Formeln

**Projektion**: Punkt (x,y,z) wird auf Punkt p in der xy Ebene (z=0) mit Kamera in Punkt e projiziert.

* *Perspektivische 3D-Projektion*

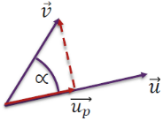
Bek: Vektoren e, (Kamera) und P, Ges: c

*Beispiel e= (0,0,-2), P= (-1,2,5)* 🡺 *c = (-2/7, 4/7)*

* *Orthogonale Projektion*

*Fläche z=0 🡺 P =(x,y) z.B. P=(1,4,-3) 🡺 c = (1,4)*  
Fläche F: 𝑥+𝑦+𝑧=0

* + n = (1, 1, 1) aus der Ebenengleichung abgeleitet
  + zur Ebene orthogonale Linie durch Eckpunkt:
  + Schnittpunkt S der Gerade mit der Ebene: Gerade in Ebenengleichung einsetzen und nach t lösen Für jeden Punkt lösen und t in der Geradengleichung einsetzen, um S zu bekommen

*Beispiel: (-1, 2, 1): (-1+t) + (2+t) + (1+t) = 0 🡺 t = -2/3 🡺 S = (-5/3, 4/3, 1/3)*

* *Vektor Projektion*:

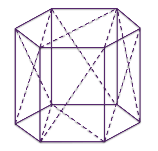
**Kreuzprodukt**: Vektorprodukt ist Verknüpfung zweier Vektoren, dessen Ergebnis wieder ein Vektor ist, der senkrecht auf beiden Vektoren steht (Paralellogramm)

**Skalarprodukt**: Zwei Vektoren stehen senkrecht aufeinander, wenn resp. ( ist.

* Positiv, , nur wenn oder = 0
* Symmetrisch
* Anwendung:
  + Länge: 🡪Distanz: 🡪 Normalisierung: 🡺Winkel:

Länge von Vektoren: )

Schnittpunkt Ebene, Gerade allgemein (Keine Lösung wenn Skalarpodukt Gerade mit Normale Ebene = 0)

Modellierung von 3D Objekten

* Triangle Meshes: Polygone aus Dreiecken, garantiert flach, drei beliebige Punke, Einfache Scan-Transformation
* Punkte in Koordinatensystem (berechnet, oder in Liste):
  + *Ohne Indexing*: Pro Punkt ein Wert (evtl. Vektor) in einer Liste von Dreiecke, ein flaches Array
  + *Mit Indexing*: Liste von Eckpunkten, Liste von Dreiecke (drei indices in Eckpunktliste), zwei flache Arrays

Euklydische Geometrie: Dreiecksungleichung

* , ; ,
* ; , (“Translation”)

Koordinatensysteme

Vektorraum aufgespannt mit Basis-Vektoren, Adresse als Koordinaten, Links- und rechtshändiges System.

* Modell-Koordinatensystem (local space)
  + Rechtshändig, Jedes Objekt ein eigenes System
* Welt-Koordinatensystem
  + Rechtshändig, Alle Objekte hier platziert
* View-Koordinatensystem
  + Linkshändig, Sicht aus Kamera
* Bild-Koordinatensystem: Werden in 2 aufgeteilt
  + *Clip Space*: Versteckt weite Objekte mittels z-Buffer, Normalized device coordinates (-1, 1)
  + *Screen Space:* 2D, Umrechnung Pixel-Koordinat

Transformation

Platzierung innerhalb des Raums, Deformieren, etc.

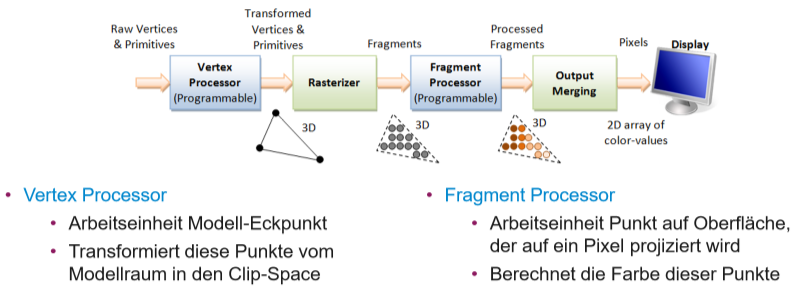
* *Translation*: Verschiebe alle um denselben Vektor, ist NICHT linear. Lösung: Homogene Koordinaten: Erweiterung um eine weiter Dimension 2D 🡺 2DH
* *Skalierung*: Verschieben um einen Faktor, ist linear
* *Rotation*: Alle drehen um gleichen Winkel, ist linear

Braucht in 3D eine Dreh-Achse

* *Gesamt-Transformation*: Zusammenrechnung Einzeltransformationen: Achtung: Reihenfolge spielt teilweise eine Rolle. (Matrixmultiplikation)

GPU-Programmierung

Double Frame Buffering: Update und auslesen nicht synchron: Vermeidet Update-Artefakte



Shader Programm:   
programmiert Vertex- (Modell-Eckpunkte) und Fragment Prozessor (Farbe eines Pixel). Beschreibt Transformation einzelnen Modell-primitive. Hat keinen Zugriff auf andere Primitive, separat vom Host.  
GLSL: in (vorheriger State), out (next), uniform (alle) 🡺*gl\_Position* wichtigster Out Parameter

Beleuchtung

Modell-Normalen dem Shader übergeben als Vertex-Eigenschaft: Umrechnen Modell, in Weltraum (Interpoliert auf Dreiecksfläche)

* *Ambient Lightning:* Diffuses Licht aus allen Richtungen. 
* *Diffuse Belichtung*: Licht von Punktquelle, Remission in alle Richtungen. Für matte Oberflächen.

vec3 normDir= norm(**normal**);  
vec3 lightDir= norm(**lightPos-** fragPos);  
float cosTheta= max(**dot**(**normDir**,lightDir),0.0);  
vec3 diffuse= cosTheta\* lightColor\*objectColor;  
fragColor = vec4(**diffuse**, 1.0);

* *Specular Lighting*: Licht Spiegelung von Punktquelle, Remission in eine Richtung. Für spiegelnde Oberflä.

vec3 normDir = norm(**normal**);  
vec3 camDir= norm(**camPos** - fragPos);  
vec3 lightDir=norm(**lightPos-**fragPos);  
vec3 reflectDir=reflect(**-lightDir**, normDir);  
float cosTheta= max(**dot**(**camDir**, reflectDir),0.0);  
float strength = pow(**cosTheta**,shininess);  
vec3 specular= strength\* lightColor\* objectColor;  
fragColor= vec4(**specular**,1.0);

* *Phong Lighting/Shading*: Zusammensetzung ambient, diffuse und specular (alles addiert \* 1/3)
  + Problem keine Spiegelung, wenn Winkel kamera und Refexion > 90°.
  + Lösung: *Blinn-Phong-Shading*: Winkel zwischen Halfway-Vector und Normalen anstelle Reflektionsvektor und Kamera

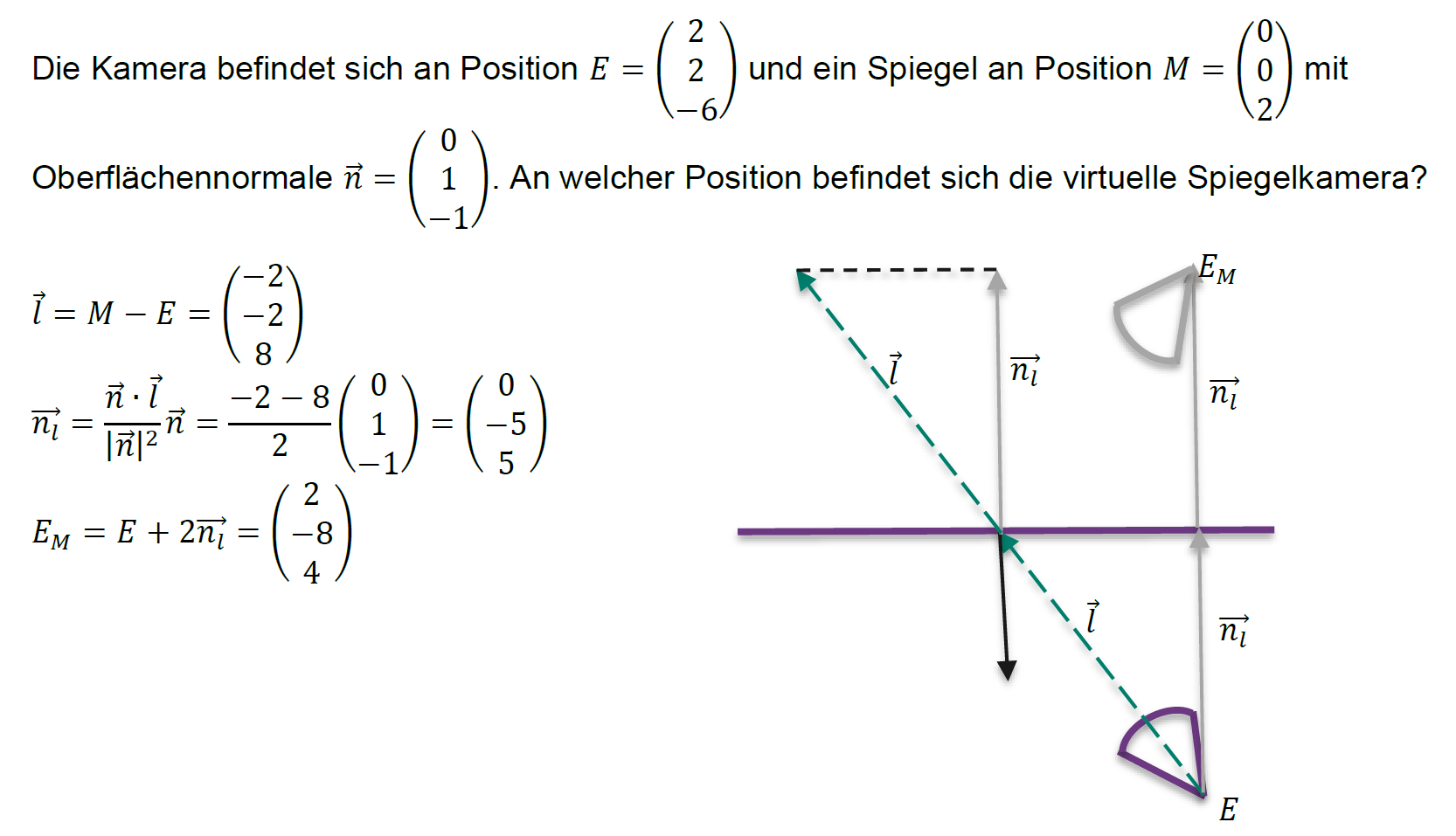
vec3 halfwayDir = norm(**lightDir+camDir**);  
float cosTheta = max(**dot**(**normDir**,halfwayDir),0.0);

Texturen

Bilddateien auf Oberflächen. Mapping zwsch. Vertex-koodinaten (x, y, z) und Textur-Koordinaten (u, v)

Spiegelung

OpenGL: Projektion auf Spiegeloberfläche mit virtuelle Spiegelkamera. Als Textur auf Spiegel kleben.  
Bei Kegel: Bounding-Box nutzen (Environmentmapping mittels OpenGL Cubemap)



Schatten

* Projiziere Objekte auf belichtete Oberfläche
  + Mit Kamera an der Position der Lichtquelle
  + Oberfläche als Projektionsfläche
  + Ableitung Tiefenwerte (Nicht Farbe).
* Depth-Map: Schwarz = Objekt Nahe (z=0), Weiss = Objekt weit weg (Z=unendlich), Umrechnung linear
* Mit Textur: Eine Textur speichert Farbe, andere die Distanz der Lichtquelle zum nächsten Objekt.

Ray Tracing

* Von Kamera aus durch jeden Bildpixel ein Strahl. Farbe des Pixel hängt dann von Objekt ab, das getroffen wird. 🡺 Bildzentriert
* Bei Verdeckung oder schneiden mehrere Objekte wird nächstes Objekt genommen.
* Rekursives Weiterverfolgen: Reflektion und Schattenstrahl wird genutzt (Sekundärstrahlen). Strahlen können beliebig oft reflektiert werden. Grenze muss aber gesetzt werden, damit aufhört. Bei Rückweg wird Farbe und Licht-Intensität bestimmt.
* Baumstruktur wenn mehrere Folgestrahlen.
* Schneiden von Strahlen ist Zentrale Operation. Je nach dem Zeitintensiv (Triangle Meshes). Implizit dargestellte Objekte können effizient berrechnet werden. (Einsetzen des Strahls in Funktion.)

Ein Bild, das Tisch enthält.

Automatisch generierte Beschreibung

Akzeleration Structures

Datenstrukturen um Anzahl Schnitt-Tests zu reduzieren. *Element-Teilend*: Bounding Volumes Hierarchy, *Raum-Teilend*: Uniform Grid; Quad-Tree

Ein Bild, das Tisch enthält.

Automatisch generierte Beschreibung

Triangulation (Punktwolke)

Aufteilung Fläche in Dreiecke, Eckpunkte vorgegeben

* *Sweep-Strategie*: Inkrementelles Hinzufügen von aussen liegendem Punkte
  + Konvexe Hülle: Umschliesst form ohne Delle, ermöglicht sichtbare Punkte zu bestimmen
* *Insert Stategie: 1.* Künstliches Anfangsdreieck  
  2. Inkrementelle Triangulierung: 2a) Zufällige Auswahl nächster Punkt 2b) Suchen des Dreiecks, das Punkt umfasst 2c) Unterteilung des Dreiecks,  
   3. Entfernen der Anfangspunkte
* Edge-Flip: Gibt viele Lösungen für eine Triangulation
* Delaunay Triangulation: Für alle 3eck-Paare gilt: Kürzere Diagonale ist Teil der Triangulation

Heat Maps

Abbildung von Funktionen. Mittels Isolinien kurven darstellen. Dazu Marching-Squares Algorithmus.