**Allgemein**

* Physiologischer Hintergrund:
  + Zäpfchen: Licht (ca. 100 Mio.)
  + Stäbchen: Farbe (rot, grün, blau) (ca. 6 Mio.)

**Farbmodelle**

* RGB:
  + Rot, Grün und blau
  + Additives & multiplikatives Verfahren
  + wird verwendet in Monitoren
  + 8-bit unsigned integer
* CMY
  + Cyan, Magenta, Yellow
  + Subtraktives Verfahren -> Farben werden zu weißem Hintergrund hinzugefügt
  + verwendet in Druckern
  + Erweiterung CMYK:
    - Graustufe wird hinzugefügt
    - K = min(C, M, Y) 🡪 C‘ = C – K, ...
* HSV
  + Hue (Farbton), Saturation (Sättigung), Value (Brightness)
  + basiert auf RGB

**Gamma**

* CRT-Displays:
  + lineares Wachstum bei Spannung != lineares Wachstum der Helligkeit
* zur Korrektur eingeführte Gamma-Berechnung
  + γ ~ 2,2 🡪 bei Bildschirm kommt xγ an
  + zur Korrektur: xout = xin1/γ
* in modernen Displays immer noch Simulation von Gamma-Korrektur

**Triangles and Pixels**

* Dreiecke sind primitive Flächen 🡪 alle möglichen Polygone können zu Dreiecken konvertiert werden
* Verfahren zum Speichern:
  + sequenzielle Liste:
    - alle Vertizes von jedem Dreieck in einer Liste speichern
    - N \* 3 \* sizeof(vertex) bytes
  + Indexed Face Set (IFS):
    - jeder Punkt wird einfach gespeichert, für Dreiecke nur Indizes der Vertizes
    - N \* 3 \* sizeof(unit) + Nvertices \* sizeof(vertex) bytes

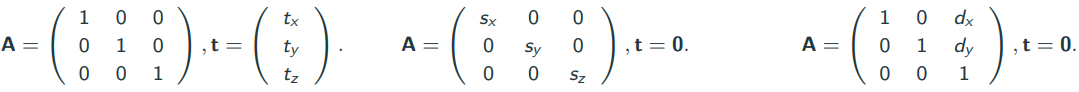
**Rasterisierung**

* Single-Sample-Rasterisierung:
  + Mittelpunkt des Pixels muss in Dreieck liegen
    - Strecken berechnen & normieren
    - Normalen von Strecken berechnen & ggf. normieren
    - Punkt in Relation zu Strecke setzen
    - Skalaprodukt ausrechnen
      * positiv: Punkt in Dreieck
      * negativ: Punkt nicht im Dreieck
* Signed-Distance Rasterisierung *Convexe* Polygone
  + Bounding Box um Dreieck 🡪 nur Pixel prüfen, die in Frage kommen können
  + Winding Order: CCW (Counter Clock Wise)
  + Hierarchisch: Divide & Conquer 🡪 Unterteilung der Bounding Box in immer kleinere Vierecke
    - wenn Eckpunkte alle innerhalb: accept
    - wenn keine Kante Dreieck schneidet: reject
* Scanline *Concarve* Polygone
  + für jeden Höhenwert wird ein Pixel auf jeder Linie gewählt
  + danach wird links nach rechts zwischen gewählten Pixeln aufgefüllt
    - EdgeTable (ET) aufstellen 🡪 Sortierung nach Ylower
      * Inhalt: Ylower, Xlower, Yupper, m
    - AET: immer 2 Kanten raussuchen, bis Yupper erreicht wird, immer eine Stufe höher gehen & Punkte markieren / rausschreiben

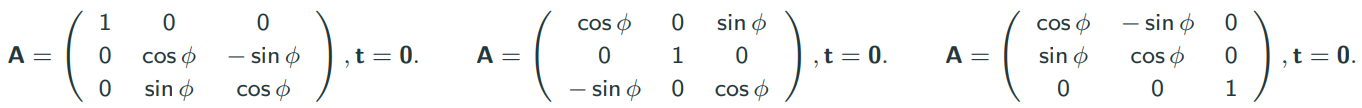
**Interpolation**

* Lineare Interpolation bzw. lerp:
  + 🡪 P liegt in diesem Fall auf
* Baryzentrische Koordinaten:
  + Interpoliert von den verschiedenen Eckpunkten, z.B. für Farbzuweißungen

**Transformationen**

* Affine:
  + Lines werden zu Lines, Parallelität bleibt bestehen, Größenverhältnisse bleiben gleich
  + dazu zählen Verschiebung, Skalierung, Scherung und Drehung

Verschiebung Skalierung Scherung



X-Rotation Y-Rotation Z-Rotation

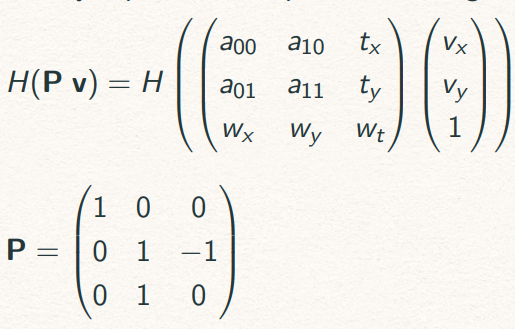
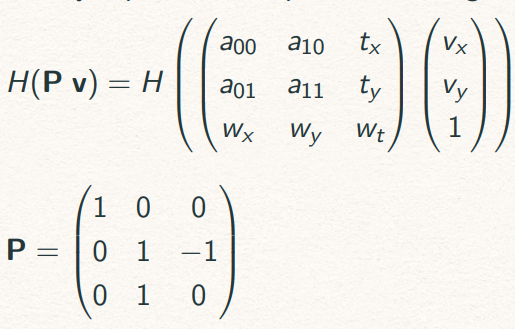
* Transformationen können kombiniert werden
* Homogene Koordinaten:
  + Translationsmatrix wird um eine Koordinate erweitert:
    - homogenisieren:
    - dehomogenisieren:
  + Kombination durch einfache Multiplikation der Matrizen (von hinten)
    - z.B. 🡪 Skalierung **vor** Translation, deshalb T \* S und nicht S \* T
* Normalen müssen renormalisiert werden
  + zum renormalisieren werden Normalen mit der Inversen der Transformationsmatrix transformiert
* Rotationen:
  + immer gleiches Schema:
    - Drehachse bleibt gleich
    - bei den anderen: 🡪 bei y als Achse wird – vertauscht
  + Euler-Winkel:
    - **Jede Rotation kann durch 3 Rotation um die Hauptachsen (X, Y, Z) erreicht werden**
    - willkürlich kombinierte Rotation um die 3 Achsen, z.B. um Z, Y und Z
    - Vorteile: Einfache Implementierung, Einfach zu benutzen, nur 3 Winkel nötig
    - Nachteile: Animationen nicht einfach, Gimbal Lock
    - Gimbal Lock:
      * Bei Rotation um 90° wird eine Achse unbrauchbar
      * anschließend sind 2 Ursprungsachsen übereinander
  + Quaternionen:
    - 2D:
    - Quaternion Punkt:
    - Quaternion Winkel: mit **uq** als Rotationsachse
    - Conjugate Quaternion:
    - Multiplikation:

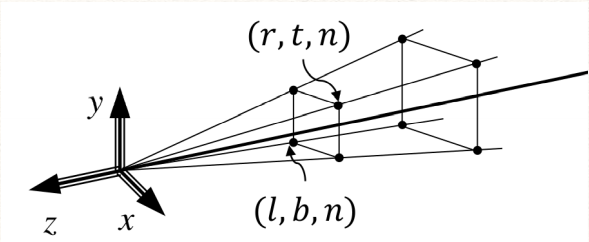
**Koordinatensysteme**

* „World“ Koordinatensystem (globales) wird definiert, anschließend alle Objekte anhand von Transformationen (Plazierung, Orientierung, Skalierung) in der Welt positioniert
  + Lokales auf globales Koordinatensystem setzen, dann rotieren, skalieren und an geeignete Position schieben
    - Multiplikation des Punkts mit jeweiligen Matrizen
* alle Transformationen, die gebraucht werden, um lokales System auf globales zu übertragen, werden in Matrix M zusammengefasst
  + durch Multiplikation von Punkt p mit M kann **lokaler Punkt in globales** System übertragen werden
  + durch Multiplikation von Punkt q mit M-1 kann **globaler Punkt in lokales** System übertragen werden
  + zur Invertierung einer starren Transformation:
    - transponierte Rotationsmatrix \* invertierter Translationsteil \* Punkt im globalen System

**Viewing**

* Virtueller Beobachter = Kamera hat neues Koordinatensystem
* Objekte werden von World in Eyespace übertragen 🡪 Projektion der Punkte auf 2D-Fläche
  + Right-handed Koordinatensystem
* **Eye Space** 
  + Matrix E ist Kombination aus Translation T und Rotation R:
    - Punkt wird zuerst um Ursprung gedreht, dann verschoben
  + Viewing Direction g und Up-Vector t für Rotation
* **normalized device coordinates (NDC)**
  + Maximale Abmessungen werden auf Koordinaten angewandt und auf [-1, 1] Intervall gemapped (für HW)
  + Tiefe wird mit einberechnet, dafür gibt es Abmessungen n und f
  + Projektionsmatrix berechnet NDC-Koordinaten für Punkt
* **Viewport Coordinates**
  + Viewport-Größe wird in w und h angegeben, NDC-Koordinaten werden damit verrechnet & so an Viewport angepasst
  + Matrix W zur Berechnung der viewport-Koordinaten
* **orthographische Projektion**
  + keine Skalierung der Objekte
  + Projektionsmatrix hat „leeren“ homogenen Teil, also nur Einberechnung der Translation
* **perspektivische Projektion**
  + Skalierung auf n abhängig von Abstand zur Kamera
  + Foreshortening: Dinge werden in der Distanz kleiner
  + Projektionsmatrix 2D:
    - homogener Anteil nicht mehr 0 0 1, sondern Einberechnung y-Wert für Entfernung
    - Translationsteil: y-Koordinate wird mit -1 einberechnet, damit Entfernung nicht verloren geht



* + Projektionsmatrix 3D:
    - The Viewing Frustum:
      * z geht ins Negative
    - Einberechnung von z-Anteil, Berechnung durch Abmaße Höhe h und Weite w
      * near distance n, aspect ratio a, field of view
    - Berechnung von z‘ zum Bestimmen, welche Objekte gezeichnet werden sollen
      * z‘ bezeichnet Koordinate z im NDC
      * wenn z‘ < -1 oder z‘ > 1, dann ist das Objekt außerhalb der Sicht
      * near plane liegt bei z‘ = -1, far plane liegt bei z‘ = 1

**Clipping**

* nur sichtbare Triangles sollen gerendert werden
* muss im Eye Space mit Grenzen gemacht werden, da sich sonst die Geometrie verändert
* in NDC wäre Clipping einfacher, aber Geometrie verändert sich, wenn z > 0 ist
* Ordnung von Triangles:
  + Depth Buffer:
    - speichert den NDC-Z-Wert für jeden Pixel
    - wenn neuer Z-Wert niedriger ist als gespeicherter Z-Wert wird Pixel überschrieben, sonst nicht

**OpenGL**

* Idee: IFS auf GPU Speichern, Rendering für komplettes Modell auf einmal starten
* Vertex und Index Buffers werden angesprochen
  + Buffer auf RAM wird erstellt durch glGenBuffer(size, \*buffer)
  + Binden des Buffers an GPU Buffer glBindBuffer(target, buffer)
    - Bindung muss am Ende wieder aufgelöst werden
    - target: für Vertices GL\_ARRAY\_BUFFER, für Indizes GL\_ELEMENT\_ARRAY\_BUFFER
  + Daten von Buffer auf GPU laden glBufferData(target, size, \*data, flags)
* **Alternativ**: Benutzen von Vertex Array Objects, die alle bindings und attribute links beinhalten
  + Erstellen durch glGenVertexArrays(size, \*vaos)
  + Binden des Vertex Array Objects glBindVertexArray(array)
* Vorbereitung zum Rendern
  + Vertex Stream aktivieren glEnableVertexAttribArray(index)
    - Vertex Stream: Buffer wird an GPU Stream Input angebunden
  + Vertex Stream spezifizieren glVertexAttribPointer

(index,size,type,normalized,stride,\*data)

* + - GL\_ARRAY\_BUFFER gibt die jeweiligen Vertices an den Shader
* Starten vom Rendering
  + Rendern mithilfe von Vertices glDrawArrays(mode, first, count)
  + Rendern mithilfe von Indizes & Vertices glDrawElements(mode, count, type, \*indices)
    - \*indices wird auf 0 gesetzt, dadurch wird GL\_ELEMENT\_ARRAY\_BUFFER benutzt
  + Typen für beide Methoden: GL\_TRIANGLES, GL\_QUADS...

**Shaders**

* Vertex Shader
  + läuft für jeden einzelnen Vertex
  + transformiert Punkt durch Renderpipeline bis zum Clip Space
* Fragment Shader
  + zuständig für die Einfärbung von Pixeln
  + Signed Distance Verfahren läuft im Fragment Shader
  + außerdem Berechnung der Farbe
* Shader werden in eigenen Files geschrieben, aber müssen in OpenGL erstellt werden
  + Erstellung glCreateProgram()
  + Shader Stage zuweißen glCreateShader(shader\_type)
    - shader\_type kann GL\_VERTEX\_SHADER, GL\_FRAGMENT\_SHADER... sein
  + Kompilierung des Shaders: Zuweißung glShaderSource(shader,count,\*\*string,\*length)
  + Kompilierung glCompileShader(shader)
  + Error-Check glGetShaderiv(shader, param\_name, \*values)
  + Shader zu Programm zuweißen glAttachShader(program, shader)
  + Programm danach linken glLinkProgram(program)
  + Resultat checken glGetProgramiv(program, param\_name, \*values)
  + Programm benutzen glUseProgram(program)

**OpenGL Shading Language (GLSL)**

* die main()-Funktion wird jedes Mal, wenn der Shader aufgerufen wird, ausgeführt
* Versionsspezifizierung #version 130
* Spezifiziert über Main-Funktion
  + Input **in** vec4 local\_vertex
  + Output **out** vec4 out\_color
  + Daten, die unabhängig vom Punkt sind **uniform** mat4 proj
* main-Funktion void main () {...}
* implizite output-Variable für Punkt gl\_Position

**Lighting**

* Physikalische Licht – Flächen Interaktionen:
  + Absorption Licht wird absorbiert
  + Transmission Licht wird weitergeleitet
  + Reflektion Licht wird reflektiert
  + Refraction Licht wird gebrochen
  + Scattering ungleichmäßige Reflektion
* lokale Beleuchtung
  + Licht wird nur von einem Punkt aus betrachtet
  + der Rest der Szene ist irrelevant
  + kein indirektes Licht
* globale Beleuchtung
  + Lichtaustausch zwischen Objekten & Lichtquellen
  + Schatten (soft Shadows)
  + Licht kann wandern & von Objekten reflektiert werden 🡪 indirektes Licht
* **Phong Lighting**
  + nutzt zum Berechnen der Beleuchtung die Richtung der Lichtquelle, die Richtung der Kamera und die Normale des aktuellen Fragments
    - die Kamera- und Lichtrichtung sind im Voraus bekannt
  + Lichttypen:
    - Directional light source unendlich weit entfernte und große Lichtquelle
      * parallelle Lichtstrahlen
      * Lichtrichtung l ist konstant
    - Point light source Lichtquelle ist Punkt im Raum
      * Licht wird von Punkt p aus in alle Richtungen gestrahlt
      * Lichtrichtung l von Punkt x aus:
      * Die Distanz zum Licht bestimmt die Intensität
    - Spot light source
      * Variante des Punktlichts
      * Kegel um Licht 🡪 Öffnungswinkel und Richtung sind Parameter
      * Licht wird durch den Kegel begrenzt
      * Lichtintensität wird am Äußeren des Kegels niedriger
  + Komponenten der Reflektion **Phong Reflection**
    - Ambient light
      * Aufleuchtung durch Umgebung, keine Spotlights
        + kamb gibt an, wie ein Objekt indirekt aufgehellt wird (Farbe)
        + iamb gibt an, wie viel indirektes Licht vorhanden ist (Intensität)
    - Diffuse reflection
      * Reflektion von matten Oberflächen
      * **Lambert’s cosine law: Die Intensität der Reflektion ist abhängig vom Eintreffwinkel der Lichtquelle**
        + = Eintreffwinkel des Lichts zur Normalen = Lichtrichtung
        + wenn negativ / kleiner 0 ist, wird Fragment nicht vom Licht bestrahlt
    - Specular reflection
      * Reflektion von glänzenden Oberflächen
      * Licht wird kegelförmig reflektiert, da sonst nichts sichtbar wäre (Lichtquelle ist nur ein Punktt)
      * Intensität wird kleiner, je weiter man sich von der Hauptrichtung der Reflektion entfernt
      * + **= Shinyness exponent** bestimmt, wie stark die Reflektion ist

hoch: glänzender, niedrig: eher diffus, : Reflektion kleiner, aber nicht heller

* + - gesammelte Formel:
      * bei N Lichtquellen:

**Shading**

* Flat Shading
  + Jede Fläche hat eine konstante Normale
  + Die Beleuchtung ist abhängig von der Normalen
  + Eine Farbe / Fläche
* Gouraud Shading
  + An Punkten vom Polygon wird Beleuchtung berechnet
  + durch Interpolation der Punkte wird die komplette Fläche berechnet
* Phong Shading (!= Phong Lighting)
  + Vertex-Punkte und Normalen werden interpoliert
  + Für jedes Fragment wird die Beleuchtung einzeln berechnet

**Texturen**

* Texturatlas:
  + viele Texturen für zusammenhängende Flächen (z.B. Charakter) werden in einer Textur zusammengefasst
  + Mapping der Dreiecke des Modells auf den Texturatlas
  + erstellt durch Designer, **Texture Painting** auf 3D-Modell, danach kann Textur „aufgeklappt“ werden
  + Textur Wrapping:
    - Für Texturen, die außerhalb des definierten Bereichs sind, werden verschiedene Modes aktiviert
    - So wird die Textur z.B. wiederholt, gestreckt, gespiegelt etc.
* Texture Sampling:
  + Verfahren:
    - Simple Solution
      * Pixel nimmt die Farbe des Texels der am nähesten da ist (Mode: GL\_NEAREST)
      * Problem: detaillierte Texturen werden stufig / pixelig
    - Bilinear interpolation
      * Durch Interpolation werden die 4 umliegenden Pixel einberechnet
  + Probleme:
    - Magnification Problem: niedrig aufgelöste Texttur, kleine Pixel -> scharfe Kanten gehen verloren
    - Minification-Problem: hoch aufgelöste Textur, weite Entfernung (Pixel größer als Texel)
      * Textur geht verloren
  + Mip Mapping:
    - Texturen werden im Voraus herunterskaliert 🡪 meist schrittweiße in 2 x 2 zu 1 x 1 Schritten
    - Je nach Pixelgröße wird entsprechendes Texturlevel zum Einfärben benutzt
    - Level Selection:
      * p = ratio projected pixel to texel size
      * Lbias wird festgelegt, entweder mehr aliasing oder stärkere Verunschärfung
      * Problem:
        + Niedrig aufgelöste Texturen können verwischen
        + Dadurch entsteht ein „harter“ und erkennbarer Übergang
    - Level-Selection verfeinern durch Trilineare Interpolation zwischen und
      * jeweilige lineare Interpolation der Farbe im oberen & unteren Level
      * lineare Interpolation zwischen den beiden Farbwerten
* GL Befehle bis hier:
  + Textur abrufen vec4 texture(sampler, tex\_coord)
  + Texturvariable uniform sampler2D diffuse\_texture
  + Texturkoordinaten in vec2 tc
  + Farbzuweißung vec3 k\_diff = texture(diffuse\_texture, tc)
    - Farbwerte speichern texture(...).rgb
  + Texture wrapping glTexParameteri(target, pname, param)
    - Modes: GL\_REPEAT, GL\_MIRRORED\_REPEAT, GL\_GLAMP\_TO\_EDGE, ...
    - Sampling Verfahren: GL\_NEAREST & GL\_LINEAR
* Perspektive:
  + Interpolation im NDC != Interpolation im Eye Space
* Texturarten:
  + **Diffuse**: Details und Farben für die Geometrie
  + **Specular**: Glanz & Mattierung für die Textur
  + **Mask**: Gibt vor, welcher Teil der Textur benutzt wird

**Verschiedene zusätzliche Texturen:**

* Normal Maps zum Speichern der Normalen
  + Object Space
    - Normalen werden auf dem Objekt für jedes Fragment festgelegt
    - Berechnung: vec3 n = vec3(2)\*x – vec3(1)
      * x sind „Farbwerte“, die aus der Textur gelesen wurden mittels texture(...).rgb
  + Tangent Space (viel besser)
    - Basis: TBN-Matrix (Achsenbezeichnung
    - Nicht nur Normale, sondern auch Tangentiale Achsen werden in der Textur abgelegt
    - Berechnung der T- und B-Achsen durch die Differenzen des Dreiecks in u- und v-Richtung
      * =
        + e sind Strecken b-a bzw. c-a, u (T-Achse) und v (B-Achse) die zugehörigen Differenzen
    - Normale n wird aus Tangent-Space genommen, damit wird die Beleuchtung berechnet
  + Probleme:
    - Mip-Mapping funktioniert nicht gut
      * Normalen können sich zwischen 2 Texeln stark unterscheiden, Mittelwert hat andere Länge & Richtung
* Bump Maps
  + Höhe der einzelnen Texel wird gespeichert
  + „Normalhöhe“ ist bei ½
  + Wird meisten zu Normal Maps konvertiert
* Environment Maps
  + Skybox
    - Cube Maps
      * hat verschiedene Texturen für verschiedene primäre Richtungen
        + GL: texture(samplerCube...)
    - Longitude / Latitude Maps
      * für kompletten Himmel
      * von Richtung d aus kann durch folgende Formeln die Texturkoordinate berechnet werden:
  + Billboards
    - Hintergrundobjekte, die nicht viel Beachtung bekommen
    - durch Mask-Textures kann einfach viel Hintergrund dargestellt werden
    - oft bei Gras der Fall

**Shadow Mapping**

* globaler Effekt 🡪 Objekte die Schatten werfen sind in der lokalen Beleuchtung nicht verfügbar
* Idee: Bild aus Sicht der Lichtquelle rendern
* Render-to-Texture:
  + Framebuffer wird erstellt glGenFramebuffers(size, \*buffer)
  + Framebuffer wird gebunden glBindFramebuffer(GL\_FRAMEBUFFER, framebuffer)
  + Texturen anhängen

glFramebufferTexture2D(target, attachment, textarget, texture, level)

* + - attachments:
      * color GL\_COLOR\_ATTACHMENTO
      * depht buffer texture GL\_DEPTH\_ATTACHMENT
    - textarget GL\_TEXTURE\_2D
  + im Framebuffer:
    - Viewport setzen glViewport(x, y, width, height)
    - Output bestimmen glDrawBuffers(size, \*bufs)
* Rendern der Shadow Map
  + Zusätzlich zu shading point, camera point und light direction wird die kürzeste Distanz zum Licht gespeichert
    - zweifaches rendern zum Bestimmen der kürzesten Distanz
    - zum Rendern: glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR)
* Berechnung:
  + Berechnung der NDCs für Kamera und Licht
  + Vergleich der Z-Koordinaten von Punkt v1 und Punkt v2 in Licht-NDC
    - wenn v1 > v2 🡪 **v1 liegt im Schatten von v2**
* Benutzung:
  + Texture Lookup: d = texture(shadowmap, tc).r
  + Shadow Lookup: float s = texture(shadowmap, tc).r
  + aktuell rasterisiertes Fragment wird in die Shadowmap projiziert und mit dem Wert verglichen
* Probleme:
  + Shadow Acne
    - durch grobe Rasterisierung werden Punkte teilweise als „im Schatten“ angesehen, obwohl sie das nicht sind
    - gelöst durch Verrechnen der Pixel mit Offset
      * Aktivieren: glEnable(GL\_POLYGON\_OFFSET\_FILL)
      * Offset: glPolygonOffset(factor, units)
      * Deaktivieren: glDisable(GL\_POLYGON\_OFFSET\_FILL)
  + Resolution Missmatch
    - Aliasing ist Problem
    - Auflösung ist teilweise zu schlecht auf kurze, aber zu hoch auf weite Entfernungen
    - Mip-Mapping nicht gut möglich, da nach ein paar Stufen nichts mehr von der Geometrie übrig bleibt
* Interpolation
  + für die GPU muss extra spezifiziert werden, dass der sampler eine shadowmap ist
    - GL: texture(sampler2Dshadow sampler, P, [float bias])
    - dadurch wird der shadow lookup abgefragt
  + Interpolation in der texture(...) Funktion funktioniert bei Tiefenwerten nicht, weil der Durchschnitt keine sinnvolle Funktion hat
  + wenn Hardwarefiltering angeschaltet ist, wird aus 4 verschiedenen Tiefenwertberechnungen der interpolierte Wert berechnet

**Formeln:**

* Gamma-Korrektur:
* lerp:
* Baryzentrische Koordinaten: *p = a + β ∗ (b − a) + γ ∗ (c − a)*
* renormalisieren:
* near distance n, aspect ratio a, field of view
* z-Koordinate im NDC:
* Phong Lighting:

  + bei N Lichtquellen:

**OpenGL Shading Language (GLSL)**

* die main()-Funktion wird jedes Mal, wenn der Shader aufgerufen wird, ausgeführt
* Versionsspezifizierung #version 130
* Spezifiziert über Main-Funktion
  + Input **in** vec4 local\_vertex
  + Output **out** vec4 out\_color
  + Daten, die unabhängig vom Punkt sind **uniform** mat4 proj
* main-Funktion void main () {...}
* implizite output-Variable für Punkt gl\_Position

**Beispiel Phong Shading mit point-light:**

*// Vertex Shader Phong Shading mit Phong Lighting*

in vec3 vert\_pos; *// vertex position*

in vec3 norm; *// fragment normal*

uniform mat4 P; *// Projection Matrix*

uniform mat4 V; *// Viewing Matrix*

uniform mat4 M; *// Model Matrix*

uniform mat4 M\_normal; *// Model Matrix for normal, inverted model matrix*

out vec4 pos\_ws; *// World Space position*

out vec3 norm\_ws; *// World Space normal*

int main() {

    pos\_ws = M \* vec4(vert\_pos, 1.0); *// position in World Space*

    norm\_ws = normalize(M\_normal \* vec4(norm, 1.0)).xyz; *// normal in World Space*

    gl\_Position = P \* V \* pos\_ws; *// point position in clip space*

}

*// Fragment Shader Phong Shading mit Phong Lighting*

in vec3 pos\_ws; *// position in world space*

in vec3 norm\_ws; *// fragment normal in world space*

uniform vec3 pointlight\_pos; *// position of point light*

uniform vec3 cam\_pos; *// position of cam*

*// color & intensity for diffuse lighting*

uniform float k\_diff;

uniform float i\_diff;

*// color & intensity for specluar lighting*

uniform float k\_spec;

uniform float i\_spec;

uniform float n\_s; *// shininess exponent*

out float color;

int main() {

*// diffuse lighting*

    vec3 n = normalize(norm\_ws); *// normalize normal*

    vec3 l = normalize(pointlight\_pos - pos\_ws); *// normalize pointlight vector to point --> light vector*

    float diff = k\_diff \* i\_diff \* max(dot(n, l), 0); *// diffuse lighting*

*// specular lighting*

    vec3 v = normalize(cam\_pos - pos\_ws); *//normalized view-vector*

    vec3 r = 2 \* n \* dot(n, l) - l; *// normalized r-vector (r = 2\*n\*<n|l> - l)*

    float spec = k\_spec \* i\_spec \* pow(dot(r,v), n\_s); *// specular lighting*

    color = diff + spec; *// color = amb + diff + spec*

}