{\pi - \theta}{\pi}$

Tessellated Models
Wie geht man bei mosaikartigen Oberflächen vor? Bestimme die Texturkoordinaten für jeden Knoten und interpoliere dann die Koordinaten durch das Dreieck mit baryzentrischen Koordinaten. (In diesem Sinne ist Texturemapping auch nur Teil der Rasterisierung). Wenn die Texturkoordinaten über 0 und 1 hinausgehen, können die Texturen wiederholt, gespiegelt, etc. werden.

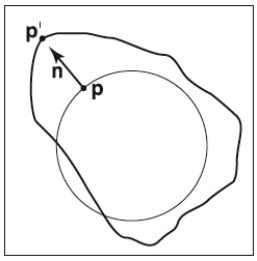
Perspective Correct Textures
Interpoliert man die Texturkoordinaten im Screenspace, erhält man eine falsche Ansicht. In diesem Falle müssen wir einige Korrekturen an der Textur vornehmen um eine korrekte Ansicht zu erhalten.
Texturkoordinaten im Screenspace zu interpolieren bedeutet, diese NACH der perspective division auszuführen. Um richtige Bilder zu erhalten, müssen wir die Interpolation aber VOR der perspective division durchführen. Wir müssen also die Texturkoordinatenberechnung mit baryzentrischen Koordinaten in Worldspace durchführen, nicht im Screenspace.



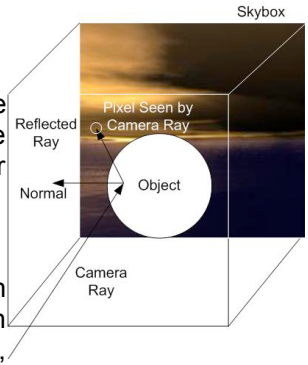


Bump Maps
Alle unsere 2D-Strukturen waren bis jetzt glatt – damit lassen sich natürlich keine rauen oder „beuligen“ Oberflächen realistisch darstellen. „Verwirrte“ Oberflächennormalen werden mit der 2D Textur in einer Datei gemeinsam mit der Oberflächenreflektion abgespeichert. (geometry + Bump Map = bump mapped geometry)

Displacement Maps
Das Problem bei Bump-Maps ist, dass diese keine realistischen Schatten erzeugen – es sei denn man ändert die Oberflächennormalen. Weiters erzeugen Bump-Maps keine realistischen „Bumpy“-Silhouetten. Um dem entgegenzuwirken gibt es Displacement-Maps. Diese Maps sind auch 2D-Maps, welche kleine Verschiebungsvorläufe (Offsets) entlang der Oberflächennormalen speichern. Anstatt die Reflektionen zu beeinflussen, ändern sie die Oberflächengeometrie, was zu realistischeren Ergebnissen führt. (Kann effizient in Z-Buffer code implementiert werden!)



Environment Maps
Offensichtlich ist Texturemapping sehr mächtig und bietet weitaus mehr Möglichkeiten als nur einfaches „Wall-papering“. Environment-Maps ist ein anderes Beispiel dafür – diese erlauben die Simulation von gespiegelten Hintergründen auf Oberflächen. Man kann hier von einer nichtplanaren Texturoberfläche sprechen, welche die Szene umgibt. Die Texturkoordinaten die hier einem Oberflächenpunkt „zugewiesen“ sind, basieren auf dem Reflektionsvektor auf der Oberfläche. Environment Maps werden auf verschiedene Arten gespeichert (sphärische Indextabelle oder kubus-basierte Tabellen)

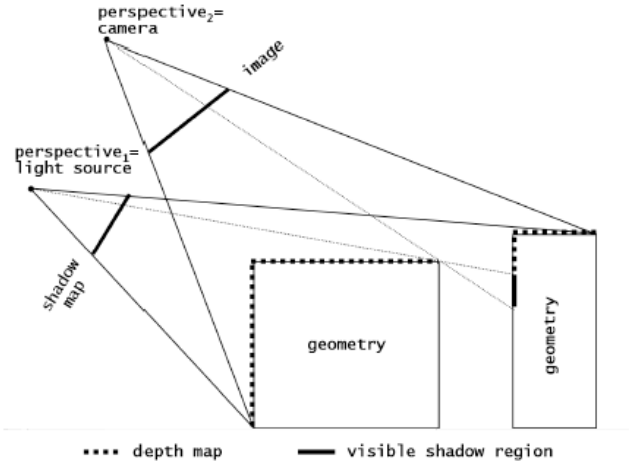


Cube Maps
Cube mMaps sind eine Spezialform von Environment Maps, welche eine würfelbasierte Umgebung realisieren. Die Generierung ist ganz einfach: Platziere die Kamera in der Mitte der Umgebung und „mache“ 6 perspektivische Bilder für alle 6 Richtungen.

Projective Texture Maps
Bei projizierten Texturemaps werden Texturkoordinaten (t) von Knotenkoordinaten (v) so generiert, dass diese genau auf die Texels passen, die von der aktuellen Ansicht aus verfügbar sind. (Wie ein Overhead-Projektor). Dies ist sehr einfach, wenn wir uns daran erinnern, wie die Transformationspipeline arbeitet:
$M_s = scene\,transformation\,, M_v = view\,transformation\,of\;\;projector\,, M_{pp} = perspective\,projection\,of\;\;projector$
h= homogeneous coordinate for perspective dividion.
Nun brauchen wir ein Mapping , welches die normalisierten Gerätekoordinaten (-1..1) in Texturkoordinaten (0..1) umwandelt: $t = M_{d2t}(M_{pp} M_v M_s v)/h$.

Einfacher ausgedrückt: Projective texture mapping kann benutzt werden, um eine Perspektive in eine andere umzuwandeln.

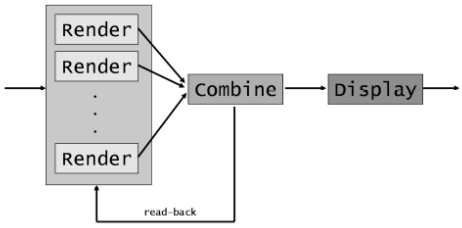
- Rendere $Bild_i$ von $Perspektive_i$
- Berechne Texturkoordinaten für $Perspektive_i$ und weise diese den Knoten zu



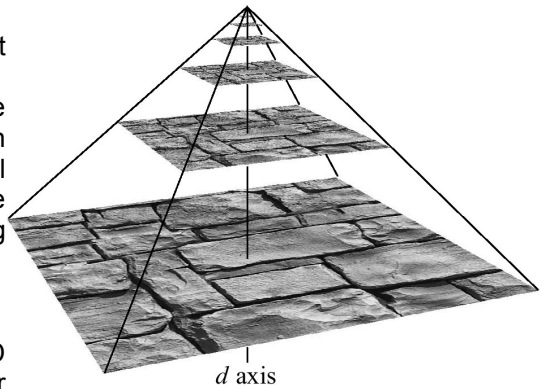
Shadow Maps
Aber auch projective Texture-Maps geben harte Schatten ab. In diesem Beispiel ist $perspective_1$ die Richtung aus der das Licht kommt und $image_1$ enthält nur den z-Wert für diese Perspektive (genannt „Shadowmap“, obwohl es eigentlich eine „Lightmap“ ist). Dieser z-Wert repräsentiert den Abstand für alle Punkte zur Lichtquelle (d_1). Anschließend wird die Szene aus der $perspective_2$ gerendert (Sicht der Kamera) – jeder von der Kamera sichtbare Punkt wird gerastert – für diese gerasterten Punkten wird ebenso der Abstand zur Lichtquelle berechnet (d_2).
Wenn $d_2 > d_1$ ist, dann ist der Punkt im Schatten, sonst nicht.

Note on 2D Textures
Es gibt viele verschiedene Arten von 2D Texturemapping (reflectance, normals, displacement, environment, shadows) – diese können alle miteinander kombiniert werden. Multiple Image Textures (zB Reflektion, Schatten, Umgebung) kann mit Alpha-Blending kombiniert werden. Zusätzlich zu den Bildtexturen können Normalen und displacements (Verschiebungen) hinzugefügt werden. Viele dieser Techniken benötigen mehr als einen Rendering-Schritt. Weiters können Texturen auch zum Speichern anderer Informationen und Parametern genutzt werden – zB als generelle 2D Matrix.

Multi-Pass Rendering
In der einfachsten Form werden immer nur einzelne Szenen gerendert. Um komplexere Effekte generieren zu können rendern wir die benötigte Szene mehrmals off-screen und speichern die Informationen in eine Textur. Jedes Bild zeigt die Szene von der selben Perspektive, rendert aber unterschiedliche Effekte. Zum Schluss werden die Bilder miteinander kombiniert und das Ergebnis gezeigt. Ergebnisse von früheren Rendering-Durchläufen können auch verwendet werden – dies wird als „Multi-Pass-Rendering“ bezeichnet.



MIP Mapping
Die Auflösung einer Textur ist immer wichtig: Qualität (hoch ist besser) oder Performance (niedrig ist besser).
Eine Kombination ist MIP mapping (multum in parvo = viele Dinge in einem). Für jede Textur werden mehrere Versionen vorberechnet – abhängig davon, wie viel eines projizierten Pixel in einem Texel ist, wird eine besser oder schlechter aufgelöste Version der Textur verwendet. Effiziente Wege zur Speicherung von MIP-Maps existieren.



3D Textures
Neben 2D Texturen gibt es auch noch 3D Texturen. Bei 2D Texturen müssen wir immer einen Weg finden, die 2D Textur auf 3D Koordinaten abzubilden. Bei 3D Texturen ist dies einfacher. 3D Texturen werden oft durch analytische Funktionen oder Algorithmen beschrieben. Sie werden verwendet um natürliche Materialien (Wasser, Holz, etc.) zu modellieren.

Example: Volume Rendering
Dreidimensionale Texturen können für Volume-Rendering benutzt werden. Dreidimensionales Textur-Mapping ist hardware-unterstützt. Eine Menge von parallelen Polygonen, welche mit den 3D-Texturen bemappt werden, werden verwendet. Die Polygone werden von hinten nach Vorne gerendert. Alpha texturemapping wird für Transparenz benötigt.

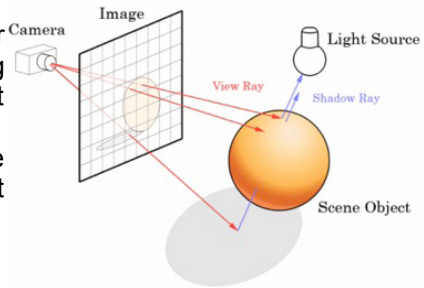
Beispielfragen zum Kapitel:
- Was versteht man unter perspektivischer Projektion? Welche Probleme gibt es bei Texturen und der Perspektive?
- Was ist Bump-Mapping? Welche Einschränkungen?
- Was ist Mipmapping?
- Was sind 3D-Texturen?

5. Global Illumination

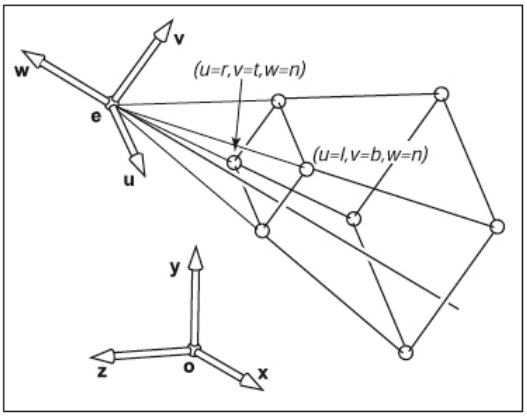
5.1. Raytracing

Local vs. Global Shading
Bisher haben wir folgende Dinge gemacht: Rendern von (Dreiecks-) Primitiven, Berechnen von Shadingparametern (Farbe, Licht, Textur, Schatten, etc.) für jeden Knoten, Interpolation der Parameter während der Rasterisierung um das Erscheinen eines jeden Pixels auf dem Bildschirm zu berechnen. Wenn die Beleuchtungsparameter für jeden Punkt der Oberfläche ohne Rücksicht auf die Lichtquellen berechnet werden (indirekte Beleuchtungen werden nicht beachtet), so spricht man von Local Shading. Nun beschäftigen wir uns mit indirekten Beleuchtungen, Reflektionen, Lichtstreuungen, Lichtbrechung – Techniken die diese Effekte berücksichtigen, nennt man Global Shading Techniken. Shadowmapping und Environmentmapping sind nur Annäherungen für Echtzeitapplikationen – ihre Qualität ist beschränkt.

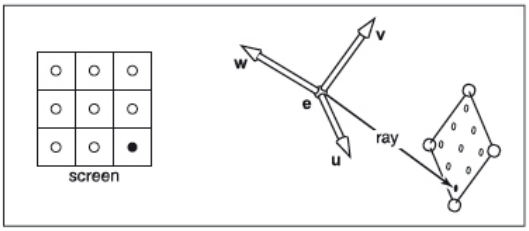
Ray Tracing
Wird benutzt um fotorealistische Bilder zu generieren. In der Filmindustrie sehr gebräuchlich da hier keine Echtzeit-Rendering Vorgänge stattfinden. Ray Tracing wird Pixel für Pixel durchgeführt (nicht Primitive für Primitive).
Für jedes Pixel auf dem Bildschirm wird ein Lichtstrahl durch die Szene „verfolgt“ und das Verhalten aller Objekte wird akkumuliert. Es ist leicht diesen Vorgang zu parallelisieren.



Ray Tracing Basics
Als erstes wird eine uns bereits bekannte Computergrafik-Kamera initialisiert.
Dann werden vom Kameramittelpunkt (e) Strahlen durch die Viewing Plane (near clipping plane) abgegeben – wir gehen davon aus dass Viewing plane = Screen plane). Wir berechnen die Unterbrechung jedes Strahls bei der ersten (Dreiecks-) Primitive, die geschnitten wurde. Eigenschaften dieses Dreiecks (Farbe, Normale, Transparenz, etc.) beeinflussen die Eigenschaften des betroffenen Strahls. Basierend auf diesen Eigenschaften kann der Strahl weiterverfolgt werden und wir können die Berechnung für das nächste Dreieck (auf dem Weg des Strahls) vornehmen.



Dieser Vorgang wird solange wiederholt, bis eine Exit-Bedingung eintritt. Auf dem Weg durch die Szene werden auch Modulationsinformationen akkumuliert und die Eigenschaften des Strahls aktualisiert. Die endgültigen Eigenschaften des Strahls werden genutzt, um die Farbe des Pixels, zu dem der Strahl gehört, zu berechnen. Dieser Vorgang kann für alle Pixel parallelisiert werden (Tracing Rays sind alle unabhängig voneinander).
Haben wir alle Informationen der Pixel gesammelt, so ergibt sich das fertige Bild.
Die ersten Strahlen die durch das Zentrum des Pixel gehen nennt man Primary Rays (oder Viewing Rays), ein Strahl der weiterverfolgt wird heißt Secondary Ray, usw.



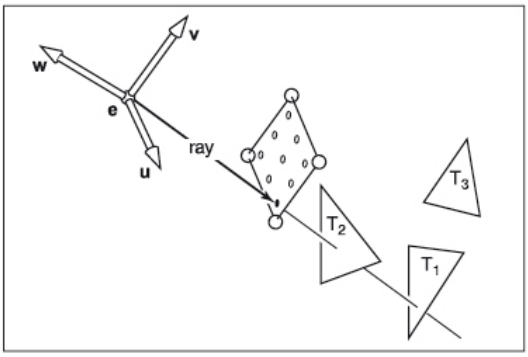
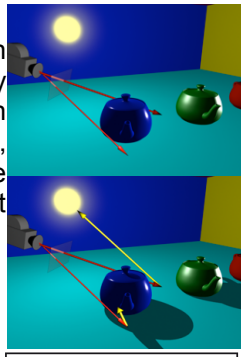


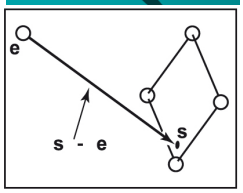
Bild 1: Die Viewing-Geometrie ist wie bisher festgelegt
Bild 2: Für jedes Pixel(-zentrum) wird ein Strahl durch die Viewing-Geometrie abgegeben
Bild 3: Die Strahlen werden in der Szene von Dreiecken geschnitten, weiterverfolgt und die Eigenschaften werden akkumuliert.

Ray Tracing vs. Ray Casting



Wenn nur Primary Rays berechnet werden und das Local Shading nur den Oberflächenpunkten bei Ray intersection zugeordnet ist, so spricht man von Ray Casting. Diese Technik benötigt kein Depthhandling oder Rasterisierung. Werden Strahlen tatsächlich „verfolgt“ (also getraced – Secondary Rays werden auch beachtet), spricht man von Ray Tracing. Dies erlaubt die Berechnung von Effekten wie Reflektionen, Schatten, Lichtbrechung, etc. (was nur mit Primary Rays alleine nicht möglich ist).
Bild 1: Ray Casting (roter Pfeil = Primary Ray)
Bild 2: Ray Tracing (gelber Pfeil = Secondary Ray)

Computing Viewing Rays



Strahlen können mit parametrisierten Geradengleichungen ausgedrückt werden, wobei e der Ursprung und s ein Punkt auf dem Bildschirm ist (auf der Viewing Plane).
p(0)=e, p(1)=s mit t<0, dann ist p hinter e. e ist gegeben und s kann durch Anwenden der Transformationsmatrizen berechnet werden.

$$p(t) = e + t(s - e)$$

Ray-Object Intersection

Wir wollen Schnittpunkte an Objekten finden, bei denen t>0 ist. Diese sind leicht zu finden, sofern eine implizite Oberflächengleichung gegeben ist. Bei den meisten Objekten ist dies aber nicht der Fall, wir müssen also die Ray Intersection in den meisten Fällen berechnen. Hierfür gibt es mehrere Möglichkeiten (wir verwenden baryzentrische Koordinaten und parametrisierte Ebenengleichungen). Wir können die Gleichungen (Strahl und Ebene) gleichsetzen und für den gemeinsamen Intersectionpoint lösen. Die Unbekannten sind t und die baryzentrischen Koordinaten, aber wir haben die 3 Gleichungen in x, y und z. Der Intersectionpoint p kann nun aus t berechnet werden, es muss vorher geprüft werden, ob sich p im Dreieck befindet.

Shadows

Wir möchten nun auch realistischere Schatten kreieren können (Soft Shadows). Diese werden erzeugt, wenn Objekte von Area-Light-Sources beleuchtet werden – es wird ein Kernschatten (umbra core shadow) und ein Penumbra Schatten erzeugt. Dies funktioniert folgendermaßen: mehrere Shadow-Rays werden in zufällige Richtungen gegen die Lichtquelle geschickt – manche werden absorbiert, manche nicht – ihr Feedback wird akkumuliert um die Intensität der Schatten bestimmen zu können.
Shadow Rays von Kernschatten werden immer geschnitten, solche ausserhalb des Kernschattens werden nie geschnitten. Shadow Rays von Punkten die sich in der Penumbra befinden werden mehr oder weniger geschnitten, proportional zu ihrer richtigen Schattenintensität.

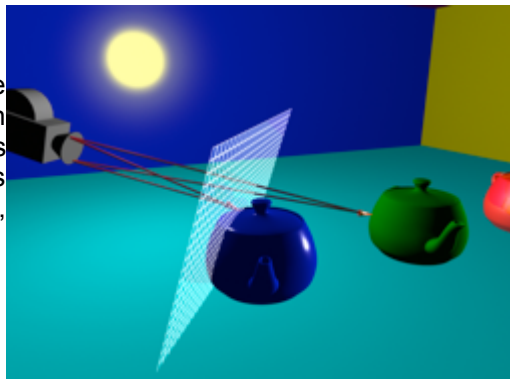
Reflection

Wenn ein Primary Ray geschnitten wird, müssen wir einen reflektierten Secondary Ray berechnen. Dies kann durch Bestimmung der Normalvektoren an den Schnittpunkten und anschließender Berechnung des reflektierenden Strahls bestimmt werden (Einfallswinkel = Austrittswinkel). Dieser Vorgang kann rekursiv wiederholt werden um Multi-Reflections zu berechnen.

Refraction

Wenn ein Primary Ray an einem transparenten Punkt geschnitten wird, so müssen wir einen gebrochenen (refracted) Secondary Ray berechnen. Dies wird erreicht, indem der Normalvektor am Schneidepunkt bestimmt wird, dazu muss man natürlich den Refraction Index des Materials kennen (Snell's Gesetz). Auch dieser Vorgang kann rekursiv wiederholt werden um Multi-Refractions zu berechnen. Es kann auch die Wellenlänge berücksichtigt werden – dies führt zu chromatischen Abweichungen – in diesem Falle müssten Refraction Rays für jede Farbe separat verfolgt werden.

Distribution Ray Tracing

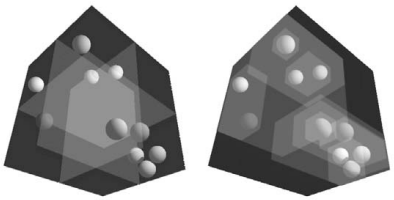


Das Senden von vielen, zufällig gewählten Strahlen, um die endgültigen Pixelwerte berechnen zu können, nennt man Distribution Ray Tracing („Verteilte Strahlverfolgung“). Dieses Verfahren erzeugt realistischere Bilder mit besserer Qualität – es unterstützt weiche Schatten, glänzende Reflektionen, Tiefeneffekte und Antialiasing.

Bidirectional Path Tracing
Darunter versteht man die Generierung von zufälligen Secondary Rays auch an zerstreuten Oberflächen um andere globale Beleuchtungseffekte (Color Bleeding, Caustics) zu erhalten. Es ist schneller und genauer als das normale Path Tracing. Generell gesehen ist Ray Tracing nicht sehr effizient, wenn es um die Erzeugung globaler Effekte an zerstreuten Oberflächen geht – besser hierfür geeignet wäre Radiosity.

Speeding Up Ray Tracing
Mit Hilfe von parallelen Prozessoren (RPUs, FPGA, Multicores, Clusters, GPUs) oder optimierten Datenstrukturen (Objektpartitionierung – Bounding Volume Hierarchiy BVH, Platzpartitionierung – BSP Trees).

Sub-Linear Ray-Object Intersection



Unsere bisherigen Verfahren verlangen, dass jeder Strahl auf jedes Dreieck getestet wird. Die Komplexität beträgt O(n) bei n Dreiecken – sehr ineffizient. Um das zu optimieren, können wir die Dreiecke in Bounding Boxes zusammenfassen. Entweder nach Objekten (object partitioning, rechts) oder nach Platz (space partitioning, links). Dies führt zu einer sub-linearen Komplexität. Die Strahlen werden gegenüber den Bounding Boxen getestet – nur wenn ein Strahl eine Box schneidet, werden auch die Dreiecke in der Box getestet. Bounding Boxen können axenausgerichtet (Weltkoordinatensystem) oder orientiert (an den Objektgrenzen orientiert) sein. Orientierte Bounding Boxen benötigen zusätzliche Transformationen, sind aber generell effizienter als axenausgerichtete Bounding Boxen.

Hierarchical Bounding Boxes
Dies ist ein Beispiel für Objektpartitionierung. Die Objekte werden nach Hierarchien gruppiert. Dies führt zu einer Baumstruktur mit größeren Boxen (Richtung Wurzel) und kleineren Boxen (Richtung Blätter). Jedes Objekt ist genau einem Zweig zugeordnet, aber mehrere Zweige können überlappenden Platz beinhalten. Hierarchical Bounding Boxes sind normalerweise unbalanciert, aber Traversieren und das Erzeugen der Boxen ist schnell und einfach.

Space Subdivision
Dies ist ein Beispiel für Platzpartitionierung. Eine einheitliche Unterteilung teilt den Raum in ein einheitliches Gitter. Strahlen werden gegen axenausgerichtete 3D Zellen getestet, bevor sie gegen die beinhaltenden Dreiecke getestet werden. Als Beispiel hierfür können wir BSP-Trees nennen. In diesem Fall wird der Raum rekursiv in axenausgerichtete Ebenen geteilt und die Szene wird sukzessiv in zwei Hälften für jede Ebene geteilt. BSP-Trees sind üblich für Ray Tracing.

Notes on GPU Ray Tracing
Profitieren von SIMD Maschinen (Single Instruction, Multiple Data).
Probleme sind:
- Effiziente Nutzung von parallelen Ausführungen ist sehr kompliziert
- Engpass: data traversal
- Echtzeit Datenstrukturaktualisierung für dynamische Szenen wird benötigt

Es gibt bereits viele Versuche, einzelne Teile von Ray Tracing Algorithmen auf die GPU auszulagern. Bessere Ressourcenkontrolle ist möglich mit zB Compute Unified Device Architecture (CUDA), etc.

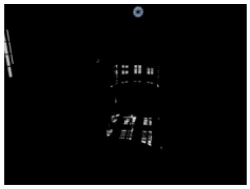
Beispielfragen zum Kapitel:
- Wie funktioniert die Berechnung von Reflexion und Schatten bei Raytracing?
- Diskutieren Sie (im Detail) was den Rechenaufwand bei Raytracing in erster Linie ausmacht, und wie man diesen optimieren kann. Beschreiben Sie konkrete Algorithmen und Datenstrukturen zur Beschleunigung von Raytracing.

5.2. Radiosity

ray tracing only (top) and
ray tracing + radiosity (bottom)

Ray Tracing als eine der ersten globalen Beleuchtungstechniken ist perfekt für das Rendern von spiegelnden Reflektionen, Strahlungen, Brechung und harten Schatten. Realistische unscharfe Reflektionen sind schwierig und teuer auf Strahlen- (Ray) Basis zB mittels Distribution Ray Tracing oder Path Ray Tracing (→ Kapitel 5.1).

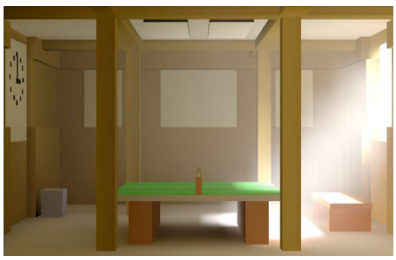
Einsatzgebiet Radiosity (effektive Strahlung)
Realistische Streuung, unscharfe Reflektionen, Color Bleeding (zB die Oberflächenfarbe eines blauen Kastens sichtbar auf der gegenüberliegenden weißen Wand), weiche Schatten, vorwiegend für Indoor-Szenen; Radiosity ist aber nicht geeignet für spiegelnde Reflektionen oder Brechung.
Radiosity und Ray Tracing sind demnach komplementär (sie ergänzen sich)

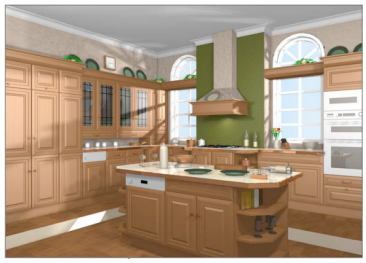


Ray Tracing (links) ist für gestreute/unscharfe gegenseitige Reflektionen (Inter-Reflexions) nicht ausreichend.

Lösung mit Radiosity (rechts)

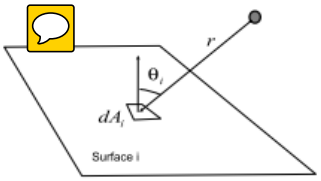
Typische Radiosity Beispiele:





Prinzipien
Basierend auf den Prinzipien der Wärmeübertragung
Teilung der Szene in Oberflächenflecken (Surface Patches) – entweder einheitlich oder uneinheitlich
Sogar Annäherung von Lichtquellen mittels Patches
Berechnet wird die Übertragung von Lichtenergie von Patch zu Patch.
Kosten des Algorithmus: O(n²) für n Patches
Im Gegensatz zum Ray Tracing, als Bildraum Algorithmus,
Durchführung von Radiosity-Berechnung im Objekt-Raum.

Point-to-Surface Form Factor
Wiederholung Lambertian Shading Model: Der Anteil von Strahlungsenergie, der mit einem bestimmten Winkel (pro Einheit; d.h. mit bestimmter Richtung) von einer Punktlichtquelle ausgeht und auf einer Einheit (Unit Area) der Oberfläche ankommt ist proportional zum Eingangswinkel des Strahls (je steiler umso heller = mehr Licht) und indirekt proportional zur Entfernung zwischen Oberfläche und Lichtquelle (je weiter umso weniger Licht)

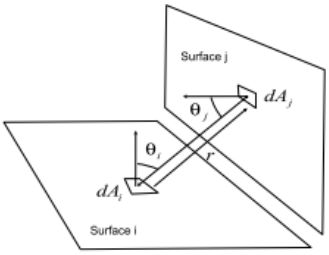


Definition Form Factor: Anteil von Strahlungsenergie, der eine Unit Area des Ziels trifft, dividiert durch die gesamte Strahlungsenergie ausgesandt von der Quelle
Punktlichtquellen strahlen Licht in alle Richtungen aus, d.h. kugelförmig über eine Winkel von 2π.
2π ist konstant und kann "implizit berücksichtigt werden", indem die Intensität der Lichtquelle für die Beleuchtung damit skaliert wird. D.h. es wird einfach durch diesen gesamten Winkel dividiert um die Beleuchtung für eine Einheit zu erhalten.

Engergietransfer von Punkt zu unendlich kleiner Fläche:
dA_i ... differential Area = unendlich kleine Fläche

$$F_i dA_i = \frac{\cos(\theta_i)}{r^2} \longrightarrow F_i dA_i = \frac{\cos(\theta_i)}{2\pi r^2}$$

Surface-to-Surface Form Factor
Ist die Strahlungsenergie ausgesandt von der Lichtquelle und auftreffend auf die Oberfläche jeweils pro Unit Area dividiert durch gesamte ausgestrahlte Energie ABER die Lichtquelle strahlt nur halbkugelförmig aus!

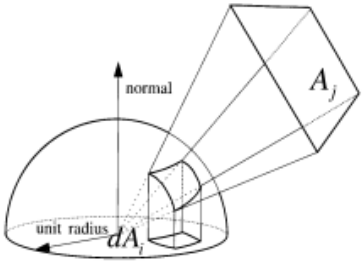


Form Faktor zwischen differentialen Oberflächen: $F_{ji}dA_i dA_j = \frac{\cos(\theta_i)\cos(\theta_j)}{\pi r^2}$

Duality of Light Transport: Form Faktoren zwischen Oberflächen sind reziprok/gegenseitig, d.h. in beide Richtungen wird die gleiche Lichtmenge übertragen, egal welche Oberfläche als Quelle und welche als Ziel angenommen wird:

The Nusselt Analog (Die Nusselt Nachbildung)
Eine von Nusselt entwickelte, geometrische Nachbildung zur einfachen und genauen Berechnung von Form Faktoren zwischen einer Oberfläche und einem Punkt einer zweiten Oberfläche.
Verwendung: Zur Ermittlung von Formfaktoren zwischen größeren, so genannten Non-Unit Area Oberflächen.

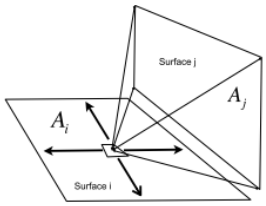
Vorgehen:
Platzierung eines $F_{ji}dA_j = F_{ij}dA_i$ Halbkugel-förmigen Projektionskörpers mit Einheitsradius auf einem Punkt einer Oberfläche (i)
Kugelförmige Projektion der 2. Oberfläche (j) auf diesen Projektionskörper
Zylindrische Projektion auf die Grundfläche (Basis) der Halbkugel
Der Form Faktor für die unendlich kleine Fläche dA_i gleicht der Fläche auf der Basis der Halbkugel dividiert durch die gesamte Basisfläche (diese ist π für r=1).



Form Faktor zwischen unendlich kleiner Fläche dA_i und Oberfläche A_j:

$$F_{ji}dA_i A_j = \int_{A_j} \frac{\cos(\theta_i)\cos(\theta_j)}{\pi r^2} V_{ji} dA_j$$

Binary Visibility Factor: Für die korrekte Beleuchtung von Oberflächen, die nicht direkt von einer anderen bestrahlt werden (nicht direkt von ihr sichtbar sind), weil sie zB verdeckt sind, wird ein binärer Sichtbarkeitsfaktor V_{ji} hinzugefügt:



Problem: Die Projektion von A_j auf die Basis der Halbkugel ist nicht für alle Punkte der Oberfläche A_i gleich!
Für dA_i wird der Formfaktor aus den projizierten Eckpunkten (A, B, C, D) von A_j angewandt – jedoch treffen die Strahlen von den (Eck-)Punkten von A_j in unterschiedlichen Winkeln auf die Halbkugel auf, wodurch unterschiedlich viel Licht auftrifft! (→ Wiederholung Lambertian Shading Model)

Lösung: Bildung des doppelten Integrals um den exakten Form Faktor zwischen zwei Oberflächen A_i und A_j zu bestimmen, das entspricht dem Umlegen der Strahlung der gesamten Fläche A_j auf die gesamte Fläche A_i.
gesamter Form Faktor zw. A_i und A_j:

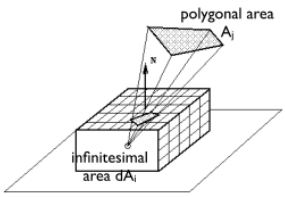
$$F_{ji}A_i A_j = F_{ji} = \frac{1}{A_i}\int_{A_i}\int_{A_j} \frac{\cos(\theta_i)\cos(\theta_j)}{\pi r^2} V_{ji} dA_j dA_i$$

Problem: Integral über beide Oberflächen ist viel zu rechenaufwändig

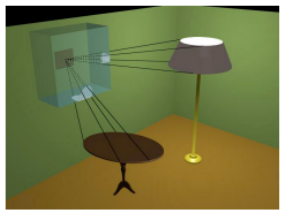
The Hemicube Method
= Annäherung an den Nusselt Analog
Prinzipien:
Bestimmung des Form Faktors zwischen einem Punkt dA_i und einem Polygon A_j
Hemicube = würfelförmige Annäherung an die Halbkugel, da die Projektion auf flache Planes weniger rechenaufwändig sind

Vorgehen:
Zentrierung des Hemicubes auf dA_i
Der Einfachheit halber wird ein Würfel mit 1 Einheit Höhe, 2x2 Einheiten für oben und somit 2x1 (w x h) seitlich angenommen.

Unterteilung des Würfels in Zellen mit bestimmter Auflösung (zB 512x512 für oben)
Projektion aller Polygone der Szene auf die Würfelzellen: auf gegenseitig orthogonale Planes (VERMUTUNG: d.h. weder von Würfel zu Polygon noch umgekehrt kommt es zur Skalierung aufgrund der Entfernung) inkl. den Z-Buffer-Werten für die Sichtbarkeit. Der Index des Patches (Polygon) wird für einen schnellen Look-Up in der jeweiligen Zelle gespeichert.

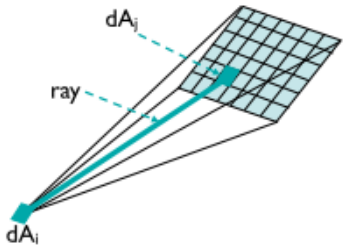


Form Faktor eines Polygons = Summe aller bedeckten Zellflächen (teilweise seitlich und oben unterschiedlich gewichtet) = Berechnung der Beleuchtung (Summe der hellblauen Flächen) – Beleuchtung noch nicht erfolgt!



Vorteil: Die Hemicube Methode ist kompatibel mit CG Techniken, sie kann also auf der Grafikkarte implementiert werden (als einfache Projektionen auf Planes).
Nachteil: Fehler aufgrund der diskreten Annäherung zB Aliasing Fehler aufgrund des Abtastens (Sampling), Sichtbarkeitsfehler, Rundungsfehler; Die Fehleranfälligkeit ist abhängig von der Auflösung, eine zu hohe Auflösung führt allerdings zu Ineffizienz.

Area Sampling
die heutzutage gebräuchliche Technik (früher war es die Hemicube Methode)
Nachteil: langsamer als die Hemicube Methode
Vorteil: für jedes Polygon so genau möglich, wie gewünscht

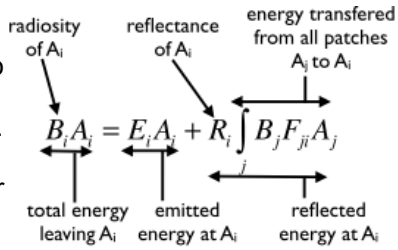


Prinzip: Die Auflösung jedes einzelnen Polygons kann nach Bedarf angepasst werden.

Algorithmus:
Nach der Unterteilung von A_j in unendlich kleine Einheiten (Unit Areas), wird von jeder Einheit dA_j ein Strahl zur Lichtquelle dA_i gesandt (ebenfalls unendlich kleine Unit Area).
Das wird für alle Unit Areas der Lichtquelle durchgeführt, um festzustellen, ob dA_j von der jeweiligen dA_i beleuchtet (sichtbar) ist.
Ist dies der Fall, wird der Form Faktor zwischen je 2 Unit Areas berechnet und zum bestehenden Form Faktor für die gesamte Fläche addiert. (Müsste lt. dem Surface-to-Surface Form Factor Kapitel auch für gegenseitige Beleuchtung, d.h. Lichtquelle wäre dA_j, funktionieren.)

```
subdivide Aj into small pieces dAj
for all dAj
    cast ray dAj-dAi to determine Vji
    if visible
        compute FdAidAj
        sum up FdAiAj +=FdAidAj
    endif
endfor
```

Radiosity Theory
Definition Radiosity B:
Radiosity ist die Energie pro Unit Area, welche den Oberflächen Patch pro Zeiteinheit verlässt.
Es ist die Summe des reflektierten (R) und direkt ausgesendeten (E ... emitted) Lichts auf einem Oberflächen Patch.
Es gilt demnach: Radiosity * Fläche = ausgesendete Energie * reflektierter Energie



Problem: Rekursion: Für die Berechnung der Radiosity der Oberfläche O1, ist die von einer anderen Oberfläche O2 erhaltene Energie (also deren Radiosity) heranzuziehen um die Reflektion zu berechnen. O2 würde jedoch möglicherweise die Radiosity von O1 benötigen, um die Reflexion für sich selbst zu berechnen.
Lösung:
Bekannter Weise ist der Form Faktor der beiden Oberflächen reziprok ist (→ Surface-to-Surface Form Faktor):

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

D.h. in der Gleichung kann $F_{ji}A_j$ mit $F_{ij}A_i$ ersetzt werden:

$$B_i A_i = E_i A_i + R_i \int_j B_j \boxed{F_{ij} A_i}$$

Durch Dividieren beider Seiten der Gleichung durch A_i ist diese nun Oberflächen-unabhängig:

$$B_i = E_i + R_i \int_j B_j F_{ij}$$

Nach Diskretisierung der Gleichung (Ersetzen der Integration durch Summierung) erhält man eine lineare Gleichung!

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

Durch Betrachtung dieser Gleichung für jeden Patch n erhält man ein lineares Gleichungssystem:

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

Ist der Sichtbarkeits-Faktor V_{ij} (→ der Nusselt Analog) 0, ergibt sich $F_{ij}=0$, somit $-R_i*F_{ij}=0$, d.h. einzig die Diagonale im Gleichungssystem ist ungleich 0.

$$\underbrace{\begin{bmatrix} 1-R_1F_{11} & -R_1F_{12} & \dots & -R_1F_{1n} \\ -R_2F_{21} & 1-R_2F_{22} & \dots & -R_2F_{2n} \\ \vdots & \vdots & \dots & \vdots \\ -R_nF_{n1} & -R_nF_{n2} & \dots & 1-R_nF_{nn} \end{bmatrix}}_{\text{known}} \underbrace{\begin{bmatrix} B_1 \\ B_2 \\ \vdots \\ B_n \end{bmatrix}}_{\text{unknown}} = \underbrace{\begin{bmatrix} E_1 \\ E_2 \\ \vdots \\ E_n \end{bmatrix}}_{\text{known}}$$

$$M_{rf} \cdot b = e$$

$$b = e \cdot M_{rf}^{-1}$$

Die Lösung des Gleichungssystems nach b erfolgt aufgrund deren Ineffizienz (viele Gleichungen!) normalerweise nicht mit direkten Methoden wie zB Gauß'scher Elimination (Aufwand = $O(n^3)$) sondern iterativ:

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

$$\begin{bmatrix} X \\ \\ \\ \end{bmatrix} = \begin{bmatrix} X \\ \\ \\ \end{bmatrix} + \begin{bmatrix} X & X & X & X & X \\ & & & & \\ & & & & \end{bmatrix} \begin{bmatrix} X \\ X \\ X \\ X \\ X \end{bmatrix} \text{gathering}$$

$$B_i = E_i + \sum_{j=1}^{n} \left(R_i F_{ij}\right) B_j$$

Gathering: verwendet direkte Methoden mit Aufwand $O(n^2)$ zB Gauß-Seidel oder Jacobi (auf GPU möglich)
Prinzip: Licht, welches einen Patch verlässt, wird berechnet als die Zusammenfassung (Gathering) des Lichts der restlichen Umgebung.
Zu Beginn gilt für einen selbst-strahlenden Patch: B=E (für einen bestimmten Patch i: $B_i=E_i$)
In jedem Iterationsschritt wird DIESER EINE PATCH i mit den Licht-Beiträgen aller anderen (j=1..n) UPGEDATED.
Das ist iterativ zu wiederholen, bis man sich ausreichend an die Lösung angenähert (→ Progressive Radiosity) hat.

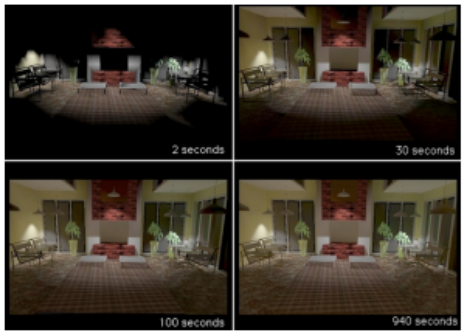
$$B_i = E_i + R_i B_j F_{ij}$$

$$\begin{bmatrix} X \\ X \\ X \\ X \\ X \end{bmatrix} = \begin{bmatrix} X \\ X \\ X \\ X \\ X \end{bmatrix} + \begin{bmatrix} X & & & & \\ X & & & & \\ X & & & & \\ X & & & & \\ X & & & & \end{bmatrix} \begin{bmatrix} X \\ \\ \\ \\ \\ \end{bmatrix} \text{shooting}$$

$$B_i = E_i + (R_i F_{ij}) B_j$$

Shooting: verwendet iterative Methoden mit Aufwand $O(n^2)$ zB Southwell
Prinzip: Licht wird von einem Patch geschossen (Shoot) um die restlichen upzudaten.
Zu Beginn gilt für einen selbst-strahlenden Patch: B=E ($B_i=E_i$)
In jedem Iterationsschritt werden ALLE PATCHES i, DIE NOCH NICHT GESCHOSSEN haben, mit dem Licht-Beitrag VOM AKTUELLEN PATCH j UPGEDATED.
Es ist möglich, die Reihenfolge der schießenden Patches zu sortieren, damit solche, die mehr Energie ausstrahlen zuerst schießen!
Das ist ebenfalls iterativ zu wiederholen, bis man sich ausreichend an die Lösung angenähert (→ Progressive Radiosity) hat.

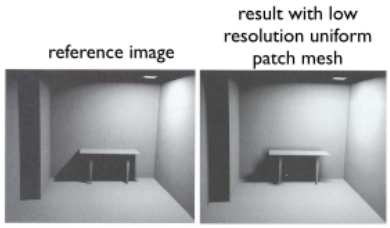
Progressive Radiosity
Vorteil der iterativen Lösung: Es gibt in jedem Iterationsschritt ein Ergebnis, welches sich dann schrittweise verbessert.



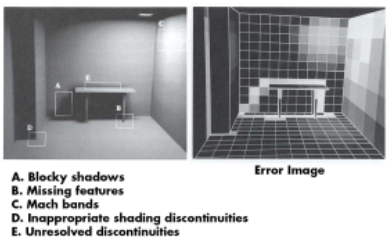
Unterschiedliche Bedingungen, wann eine Lösungs-Annäherung ausreichend ist, sind möglich zB kann solange eine schrittweise Verfeinerung erfolgen, bis sich die Szene nicht mehr merkbar ändert.

Jeder neue Frame liefert ein besseres Bild. Das bedeutet auch, dass die Frame-Rate indirekt-proportional zur Zeit einer Iteration steigt. (Es werden immer mehr Details gerendert!)

Patch Subdivision
Ausgangssituation/Problematik:
Die Unterteilung einer Szene in gleich große Patches führt bei zu geringer Patch-Auflösung zu Artefakten, vor allem Schattenregionen erscheinen blockartig und unstetig/unterbrochen.



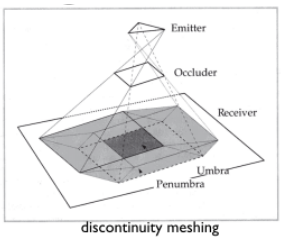
Schlechte Lösung: Steigerung der geometrischen Auflösung
Problem: maßgeblicher Performance-Verlust
Bessere Lösung: lokale Anpassung des Patch Netzes (Patch Mesh) Prinzip: Steigerung der Auflösung, nur dort, wo es nötig ist und belassen der niedrigen Auflösungen bei den restlich Regionen.



Meshing Strategien (Netzstrategien)

Discontinuity Meshing:

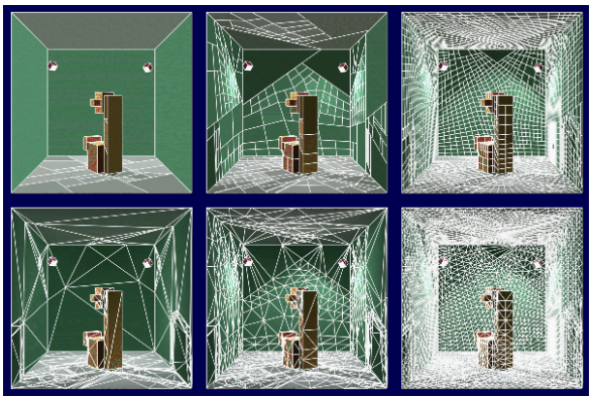
Vorhersage von Unstetigkeiten (Discontinuities), zB in Schattenregionen, bevor die Radiosity-Auflösung berechnet wird.
Verfeinerung des Netzes erfolgt an Stellen großer Unstetigkeit. Solche Stellen sind im Halbschatten (Penumbra). (Falls sie vollständig im Schatten (Umbra) liegen, d.h. aufgrund einer vollständigen Verdeckung (Occluder) gelangt kein Lichtstrahl auf die Oberfläche, ist eine Verfeinerung nicht nötig).

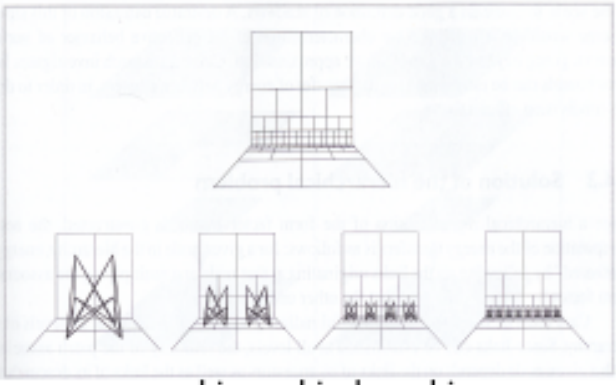


discontinuity meshing

Adaptive Meshing:

Verfeinerung einer schlechten Version während dem Fortschreiten des Radiosity Lösungsprozesses (siehe Progressive Radiosity)
Besitzen die Sub-Patches der nächst höheren Auflösung starke Unstetigkeiten, wird der Patch weiter unterteilt.

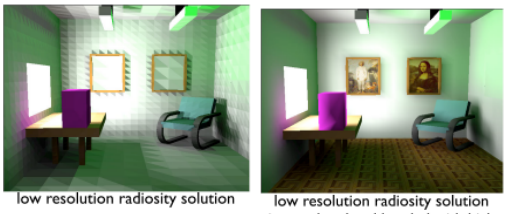



hierarchical meshing

Zusammenfassung Radiosity

1. Unterteilung der Oberfläche in ein Netz aus Patches – einheitliche oder angepasste (adapted)
2. Berechnung des Form Faktors zwischen den Patches – Hemicube oder Area Sampling
3. Lösen des linearen Gleichungssystems – direkt oder iterativ
4. Rekonstruktion/Umbau der Lösung, d.h. Interpolation und Mischen (Blending) der gering aufgelösten Radiosity Lösung mit dem hoch aufgelösten Szenen-Bild

Radiosity ist Sicht-unabhängig, d.h. eine erneute Berechnung ist nur erforderlich, wenn sich die Szene ändert, nicht aber bei bloßer Änderung der Kamera.
Radiosity kann von der GPU berechnet werden! (Referenz auf den Folien: http://realtimeradiosity.com)


low resolution radiosity solution

low resolution radiosity solution interpolated and bended with high resolution scene image

Beispielfragen zum Kapitel:

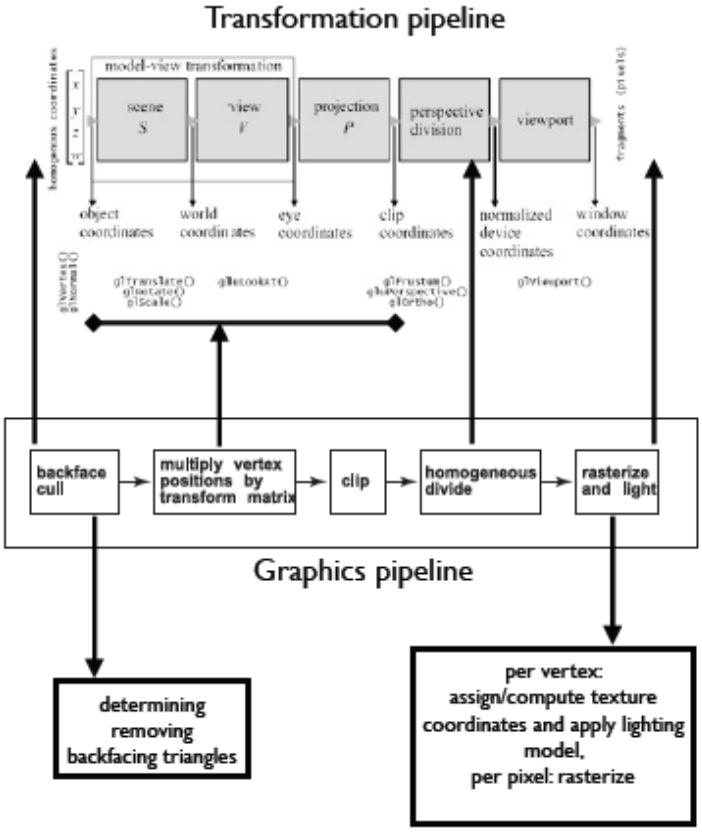
- Wie ist das Prinzip von Radiosity? Laufzeit von Radiosity?
- Warum muss man nach Radiosity - Lösung das Bild noch mal rekonstruieren?
- Muss man Radiosity neu berechnen, wenn sich die Kamera ändert?
- Beschreiben Sie (in aller Kürze) den Unterschied zwischen Radiosity und Raytracing. Worin sind diese beiden Verfahren komplementär?
- Erklären Sie (in aller Kürze) wie Formfaktoren effizient für das Radiosity – Verfahren berechnet werden können.

6. Graphics Pipelines and Scene Graphs

Zusammenführen der Bits und Pieces
Bereits besprochen für Echtzeit Grafiken:

- Transformationen (Szene, Ansicht, Projektionen, Viewport → Transformationspipeline)
- Depth-Handling, Backface Eliminierung/Culling und Rasterisierung
- Lighting und Shading
- Texture Mapping

Fügt man diese Komponenten in richtiger Reihenfolge zusammen, führt das zur Grafikpipeline.
Das Ziel ist, so wenig Dreiecke wie möglich durch die Grafikpipeline zu senden, um eine optimale Performance zu erreichen.



Clipping vs. Culling
Sind Dreiecke im nicht sichtbaren Bereich oder vollständig außerhalb des Betrachtungsvolumen, dann werden diese gänzlich aus der Pipeline entfernt (d.h. aussortiert (culled)).
Schneiden sichtbare Dreiecke das Betrachtungsvolumen, werden sie abgeschnitten (clipped).
In der Praxis ist culling sehr teuer, wenn es für jedes Dreieck einzeln ausgeführt wird. Culling in abgeschlossenen Volumen (Kugel oder Box) ist effizienter.

Backface Culling
Das funktioniert gut für geschlossene Flächen, aber nicht für offene. Im letzteren Fall sind Dreiecke sichtbar oder nicht sichtbar (durch die Anordnung von 2 Dreiecken mit umgekehrten Normalen – siehe Z-Buffer fighting) oder Backface Culling ist ausgeschaltet (das ist schlecht für die Performance von anderen geschlossenen Flächen in der Szene)
Alternative:

1. Backface Culling einschalten und nur die geschlossenen Flächen rendern
2. Backface Culling ausschalten und den Rest rendern.

Clipping
Clipping entfernt Teile von Dreiecken welche den Betrachtungskegel schneiden oder (teilweise) außerhalb sind. Das kann in der Pipeline durchgeführt werden, nachdem die Dreiecke die Transformationspipeline durchlaufen haben und im Kamerakoordinatensystem sind (also vor der Perspective Division) – deshalb werden diese Koordinaten „Clip Coordinates“ genannt.

$$\begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -fn \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} nx \\ ny \\ (n+f)z - fn \\ z \end{bmatrix}$$

$$\begin{bmatrix} nx \\ ny \\ (n+f)z - fn \\ z \end{bmatrix} / z = \begin{bmatrix} \frac{n}{z}x \\ \frac{n}{z}y \\ n+f-\frac{fn}{z} \\ 1 \end{bmatrix}$$

perspective transform

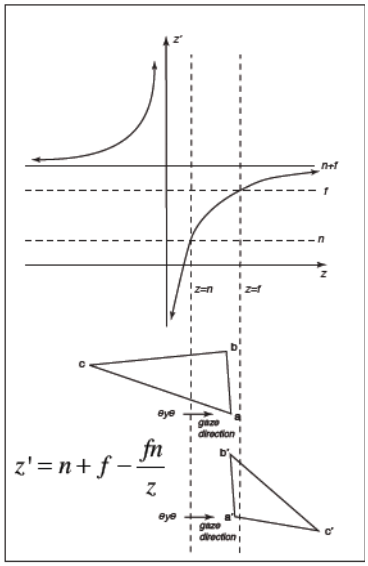
Warum wird Clipping vor der Perspective Division durchgeführt (1) und nicht danach (2)?

Beides ist möglich, solange die korrekten Flächengleichungen verwendet werden (die homogene Koordinate w ist angemessen zu berücksichtigen).
Das Problem nach der Perspective Division ist:

- Die Tiefenreihenfolge ist nach der Perspective Division gut erhalten, für Tiefen größer als Null (gut für Z-Buffering).
- Es ist unterbrochen bei Tiefe Null.
- Wenn z sich vom positiven zu negativen Werten bewegt, z' ändert sich vom Negativen ins Positive, siehe nachfolgende Abbildung

plane equations for (1)	plane equations for (2)
$-x+lw=0$	$-x+l=0$
$x-rw=0$	$x-r=0$
$-y+bw=0$	$-y+b=0$
$y-tw=0$	$y-t=0$
$-z+nw=0$	$-z+n=0$
$z-fw=0$	$z-f=0$

Beleuchtung
siehe erste Grafik im Kapitel: „rasterize and light“ am Ende der Pipeline.
In der Tat könnte die Beleuchtungsberechnung schon früher sein (zB nach der Szenentransformation, wenn wir noch im Weltkoordinatensystem sind). Das war auch so in traditionellen Pipelines der Fall, welche simple Per-Vertex Shading Modelle, wie Gouraud Shading verwendeten.
Aber für komplexere Shading Modelle, wie Phong Shading mit normaler Interpolation, muss die Schattierung auf Per-Pixel Basis berechnet werden – das ist der Trend in modernen Pipelines.



(Diese Grafik gehört noch zu Clipping!)

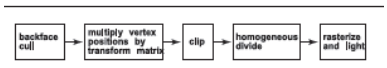
Dreiecksstreifen und -fächer
Die Rendering Performance kann nicht allein durch Reduzieren der Dreiecke erhöht werden, sondern auch durch Reduzieren der Ecken. Oberflächen bestehen meist aus einer Anzahl von verbundenen Dreiecken. Wenn Dreiecke verbunden sind, haben sie gemeinsame Ecken. Es macht keinen Sinn, Knoten mit gleichen Koordinaten mehrmals zu verarbeiten. Dazu bietet die Grafikpipeline einen effizienten Umgang mit Netzstrukturen, wie Fächer und Streifen. In der Praxis kann ein Speed-up vom Faktor 3 erreicht werden wenn Dreieckstreifen als Einzeldreiecke verwendet werden.

Erhaltene Zustände
Attribute wie Farbe oder Materialeigenschaften für jedes Dreieck oder jede Ecke einzeln kosten eine erhebliche Menge an Zeit. Das ist der Grund, warum viele Pipelines die gemeinsame Nutzung von Attributen und Zuständen vieler Dreiecke anbieten (OpenGL zum Beispiel ist eine Zustandsmaschine) – dies kann erhebliche Speed-ups erzielen.
Ein anderer Zustand, der gemeinsam genutzt werden kann, ist die Geometrie: Das ist dann vorteilhaft, wenn diese statisch ist, und sich nicht von Frame zu Frame ändert. Es wird in sogenannten Display Lists gespeichert. Display Lists sind effizient, weil sie auf der Grafikkarte gespeichert werden (sie müssen nicht ständig über den Bus übertragen werden) und sie können automatisiert optimiert werden (zB Dreiecke, die gemeinsame Ecken nutzen, werden gruppiert, sodass sie in der gleiche Aufruffolge mit gemeinsamen Attributen existieren).

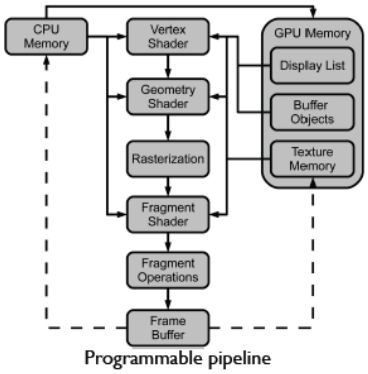
Without state-machine:
set triangle atributes T1
draw triangle T1
set triangle atribute T2
draw triangle T2
...

With state-machine:
set state triangle atribut
draw triangle T1
draw triangle T2
...

Fixe Funktionen vs. Programmierbar
Durch die Zustände müssen wir aber auch mit Flexibilität bezahlen! Traditionelle Pipelines teilen viele Zustände (zB Transformationen, Beleuchtung, Parameter, etc.). Das ist auch der Fall bei OpenGL – solche Pipelines werden auch „Fixed Function Pipelines“ genannt. Viele Zustände wie Transformationen sind gleich für alle Knoten in „Fixed Function Pipelines“ - sie können während des Renderns eines Frames nicht geändert werden. Neue programmierbare Pipelines erlauben mit Shadern individuelle Knoten-, Fragmente- und Geometrie-Operationen. Flexibler Texturzugriff ist in allen Shadern möglich. In Zukunft wird die Rasterisierungseinheit möglicherweise auch programmierbar sein (zur Zeit nicht der Fall).



Fixed function pipeline



Programmable pipeline

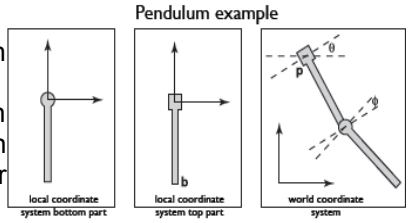
Entwicklung von Grafikhardware
GPUs verwandelten sich in parallele Hochleistungsprozessoren
Nach 2000:
- Einführung von Texturkombinierern, die neue (Multi-) Texturberechnungen ermöglichen
- Shader-Einheiten wurden eingeführt zum freien Programmieren von (1) Eckentransformationen, (2) Fragmentverarbeitung und letztlich auch (3) Geometrieaufbau.
- Heutzutage sind diese Ecken-, Fragment und Geometrieshader alle als einheitliche Shader in der Hardware implementiert, die willkürlich für eine optimale Lastverteilung in Abhängigkeit von dem angewandten Algorithmus verwendbar sind.

Die Rechenleistung erhöhte sich deutlich während des letzten Jahrzehnts und die GPU-Transistoren zählen skaliert sogar schneller als Moores Gesetz (Komplexität der ICs verdoppelt sich alle 2 Jahre).
Aktuell ist die Fließkomma-Leistung von 2,5 TFLOPs (in einfacher Genauigkeit) einer einzelnen GPU bereits überschritten. Das macht sie immer mehr interessant als die Mehrzweck-Hochleistungsvektorprozessoren.

GPU Programming Languages
Zusammen mit Grafikhardware, haben sich neue Programmiersprachen entwickelt. Assemblersprachen waren kurz nach der Einführung von programmierbaren Shadern verfügbar. Heute haben wir verschiedene High-level – Shader-Sprachen (Cg von Nvidia, GLSL für OpenGL (herstellerunabhängig), HLSL für DirectX).

General Purpose GPU (GPGPU)
Die zunehmende Fließkomma-Performance von GPUs hat sie mehr und mehr interessanter für andere Berechnungen neben Standardgrafikalgorithmen gemacht. Mehrere Sprachen wurden für GPGPU entwickelt (Nvidia CUDA, ATI Stream, DirectX Compute Shaders).
Als Widerstand gegen den Fortschritt mehrerer unvereinbarer GPGPU-Programmiersprachen von einzelnen GPU-Herstellern wurde die Spezifikation der Open Computing Language, OpenCL im Jahr 2008 vorgeschlagen. Die Entscheidung von ATI, Nvidia, Apple und anderen Firmen die OpenCL-Spezifikation zu unterstützen wird den zukünftigen Standard für allgemeine Berechnung auf der GPU ermöglichen.
Höhere Abstraktionsebenen für Grafikprogrammierung gibt es zusätzlich: Szenengraphen

Gekapselte Transformationen
Implementieren von komplexeren, eingekapselten Transformationen von verbundenen Komponenten, wie das Pendel rechts.
Wir können davon ausgehen, dass jede Komponente in einem eigenen lokalen Koordinatensystem transformiert wird und dass die lokalen Koordinatensysteme in Bezug auf die hierarchische Verbindung der Komponenten transformiert werden.
Zum Beispiel: der obere Teil des Pendels ist in seinem lokalen Koordinatensystem transformiert (das führt zu M_3) und der untere Teil wird in einem eigenen Koordinatensystem transformiert (M_c) – der untere Teil ist abhängig vom oberen Teil, daher muss das untere Koordinatensystem mit dem oberen mitbewegt werden ($M_d = M_3 * M_c$).
Das führt zu einer hierarchischen Transformationsstruktur, welche all diese Transformationen und lokalen Koordinaten leicht verwaltet. Die richtige Transformation, die für ein bestimmtes Objekt in dieser Datenstruktur benötigt wird, ist einfach das Produkt aller Transformationen auf dem Weg von dem Wurzelknoten zu den Objektknoten. Die einzelnen Transformationen innerhalb eines einzelnen Knotens sind die lokalen Transformationen in dem lokalen Koordinatensystem des Objekts. Alle Transformationen akkumuliert vom Wurzelknoten bis zu den Objektknoten ergeben eine zusammengesetzte Transformation im Weltkoordinatensystem.



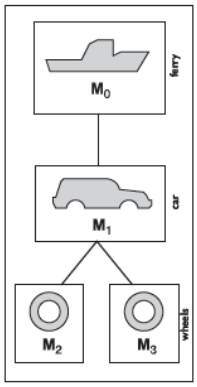
Top part:
M_1=rotate(θ)
M_2=translate(p)
M_3=M_2M_3
apply M_3 to all points of upper part

transformations in local coordinate systems

Bottom part:
M_a=rotate(ϕ)
M_b=translate(b)
M_c=M_bM_a
M_d=M_3M_c
apply M_d to all points of lower part

hierarchical update

Scene Graphs Basics
Diese hierarchische Datenstruktur kann beliebig komplex sein – mit mehreren untergeordneten Knoten. Es ist eine Baumstruktur. Dennoch ist das Prinzip immer das gleiche: Transformationen innerhalb einzelner Knoten sind Veränderungen im lokalen Koordinatensystem des Objekts. Das Akkumulieren dieser von oben nach unten führt zu der zusammengesetzten Transformation in das Weltkoordinatensystem.
Szenengraphen sind effizient implementiert mithilfe von Matrix-Stacks:
- push() und pop() löschen und fügen Matrizen von der rechten Seite vom Matrix-Produkt hinzu
- Sie können zum Durchlaufen eines Szenengraphen verwendet werden.



Example:
transformation of right wheel in world coordinates is $M_0M_1M_3$ and of left wheel $M_0M_1M_2$

Example:
push(M_0)
push(M_1)
push(M_2)
leads to $M_0M_1M_2$

Die modernen Szenengraph – Architekturen erben nicht nur Transformationen, sondern auch Zustände wie Beleuchtung, Material usw. Sie können mithilfe von einer Vielzahl von verschiedenen Arten von Knoten modelliert werden.

traverse(node)
push(M_{local})
draw(object using composite matrix from stack)
traverse(left child)
traverse(right child)

Beispiele für Bibliotheken, die Szenengraphen implementieren: Open Inventor, OpenSG (OpenSceneGraph)
Viele CG Applikationen basieren auf Szenegraphen für visuelle Programmierung.

Beispielfragen zum Kapitel:
- Warum findet die Beleuchtung erst am Ende der Pipeline statt?

7. Human Vision and Basic Displays

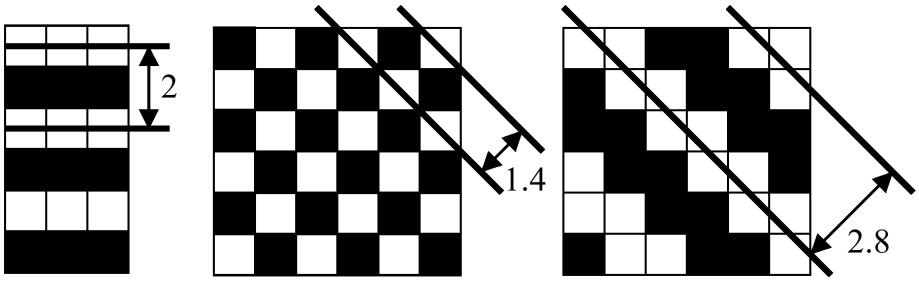
Wenn wir die Grafikpipeline durcharbeiten kommen wir von einem 3D Bild zu einem 2D Bild, welches auf dem Bildschirm dargestellt wird, dies ist aber nicht das Ende der Verarbeitung – die zur Verfügung gestellten Bildschirmtechnologien beeinflussen die vorhergegangenen Schritte in der Pipeline genauso wie der Bildschirm selbst.

Displays and Graphics
Es gibt sehr viele verschiedene Bildschirmtechnologien (LCD, CRT, LED, etc.). Zukünftig werden weitere Technologien am Markt erscheinen, so zB große stereoskopische Displays (3D Bilder ohne 3D-Brillen), volumetrische Displays oder holografische Displays (echte Wellenfrontgenerierung).
Für manche dieser Displays wird sich die Art, wie wir mit Computergrafik umgehen, radikal ändern (bei Hologrammen zB wird keine Rasterisierung mehr benötigt.)

Perception and Graphics
Wenn man bei der Entwicklung von Rendering-Algorithmen die menschliche Wahrnehmung beachtet, könnte dies helfen, über die technischen Limitation von Displays, Speicher und Prozessoren hinweg zu kommen.

- Man kann die Rendergeschwindigkeit erhöhen, wenn man die Einschränkungen des menschlichen Sehens beachtet (Wahrnehmungsgeschwindigkeit/Unterscheidungskraft)
- Man kann Speicherplatz mithilfe von Bildkomprimierung sparen
- Wahrgenommene Bildqualität kann mit Kontrasterhöhung erreicht werden
- Man kann überprüfen, ob dargestellte Bilder korrekt vom Menschen aufgenommen werden
- Man kann Informationen im Bild verstecken (Wasserzeichen, zeitliche Codierungen)
- Entwicklung realistischerer Rendertechniken ist möglich

Display Resolution
Die Bildschirmauflösung ist eine der wichtigsten Maße von Rasterdisplays. Normalerweise werden die vertikalen und horizontalen Pixelmaße gezählt. (Anzahl der einzelnen Linien wäre eigentlich genauer). Die diagonale Auflösung kann mit der Quadratwurzel der Abstände der horizontalen Linien ermittelt werden (wenn die Pixel quadratisch sind). Unter gewissen Wahrnehmungsaspekten ist die Winkelauflösung wichtig.



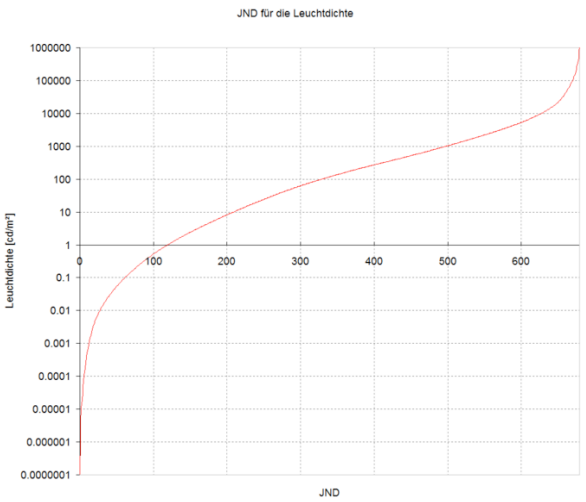
Horizontale und vertikale Auflösungen

name	resolution (HxV)	aspect ratio
VGA	640 x 480	4 : 3
SVGA	800 x 600	4 : 3
XGA	1024 x 768	4 : 3
WXGA	1365 x 768	16 : 9
SXGA	1280 x 1024	5 : 4
WSXGA	1680 x 1050	16 : 10
UXGA	1600 x 1200	4 : 3
WUXGA	1920 x 1200	16 : 10
QXGA	2048 x 1536	4 : 3

The Resolving Power of the Eye
Die Retina besteht aus kleinen konischen und zylindrischen lichtempfindlichen Zellen, welche Photorezeptoren genannt werden. Sie werden unterschieden in Rods/Stäbchen (Hell-Dunkel bzw. Nachtsehen, scotopic/rod vision) und Cones/Zäpfchen (drei Verschiedene für die Primärfarben Rot-Grün-Blau, Tagsehen, photopic/cone vision).

Brightness and Contrast
Die Helligkeit bei verschiedenen Displays ist niemals gleich – verschiedene Stellen werden gemessen und dann die Werte kombiniert, um eine möglichst realistische Darstellung zu erlangen.
ANSI Helligkeit: 9 Stellen werden gemessen und der Durchschnitt verwendet
ANSI Kontrast: 8 weiße und 8 schwarze Stellen werden gemessen und Durchschnitt verwendet.
Alternativ dazu gibt es „Full-on/Full-off“ Kontrasteinstellungen. Die einfachste ist $L_v weiss / L_v schwarz$, diese beachtet aber keine Umgebungshelligkeiten (La). Besser ist $(L_v weiss + La)/(L_v schwarz + La)$. Annäherung an den wahrgenommenen Kontrast: Weber Kontrast (max-min)/min für Niedrigfrequenzbilder und Michelson contrast (max-min)/(max+min) für Hochfrequenzbilder.

Luminance and Contrast Perception



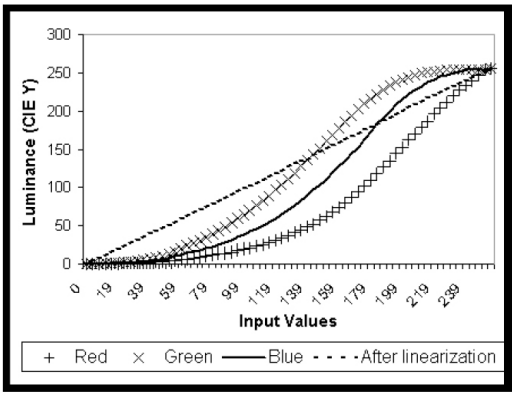
Wie wir wissen ist die räumliche Auflösung durch die Auflösung der Photorezeptoren der Retina beschränkt. Eine weitere Einschränkung ist die Kontrastwahrnehmung (ist dieser zwischen Bildern/Bildteilen zu gering, so können diese Bilder/Bildteile nicht unterschieden werden). Kontrastwahrnehmung hängt von der angepassten Leuchtdichte und der räumlichen Bildfrequenz ab. Genau sowenig werden Intensitäten linear wahrgenommen (eher logarithmisch). Der einzige Schritt, der linear wahrgenommen wird ist der „just-noticable-difference“ (JND) Schritt.
Menschen im Alter von ca. 20 Jahren können Kontrastunterschiede am besten wahrnehmen.
Allgemein gesehen können Menschen einen Kontrastunterschied von 1:4000 wahrnehmen. Bildschirme bringen heute erst einen Unterschied von 1:1500 zustande. Weiters unterscheidet sich der wahrgenommene Kontrast immer vom „echten“ physischen Kontrast (abhängig von der räumlichen Bildfrequenz und Helligkeit)

Tonal Resolution

Mit einem 8bit Display kann man $3*2^8=2^{24}=16.777.216$ Farben darstellen (oft ist die Farbe die gleiche, nur die Helligkeit wird merkbar verändert). Moderne Displays, Grafikhardware und Programme unterstüzen mehr als 8 bit (16bit oder 32bit) – damit werden Farbauflösungen von 2^{48} oder 2^{96} möglich. Dies wird als HDR (high dynamic range) bezeichnet. Photoshop zB erlaubt das Arbeiten mit 8, 16 oder 32bit Bildern, der Bildschirm kann aber nur 8bit Farbtiefe darstellen. Die Umwandlung eines Bildes mit hoher Bittiefe in niedrigere Bittiefe wird „tone-mapping“ genannt (umgekehrt „inverse tone-mapping“)

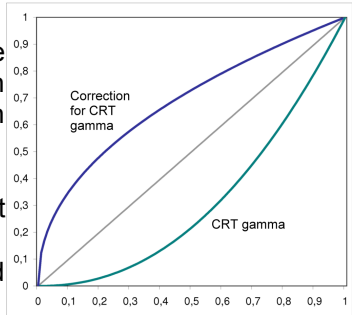
Transfer Function

Der Vorgang, wie ein Display die RGB Werte im Framebuffer aufs Display in sichtbare Farben und Intensitäten darstellt, wird „Transfer Function“ genannt. So wenig wie die menschliche Wahrnehmung ist auch die Transfer Function eines Displays nicht linear. Nachdem der RGB Farbraum geräteabhängig dargestellt wird, ist es wichtig, die Farb- und Helligkeitswerte in geräteunabhängige Werte umzuwandeln. Beispielsweise erscheint die RGB Farbe [18, 101, 32] unterschiedlich auf verschiedenen Displays, während CIE xy = [0.5, 0.3] bei einer Helligkeit von 100 cd/m² auf allen Displays gleich aussieht. Manchmal sind die Transfer Functions direkt in der Hardware implementiert. Moderne Grafikkarten erlauben, die Funktionen manuell zu kalibrieren und Farbprofile auf die Hardware zu laden. Die Invertierung der Transfer Function wird Linearisierung genannt.



Transfer Function für RGB

Display Gamma



Korrigiert Nichtlinearitäten in Displays. Farbkalibrierung für Displays ist eine wichtige und schwierige Aufgabe. Wie bereits erwähnt ist die Reaktion von Displays nicht linear. Man kann diese Reaktion mithilfe der Gammafunktion aber annähern.
Wie kann man schnell die Gammawerte eines Displays ermitteln?

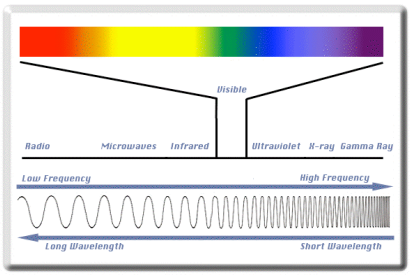
- entweder mit Display Checkerboard und Distanzblick, verglichen mit einem angezeigten Grau oder
- finde einen Grauwert, der zu einem d mit 50% Intensität führt und berechne gamma mit a=0.5

Wenn wir gamma kennen, können wir $a^{1/\gamma}$ anstatt a für die Gammakorrektur verwenden.

$d=m*a^{\gamma}$ mit d (Displayintensität), m (maximale Intensität) und a (erwünschte Intensität) $\gamma=\frac{\ln(0.5)}{\ln(a)}$

Radiometrie und Photometrie

Radiometrie bezeichnet die Wissenschaft der Lichtmessung (Strahlung im elektromagnetischen Spektrum) – Sichtbares Spektrum ist zwischen 380nm und 780nm.
Photometrie ist die Wissenschaft der Lichtmessung mit Rücksicht auf die menschliche Wahrnehmung.
Das menschliche Auge ist nicht für alle Wellenlängen und Intensitäten gleich empfindlich.



Die CIE photopic luminous efficiency curve V(λ) beschreibt, wie der Mensch Licht wahrnimmt. Alle radiometrischen Konditionen haben photometrische Gegenstücke.
Radiometrische Werte können mittels spektraler Gewichtung von V(λ) in photometrische Werte umgewandelt werden

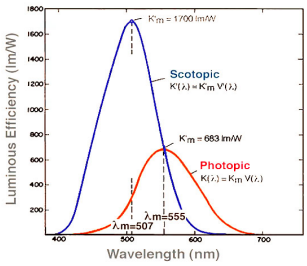


Figure 9. The scotopic and the photopic curves of spectral luminous efficacy (non-normalised values).

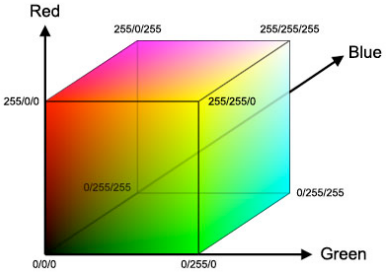
Colorimetry

Die Farbmetrik ist die Wissenschaft der Wahrnehmung, Messung und Repräsentation von Farben. Der Mensch kann zwischen 10 Millionen verschiedenen Farben unterscheiden (mit einer Farbvariation von ca. 2nm und Helligkeitsvariation von 2%). Wie wir bereits wissen, hat die Retina drei verschiedene Arten von Zäpfchen (Cones), welche die Primärfarben Rot, Grün und Blau unterscheiden können. Dies ist der Grund, warum die RGB-Farben für die Darstellung bei grafischen Displays ausreichen.

Color in Graphics

Mit 8bit Farbtiefe sind 16.777.216 verschiedene Farben darstellbar – damit lässt sich der so genannte RGB-Farbwürfel aufspannen. Probleme mit diesem Farbmodell sind:

- wir wissen nicht, wie der Bildschirm die Farben darstellt (abhängig von der Transfer Function ist der Farbraum nicht linear)
- Menschen nehmen Farben und Intensitäten nicht linear auf.

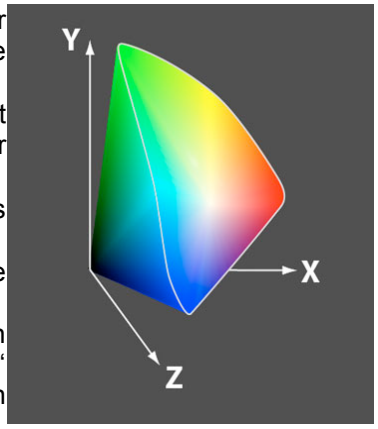


Damit ist das RGB Farbmodell geräteabhängig. Wir möchten aber, dass Bilder auf verschiedenen Displays gleich aussehen, des weiteren wollen wir Berechnungen im Farbraum vornehmen können, welche auf der menschlichen Wahrnehmung basieren. Hierfür benötigen wir Farbmodelle, welche geräteunabhängig sind und auf der menschlichen Wahrnehmung basieren.

CIE XYZ

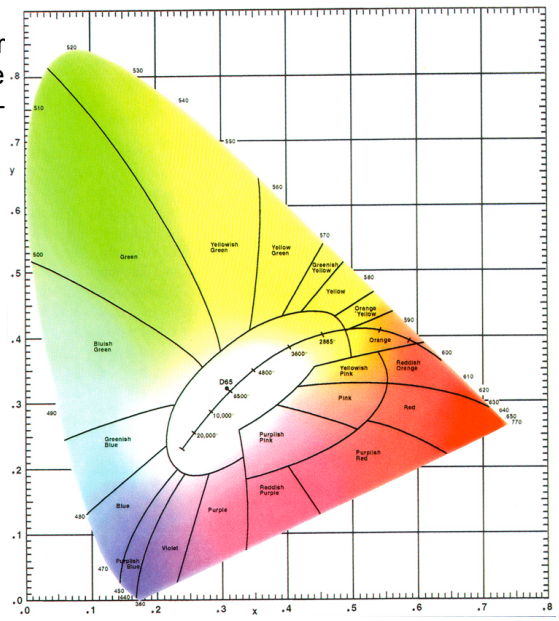
Die Frage lautet, welche Kombination der Primärfarben gleicht der wahrgenommenen Farbe? Die CIE hat hierzu ein paar Farbexperimente gemacht:

- drei monochromatische (primärgefärbte) Lichter wurden mit Überdeckung auf einen weißen Hintergrund projiziert und mit einer Testfarbe verglichen
- Benutzer mussten die Intensitäten der Lichter so lange ändern, bis das Ergebnis der Testfarbe glich
- Die resultierenden Intensitäten können nun benutzt werden, um die Primärfarben in Spektralfarbeinträge umzuwandeln.



Verschiedene Spektralfarbeinträge können zu den selben Primärfarben führen (sehen für den Menschen identisch aus) – diese werden „Metamers“ genannt. Die Primärfarben RGB können nicht direkt alle sichtbaren Farben darstellen – es existieren zB negative Werte mit verschiedenen Wellenlängen. Deswegen hat die CIE drei hypothetische (nicht monochromatische) Lichtquellen vorgeschlagen, um diesem Problem entgegenzuwirken ($\bar{x}\,\bar{y}\,\bar{z}$). Das Produkt einer gegebenen Lichtquellenstärke und Oberflächenreflektion definiert die Farbfunktion C(λ). Multipliziert man diese Farbfunktion zur Farbübereinstimmungskurve und integriert man über das resultierende Spektrum, so erhält man den CIE XYZ Farbraum.

CIE xy Chromaticity Diagram



Da das Arbeiten in einem dreidimensionalen Farbraum sehr umständlich für viele Algorithmen ist, wird normalerweise die X+Y+Z = 1 Ebene verwendet. (Z gibt keine Informationen – kann also ignoriert werden).
Beschreibung des „Horseshoe-Diagrams“:

- äußere Kanten: sichtbare Farben
- gerade Linie: Mischfarben (keine Komplementärfarben vorhanden)
- „Gebiet/Fläche“: Mischung aus sichtbaren Wellenlängen
- Weißer Punkt im Zentrum
- Wenn man sich vom weißen Zentrum nach außen bewegt, bedeutet dies eine Zunahme der Farbsättigung
- Winkeländerung einer Linie, die durch das Zentrum geht bedeutet eine Änderung des Farbtons
- Gerade Linie zwischen zwei Punkten zeigt alle möglichen Mischungen dieser zwei Farben (bei mehr als 2 Punkten: „Gebiet“, Skala eines Displays mit n Primärfarben)

Color Gamut

Farben werden normalerweise durch Filterung von weißem Licht erzeugt. Diese Filter sind nicht perfekt. Wenn ein Display RGB als Primärfarben nutzt, dann kann die wahrgenommene Farbe auf da CIE xy Modell umgewandelt und im „Horseshoe“-Diagramm gezeigt werden. Verbindet man die primären Einträge mit Linien so wird ein Farbraum definiert, der vom Display angezeigt und vom Menschen wahrgenommen werden kann. Dies wird als color gamut (Farbskala) bezeichnet. Beachte, dass die Einträge in der Skala diskretisiert sind, dh nicht alle Farben können angezeigt werden.
Je mehr Primärfarben und je weiter diese am Rand des Diagrammes sind, desto größer ist der Farbraum, der vom Display generiert werden kann. (Laserprojektoren zB unterstützen mehr als drei Laserlichtquellen um eine große Skala zu gewährleisten)

Temporal Vision

Die Frequenz, die nötig ist, damit ein Signal als flickerfrei wahrgenommen wird heißt „Flicker Fusion Threshold/Rate“. Sie hängt von der Helligkeit (bzw. Kontrast) des Signals (heller und höher) und von der Position des Signals in der Retina ab. (Auge ist im peripheren Bereich empfindlicher). 60Hz sind für den peripheren Bereich und nur 4Hz für den fovealen Bereich nötig um flickerfrei zu sehen. Die Framerate eines Displays sollte nicht unter dieser Grenze liegen.

Graphics, Displays und Perception

Zusammenfassend korrelieren folgende Displayfaktoren mit der physiologischen Wahrnehmung:

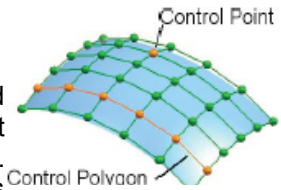
- räumliche Auflösung (Anzahl Pixel, Auflösung des menschlichen Auges, Abstand zum Display)
- Kontrast und Helligkeit sowie Kontrastwahrnehmung (frequenzabhängig)
- Wiederholungsrate und Flicker-Fusion-Rate (>60Hz, Framerate kann geringer sein)
- Farbraumabdeckung und Farbauflösung sowie Helligkeit und Farbwahrnehmung

Aber: kognitive Verarbeitung spielt auch eine wichtige Rolle (zB Salienz)

Beispielfragen zum Kapitel:

- Was bedeutet temporale Wahrnehmung?
- Unterschied zwischen Radiometrie und Photometrie
- Was ist das Problem mit RGB?
- Wie kann man Kontrast / Tonauflösung bei einem Display erhöhen?

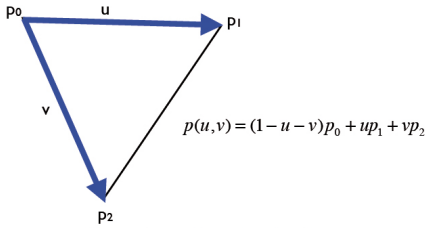
8. Curves and Surfaces



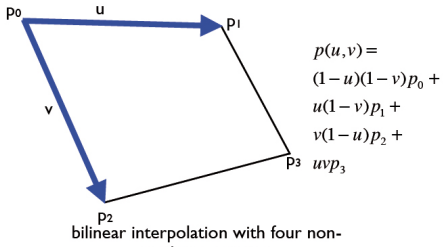
Splines
Komplizierte Oberflächen können nicht einfach mittels Dreiecken modelliert und animiert werden. Stattdessen werden Kontrollpunkte und/oder -polygone definiert und dazwischen liegende Oberflächenteile werden durch Interpolation berechnet. Dies gilt für Kurven und Oberflächen. Solche interpolierten Kurven werden Splines genannt (und Oberflächen werden Spline-Oberfläche genannt.

Funktionsweise und Berechnungen:

Bilineare Patches (Füllstücke)
Bilineare Interpolation in Dreiecken war für viele Komponenten der Schattierung und Rasterisierung notwendig. Bilineare Interpolation kann um mehrere Ecken (Kontrollpunkte) erweitert werden (zu Patches, Quadrate). Die 4 Ecken können nicht koplanar sein, d.h. Ihre 3D Koordinaten sind nicht auf der gleichen Ebene.
Interpolation von 3D-Punkten über die Parameter u,v erlaubt das Berechnen von dazwischen liegenden 3D Punkten auf Patches (glatt). Dies begrenzt aber das Potenzial zur Beschreibung von gesamten Oberflächen, da mehrere Patches zusammengeführt werden müssen – was zu Unterbrechungen an den Kanten führt.

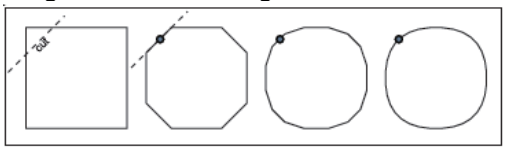


bilinear interpolation in triangle



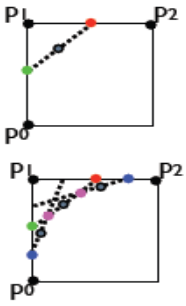
bilinear interpolation with four non-coplanar vertices

Quadratische Bézierkurven
Beginnend bei einem Rechteck (4 Kontrollpunkte) werden rekursiv Ecken weggeschnitten, sodass am Ende eine glatte Kurve (auch begrenzte Kurve) entsteht. Interessant ist, dass jeder Mittelpunkte einer neuen Kante bereits auf der begrenzten Kurve liegt:



subsequent corner cutting results in limited curve

subdivide (p_0,p_1,p_m)
$p_{01}=(p_0+p_1)/2$
$p_{12}=(p_1+p_2)/2$
$p_m=(p_{01}+p_{12})/2$
subdivide(p_0,p_{01},p_m)
subdivide(p_m,p_{12},p_2)



in general for t=0..1:

$$p(t) = (1-t)((1-t)p_0 + tp_1) + t((1-t)p_1 + tp_2) = \boxed{(1-t)^2}p_0 + \boxed{2(1-t)}p_1 + \boxed{t^2}p_2$$

blending of control points:

$$p(t) = \sum_{i=0}^{N-1} w_i(t)p_i$$

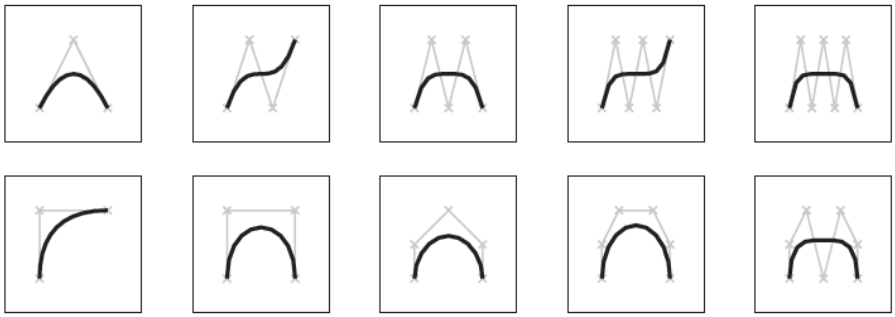
blending functions for different control points:

$$w_0(t) = (1-t)^2$$
$$w_1(t) = 2(1-t)t$$
$$w_2(t) = t^2$$

Note, that for each parameter t, the sum of all w_i equals 1, with maximum influence at t=0,1/2,1!

Um den Mittelpunkt zu parametrisieren, wird ein quadratischer Ausdruck mit 3 Kontrollpunkten aufgestellt. Die Kurve, die von allen Mittelpunkten gebildet wird, wird daher quadratische Bézierkurve genannt.
Die Funktion p(t) zur Berechnung der Mittelpunkte, ist der gewichtete Durchschnitt der Kontrollpunkte p_i (sie sind mit Blending-Funktionen gemischt).

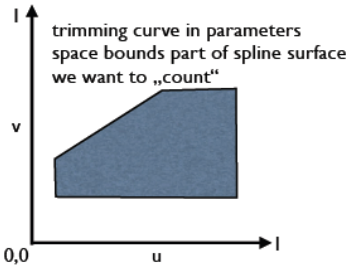
Bézierkurven höherer Ordnung
Die Erweiterung auf 4 Kontrollpunkte erfolgt nach dem gleichen Verfahren. Man spricht von kubischen Bézierkurven. Die Unterteilungsidee kann zu einer beliebigen Anzahl von N Kontrollpunkten verallgemeinert werden. Der Grad der entsprechenden Blending-Funktionen ist N-1. Die Koeffizienten haben alle das gleiche Muster von Binomialkoeffizienten mal $(1-t)^{N-i-1}t^i$.



Die Rekursion der Blending-Funktionen ist leicht zu implementieren. Bézierkurven sind eine spezielle Form von Splines. Der Grad der Bézierkurve ist eins weniger der Anzahl von Kontrollpunkten. Alle Kontrollpunkte beeinflussen alle Teile der Kurve, außer die Endpunkte. Die Menge von Liniensegmenten, welche Kontrollpunkte verbinden, wird Steuerpolygon genannt. Der Anfang und das Ende der Kurve sind tangential zum kontrollieren des Polygons. Sind Kontrollpunkte 3D, erhalten wir eine Raumkurve.

Bézieroberflächen

Die Erweiterung auf Flächen geht wieder nach dem gleichen Verfahren. Wir haben also 3D-Kontrollpunkte in 2 Dimension über einer Oberfläche. Die Blending-Funktionen werden 2D (u,v-Parameter) und die 2D-Blending-Funktionen können durch Multiplikation zweier 1D-Blending-Funktionen hergeleitet werden (eine in u, eine in v).



Solche Bézierflächen funktionieren gut bei rechteckigen Oberflächen, wir können das gleiche Prinzip auch auf Oberflächen, die nicht rechteckig sind, anwenden. Hierzu wird der Parameterraum u,v zu einer bestimmten Oberflächenbasis unter Verwendung eines Pseudopolygons u,v eingeschränkt. Parameter außerhalb dieses Polygons werden nicht berücksichtigt und beeinflussen die Oberfläche nicht.

Einheitliche kubische B-Splines

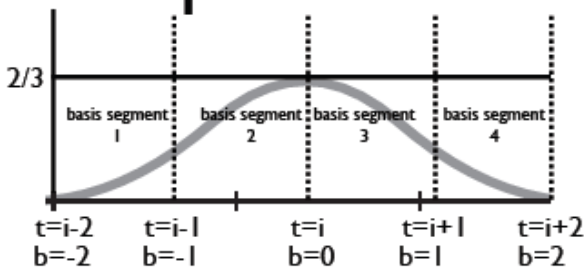
Unbedingt hier nachlesen http://de.wikipedia.org/wiki/Spline

Ein Problem der Bézierkurven ist, dass, wenn sie viele Kontrollpunkte haben, der Grad des Polynoms sehr hoch wird. Das führt zu rechnerischen Problemen. Ein weiteres Problem ist, dass jeder Kontrollpunkt die Kurve beeinflusst, was unnötig ist, wenn lokale Sektionen geändert werden müssen.

Können Polynome mit niedrigem Grad erstellt werden, die Blending-Funktionen nutzen und welche auf eine kleine lokale Region eingeschränkt sind? Wir können Blending-Funktionen aus Stücken von Polynomen erstellen, d.h. eine B-Spline-Blending-Funktion – die daraus resultierende Kurve wird B-Spline genannt. Diese Stücke (die sogenannte Basisfunktion) sind entlang eines Parameterraums t=4..N-1 (statt t = 0..1), sodass der maximale Einfluss jeder Basisfunktion mit dem entsprechenden Kontrollpunkt übereinstimmt.

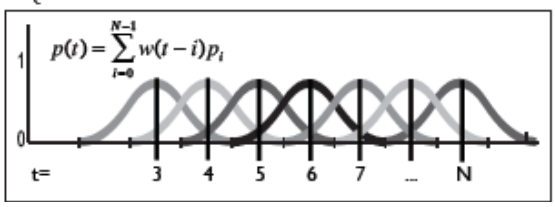
Uniform Cubic B-Splines

- In this example, the same basis function is shifted in such a way that its maximum is at t=i
- The basis function is stitched together from four basis segments
- Each basis segment has to be evaluated in the local rage of 0..1 (thus we have to normalize t with respect to the corresponding segment)
- The sum of all blending functions at every t is 1, again
- On a global basis, t's range is 4..N-1 (parameters between 1..3, and N..N+2 are not considered, since there are not enough points for the blending functions to sum to 1 in those regions)



$$w(b) = \begin{cases} \frac{1}{6}t^3, t = b+2 & -2 < b \le -1 \\ \frac{1}{6}\left(-3t^3 + 3t^2 + 3t + 1\right), t = b+1 & -1 < b \le 0 \\ \frac{1}{6}\left(3t^3 - 6t^2 + 4\right), t = b & 0 < b \le 1 \\ \frac{1}{6}\left(-t^3 + 3t^2 - 3t + 1\right), t = b-1 & 1 < b \le 2 \\ 0 & otherwise \end{cases}$$

basis segment | normalizing to 0..1 | local range

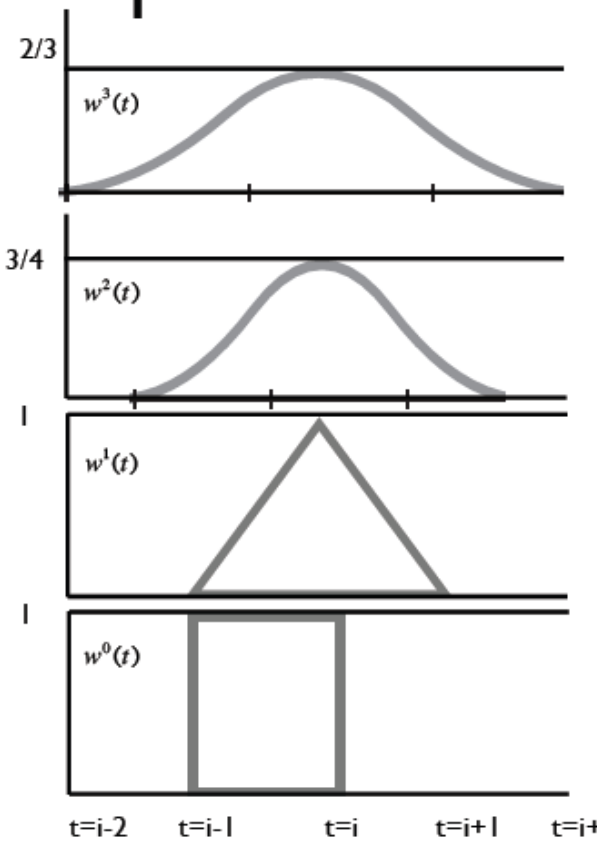


Non-Cubic B-Splines

- One control point of a cubic B-spline influences a local region of t=i-2...t=i+2 on the curve
- This region can be made smaller or larger by using lower-order or higher order blending functions
- Cubic B-spline blending functions are just one of a family of blending functions
- As for Bézier blending functions, the B-spline blending functions can be computed recursively (after mapping t to local range!):

$$w^k(t) = \frac{t}{k-1} w^{k-1}(t) + \frac{k-t}{k-1} w^{k-1}(t+1)$$

$$w^0(t) = \begin{cases} 1 & 0 < t \le 1 \\ 0 & otherwise \end{cases}$$



NURBS
In der Praxis wird eine kompliziertere Form der B-Splines verwendet, anstatt einheitlicher kubischer B-Splines. Diese werden als non-uniform rational B-Splines (NURBS) bezeichnet. Das Mischen von NURBS-Funktionen ist das Verhältnis zweier Polynome (per Definition sind das rationale Funktionen). Der Vorteil der (rationalen) NURBS im Vergleich zu (Polynom-) B-Splines ist, dass sie genau Kegel darstellen. Daher werden sie bei der Modellierung mit Zylindern und Kugeln bevorzugt.
Ungleichmäßig bedeutet, dass die Lücken in t zwischen den Biege-Funktionen weder einheitlich noch eine ganze Zahl sein müssen (benutzerdefiniert).

Beispielfragen zum Kapitel:

- Nachteile und Einschränkungen von Bezier?
- Beschreiben Sie (im Detail) die Funktionsweise von Bezier-Kurven und Flächen.
- Erklären Sie (in aller Kürze) den Unterschied, Vor- und Nachteile zwischen Bezier und B-Spline Kurven/Flächen.
- Beschreiben Sie (in aller Kürze) wie mehrere Bezier Kurven/Flächen aneinandergesetzt werden können. Welche Bedingungen müssen erfüllt sein, um eine korrekte Interpolation zwischen den Teilstücken sicherzustellen?

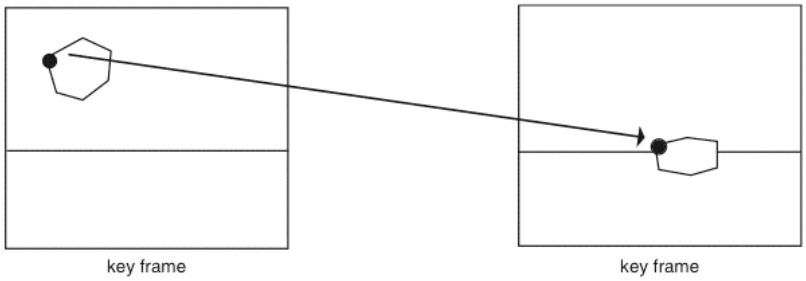
9. Basics of Animation

Motive für Computeranimation:

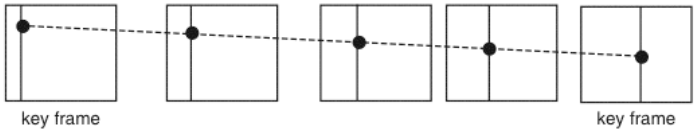
- physikalisch-basierte Animation (vorwärts / inverse Kinematik, Teilchensysteme, „Rigid Body“ Animation, etc.)
- Bewegungserfassung von realen Dingen wie Menschen (Gesicht, Körper, Bewegungen, etc)
- Animation von Flüssigkeiten und Gasen
- Modellieren und Animieren menschlicher Figuren (greifen, gehen, anziehen, etc.)
- Gesichtsanimation (Muskelmodelle, Haut, Lippensynchronisation, etc.)

Das grundlegendste Thema zur Computeranimation ist **Interpolation-basierte Animation:**

Die Grundidee ist besondere Zustände („Key frames“) vorzudefinieren und diese mittels Interpolation als flüssige Animation darzustellen. Die „Keyframe“ Animation errechnet automatisch mittels Interpolation die Zwischenrahmen. Die Parameter, welche interpoliert werden, können willkürlich gewählt werden (Knoten, Normalen oder andere tangentiale Informationen).
Die Parameter, wie Ecken und Tangenten bei Segmenten, werden auf Schlüsselstellen definiert, diese werden dazwischen interpoliert, siehe Abbildung.
Die Art des Parameters beeinflusst generell die Interpolationsmethode nicht.



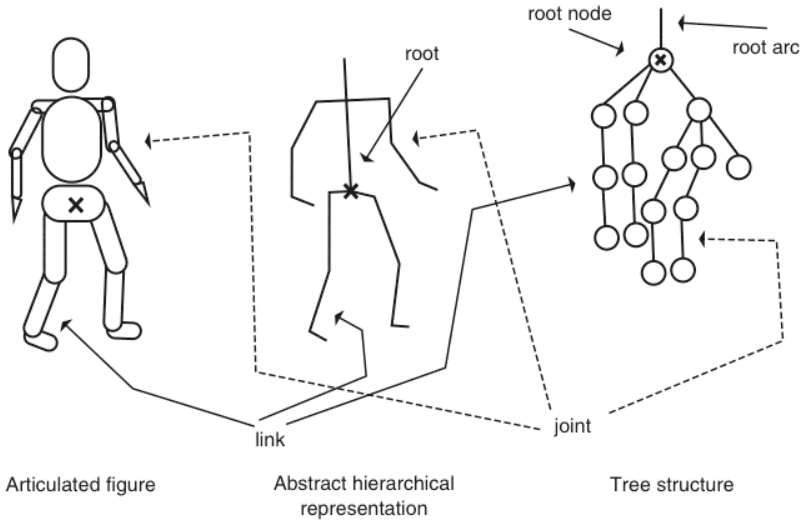
Simple key frames in which each curve of a frame has the same number of points as its counterpart in the other frame



Keys and three intermediate frames with linear interpolation of a single point (with reference showing the progression of the interpolation in *x* and *y*)

Die einfachste Möglichkeit ist die **lineare Interpolation**, jedoch wird damit nicht die erwünschte Animation erreicht:
Ist die Oberfläche mit splines definiert (Bézier, B-Spline, NURBS), dann können die Keyframes durch Modifizieren der Kontrollpunkte definiert werden. Diese werden interpoliert, während die tatsächlichen Oberflächenpunkte mittels den entsprechenden Funktionen (Bézier, B-Spline, NURBS – Interpolation) berechnet werden.
Jeder Keyframe muss die gleiche Anzahl von Parametern haben → bijektive Korrespondenz. Wird das bei der Modellierung nicht berücksichtigt, dann können Algorithmen eingreifen um dies in den Keyframes sicherzustellen.

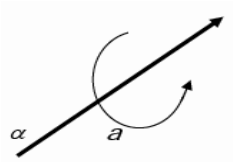
Rigid-Body Animation von Hierarchien



Starre Körper, welche hierarchisch aufgebaut sind, können mittels Keyframes und Interpolation animiert werden. Es muss darauf geachtet werden, dass die Komponenten von willkürlichen Transformationsmatrizen nicht interpoliert werden können: Verschiebungen und Skalierungen können interpoliert werden, Rotationsmatrizen nicht → für Rotationen können andere Repräsentationsformen benutzt werden, wie Euler-Winkel für jede Hauptachse oder Quaternionen (sphärische lineare Interpolation erforderlich).

Quaternionen in Kürze
Die Quaternionen sind ein Zahlensystem ähnlich den komplexen Zahlen und eine Erweiterung der reellen Zahlen. Sie bieten eine Möglichkeit zur geeigneten Darstellung von Rotationen im Raum.
Diese werden durch einen Vektor a (imaginärer Teil) und einem Winkel α = (b, c, d) (realer Teil) definiert:
<u>q = a + α</u>
Dabei ist q eine Einheitsquaternion, wenn $|q|=a^2+b^2+c^2+d^2=1$. Die Einheitsquaternion kann in eine Rotationsmatrix umgewandelt werden.
Zu beachten ist, dass a, b, c, d nicht voneinander unabhängig sind - somit können wir nicht einfach interpolieren. Sie müssen sphärisch interpoliert werden (kugelförmig lineare Interpolation), um sicherzustellen, dass sie normal bleiben.

$$R=\begin{pmatrix} a^2+b^2-c^2-d^2 & 2(bc-ad) & 2(bd+ac) \\ 2(bc+ad) & a^2-b^2+c^2-d^2 & 2(cd-ab) \\ 2(bd-ac) & 2(cd+ab) & a^2-b^2-c^2+d^2 \end{pmatrix}$$

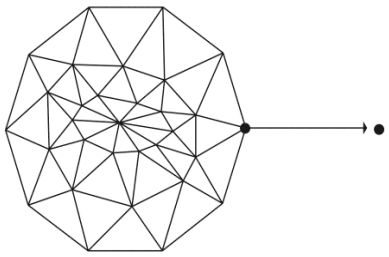


Weitere Regeln:
- Wenn q_1 und q_2 Einheitsquaternionen sind, gilt $q_1 * q_2 = R_1 * R_2$
- Das Konjugat von q = a + α ist $\bar{q}$= a – α
- Bei gegebenen Vektor v, gilt $Rv=qv\bar{q}$ und $q\bar{q}=\bar{q}q = |q|^2 = 1$

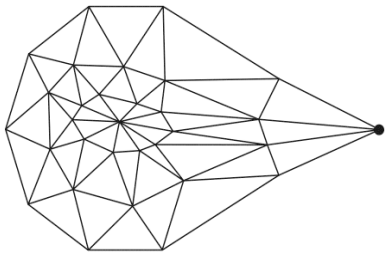
Vorwärts Kinematik
Durch das Traversieren des Szenegraphen in depth-first Reihenfolge von der Wurzel bis zum Blatt entspricht der Vorwärts-Kinematik, d.h. Schritt für Schritt wird jedes Gelenk in Position gebracht, indem der Baum von oben nach unten (top-down) in Reihenfolge betrachtet wird. Das Blatt-Objekt wird also in Position gebracht und wird somit Endeffektor genannt.

Inverse Kinematik
Manchmal ist es einfacher, die Endlagen der Endeffektoren zu definieren und die kinematisch möglichen Transformation durch Traversieren des Baumes in Bottom-Up Reihenfolge zu berechnen. Für die inverse Kinematik müssen (manchmal recht große) Gleichungen numerisch gelöst werden.

Verformbare, nicht starre Objekte können nicht mit Kinematik animiert werden.
Ein Vorteil der Keyframe-Animation ist, dass Zwischen-Frames nicht mehr manuell festgelegt werden müssen. Es wäre auch zu viel des Guten, jeden individuellen Parameter (jeden von tausenden Knoten) manuell zu modifizieren innerhalb jedes Keyframes.
Versucht man den Animationsprozess mehr zu automatisieren, senkt man damit die Objektmodellierung wesentlich (d.h. Verformen der Objekte jedes Keyframes).

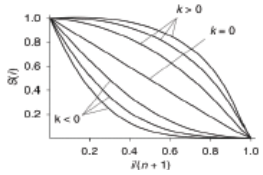

Displacement of seed vertex

Eine Möglichkeit ist die Ecken direkt zu verändern, während sich das Netz von verbundenen Knoten sinnvoll mit verändert bzw. die anderen Knoten folgen.
Wenn ein (Start-) Knoten (seed vertex) bewegt wird und sich die verbundenen Ecken folgen basierend auf der Distanz des Startknotens, nennt man das „Picking and Pulling“ (auswählen und ziehen).

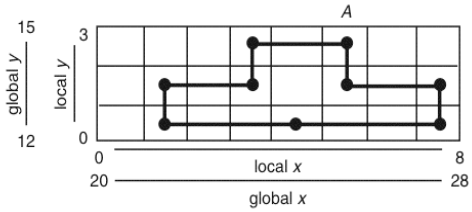

Attenuated displacement propagated to adjacent vertices

Picking and Pulling
Für jede Ecke wird ein Dämpfungsfaktor S berechnet, der beschreibt, um welches Maß die Ecke den Bewegungen des Startknotens folgen muss.
Der Abstand zwischen Ecke und Startecke kann also proportional zu der Anzahl der Kanten i auf dem minimalen Weg von der Ecke zur Startecke gesehen werden (nicht genau, aber als Angleichung → besser: Kantengewichte).
Der Kantenabstand i ist mit dem maximalen Abstand n normalisiert, welcher vermehrt einen Einfluss auf die Bewegung des Startknotens haben soll (benutzerdefiniert).

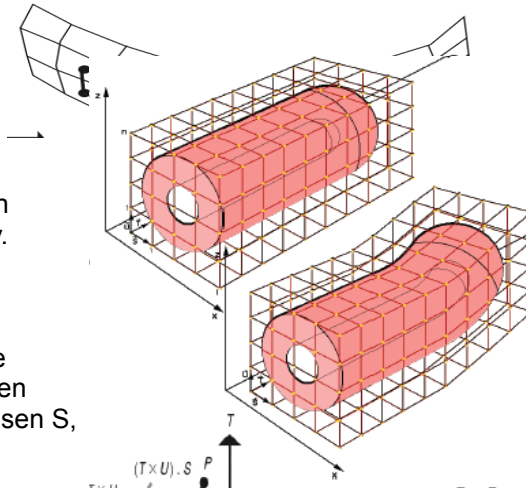
$$S(i) = 1.0 - \left(\frac{i}{n+1}\right)^{k+1} \qquad k \geq 0$$
$$= \left(1.0 - \frac{i}{n+1}\right)^{-k+1} \qquad k < 0$$



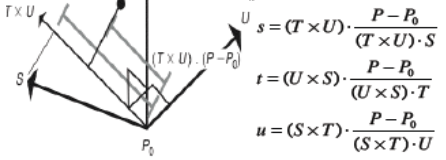
Der Dämpfungsfaktor kann exponentiell skaliert werden mit k:
- k = 0: entspricht einer linearen Dämpfung
- k < 0: schafft einen elastischen Eindruck
- k > 0: schafft starre Effekte



Free-Form Deformation
Verformen der Ecken durch Bewegen der Objektecken ist nicht der effizienteste Weg. Die Free-Form Deformation ist sehr häufig. Eine andere Möglichkeit ist ein lokales Koordinatensystem, das die Oberfläche des Objekts umschließt, welches in eine Gitterstruktur unterteilt wird, zu verformen.
Anstatt der Objektecken werden nun die Gitterecken verschoben, die neuen Positionen der eingeschlossenen Objektecken sind in jeder Zelle berechnet.
Zuerst also die Berechnung der lokalen Koordinaten u,v von Objektecken innerhalb der unverformten Zellen vornehmen, dann verformen der Zellen und berechnen der neuen Objektecken u,v. Dies ist effizienter, wenn viele Objektecken existieren, denn das Koordinatengitter kann eine viel niedrigere Auflösung haben.



In 3D ist es im Grunde gleich; die Gitterzellen werden 3D und die Interpolation der Gitterecken wird kubisch. Die lokalen Koordinaten werden 3D (genannt trilineare Interpolanten): s, t, u. Die Netzachsen S, T, U müssen nicht notwendigerweise orthogonal sein.
Der Nenner normalisiert die lokalen Koordinaten, siehe Formeln.



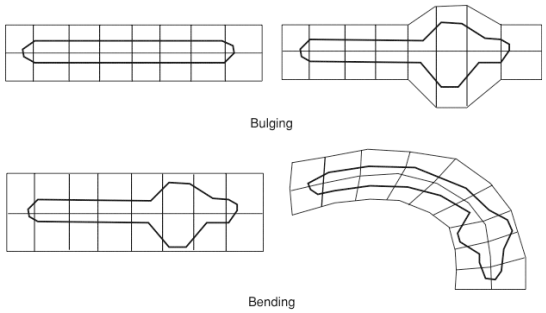
$$s = (T \times U) \cdot \frac{P - P_0}{(T \times U) \cdot S}$$
$$t = (U \times S) \cdot \frac{P - P_0}{(U \times S) \cdot T}$$
$$u = (S \times T) \cdot \frac{P - P_0}{(S \times T) \cdot U}$$

Hat man die lokalen Koordinaten eines Objekts, dann können wir mit einfachen linearen Kombinationen die Originalpositionen von P rekonstruieren.
Kann man die Kontrollpunkte berechnen, so kann man auch mit Bézier-Interpolation die Objektecken innerhalb der Zellen berechnen.

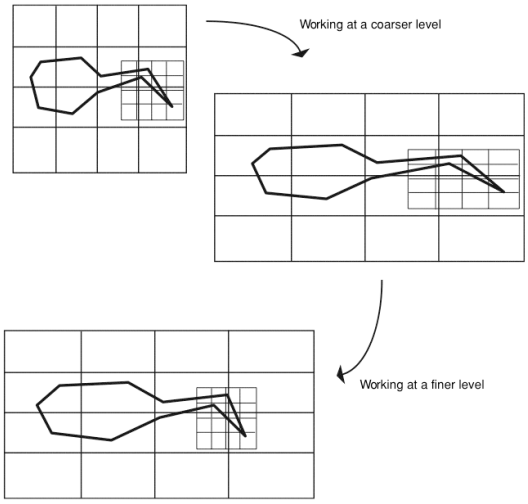
Kombination von FFD
Ein Objekt kann in mehrere FFDs unterteilt werden, welche aber zusammengeführt werden können. Das ermöglicht unterschiedliche Animation von Objektteilen – Kontinuität an den Grenzen muss aber gewährleistet sein.
Wie Bézier – Kurven und -oberflächen, können auch Bézier – Festkörper unter Kontinuitätsbedingungen über Grenzen zusammengefügt werden. Um positionelle Kontinuität sicherzustellen, müssen benachbarte Gitter entlang der Grenze die gleichen Kontrollpunkte aufweisen: Sicherstellung durch co-Linearität (Punkt auf Linie) zwischen benachbarter Kontrollpunkte über die gemeinsame Grenze

Vermischung von FFD
FFD kann sequentiell oder hierarchisch zusammengesetzt sein.
Sequentiell: Fortschritte durch eine Sequenz von FFDs (erlaubt die Kombination verschiedener Modellierungsschritte)
Hierarchisch: FFDs mit feiner Skala sind in FFDs mit gröberer Skala eingebettet (erlaubt das Arbeiten mit unterschiedlichen Detaillierungsgraden), Objekt- und Kontrollpunkte von feinen FFDs können mit gröberen FFDs animiert werden.

Sequentiell



Hierarchisch



FFD Tools
Individuelle FFDs können modelliert und animiert werden durch Objektgeometrie als (wiederverwendbares) Werkzeug für die Erstellung spezieller Animationseffekte.
Anstelle der manuellen Festlegung der FFD-Kontrollpunkte können sie mit physisch basierter Simulation und weiteren Funktionen berechnet werden.

Beispielfragen zum Kapitel:
- Wie wird Keyframe-Interpolation auf Splines angewendet?
- Unterschied zwischen Vorwärtskinematik und inverser Kinematik?
- Wie werden Free-Form-Kombinationen kombiniert? Wie kann der Übergang bei Flächen flüssig gestaltet werden?