Computergraphik 1 LU (186.095)

Ausarbeitung Abgabe 2

Beispiel 4

CG1Object

if (doBackfaceeCulling && numVertexIndec >=3)

Bei dieser Aufgabe geht es darum Backface Culling zu programmieren. Da man aber eine 2D Geometrie braucht („Face“), und somit mindestens eine Geometrie mit 3 Eckpunkten haben muss (1 ist Punkte, 2 ist eine Strecke), überprüft man bei **if (doBackfaceeCulling && numVertexIndec >=3)** ob es mehr als 3 Punkte hat.

Dann berechnet man mit **CG1Vector.diff (transVertices[faceTable[i][2]], transVertices[faceTable[i][3]]);** und **CG1Vector.diff (transVertices[faceTable[i][2]], transVertices[faceTable[i][4]]);** die beiden Vektoren. Dabei verwendet man **transVertices** da dort die Punkte NACH der Transformieren in der Pipeline gespeichert sind. In [0] ist der Vertexcount gespeichert, in [1] colourIndex, in [2] und [3] und [4] die eigentlichen Punkte des Face. Anschließend speichert man sie ab.

Nun berechnet man via **CG1Vector.crossProduct** das Kreuzprodukt der beiden Vektoren. Das macht man, da das Kreuzprodukt die Normale zum Aufgespannten „Face“ der 2 Vektoren darstellt.

Am Schluss muss man noch überprüfen, ob das Kreuzprodukt kleiner 0 (sprich es ist nicht sichtbar) ist zum Beispiel mit **if (Kreuzprodukt.z < 0)**. Wenn es true ist, sprich es in die „falsche“ Richtung zeigt da es wegschaut, ist es nicht sichtbar und wird nicht gezeichnet. Mit **{continue;}** wird dann weitergemacht in der Schleife und es kommt nicht zum **polygon.draw (canvas);** weiter unten. Wäre es false, würde man aus der if Schleife heraus kommen, zum **polygon.draw (canvas);** kommen und würde es zeichnen da es sichtbar ist

Detail am Rande: Würde man bei **if (Kreuzprodukt.z < 0)** das **<** zu einem **>** machen, hätte man das „Gegenteil“, nämlich Frontface Culling das die Vorderseite „löscht“ und die Rückseite lässt. Dies braucht man für einige Effekte wie Shadow Volumes.

CG1ScanfilledPolygon CG1Polygon

Bei dieser Datei geht es darum, gefüllte Polygone zu zeichnen.

buildEdgeList ()

In dieser Methode geht es darum, dass edgelist[] Array zu füllen

Zuerst machen wir 2 Arrays, da eine Kante ja aus einem Anfang und einem Ende besteht. Da wir im Moment mit X und Y Werten arbeiten, sollte man die auch dort speichern können. Also machen wir 2 Mal **new int[2];** und speichern das zum Beispiel in **list1** und **list2**. Da wir später eine temporäre

Variable brauchen, legen wir die am besten auch gleich an um nicht darauf zu vergessen und alles beisammen zu haben

Jetzt sollten wir in list1 unser X und Y geben und geben dort das letzte Element hinein. Die Punkte sind in clipped und deswegen macht man das mit **clipped[numVertex - 1][X];** bzw. entsprechend mit **Y** für **Y**. Man verwendet dabei **numVertex – 1**, da man ja aufs letzte Element zugreifen will. Wenn man ein Array mit 10 Elementen hat, beginnt das mit 0 und geht bis 9, sprich 10-1. Allgemein gesagt: Anzahl-1. In unserem Fall eben numVertex – 1.

Nun durchläuft man in einer Schleife (ich machte eine **for Schleife**) alle numVertex. Jetzt macht man das Befüllen der list2 genauso wie zuvor mit list1.

In einer **if Schleife** wird abgefragt ob **(list1[Y] != list2[Y])**. Wenn es Unterschiede gibt, soll fortgeführt werden, ansonsten übersprungen werden und nachher weitergemacht werden.

Nun muss man nur mehr schauen, ob die Kante rauf oder runter geht und entsprechend rechnet man **(double)(list2[X] - list1[X])/(double)(list2[Y] - list1[Y]);** im Falle von **(list1[Y] < list2[Y])**, ansonsten mit **(double)(list1[X] - list2[X])/(double)(list1[Y] - list2[Y]);**

Dann speicherte ich es via **edgelist[list1[Y]].insert (list2[Y] - 1, (double)list1[X], temp);** ab (glaube das ist yOben, xIntersect und dann die Änderung pro Scanline).

Nach der Schleife wird immer **list1[X] = list2[X];** und **list1[Y] = list2[Y];** gemacht, damit man in list1 den Punkt von list2 hat und list 2 wird dann wieder in der Schleife neu befüllt.

fillScan (int scan, CG1Canvas canvas)

Zuerst speichert man die Anfangskante und die darauf folgende Kante mit **activelist.getHead** fürs erste Element bzw. **Anfang.next;** (Anfang da ich activelist.getHead in Anfang speicherte) für die Kante danach in 2 Variablen ab.

In einer **while Schleife** überprüft man immer ob **(Anfang != null)** und zeichnet anschließend in einer **for Schleife** mit **canvas.setPixel (i, scan, color);** das Pixel. In „Anfang“ könnte es sein das nichts drinnen ist, da z.b. die zu zeichnende Geometrie nicht den ganzen Bildschirm ausfüllt und man sich gerade unter oder über der Geometrie befindet. Die for Schleife macht man mit **(int i = (int)Anfang.xIntersect; i<(int)Naechstes.xIntersect; i++)**.

Am Schluss darf man nicht vergessen in der ersten Variable die nächste Kante hineinzugeben, damit diese stimmt und auch die zweite Kante richtig geladen werden kann (da die man über die Kante davor bekommt). Das kann man mit **Naechstes.next;** machen, falls man die zweite Kante in Naechstes gepeichert hat.

buildActiveList (int scan)

Während man bei EdgeList die Kanten im Canvas speichert, speichert man bei Active List die Kanten der „Scanline“. Wenn man zum Beispiel ein Canvas mit 600 *400 Pixel hat und ein Dreieck zeichnen will, hat man in der EdgeList 3 Einträge da es 3 Kanten gibt (wobei glaube ich 2 gleich sind da sie den gleichen Anfangspunkt haben, bin mir aber nicht sicher!). Bei ActiveList hätte man aber ein Array mit 400 Elementen und wenn das Dreieck das ganze Canvas ausfüllt, auch mit 400 Einträgen.

Zuerst holt man sich von der **edgelist** mithilfe des übergebenen Variable **scan** die Kante, sprich **edgelist[scan].getHead ();**. Am besten legt man sich auch gleich eine Variable für die zweite Kante an damit man nicht vergisst und alles zusammen hat.

In einer **while Schleife** überprüft man ob es **ungleich Null** ist. Anschließend speichert man in der zweiten Kante mit **Kante1.next;** die Nachfolgekante von Kante1.

Über **activelist.insert (Kante1);** speichert man die Kante in der „Liste“. Man darf dann nicht vergessen wieder Kante1 via **Kante1 = Kante2;** die Kante2 zu übergeben, damit Kante1 wieder korrekt ist und folglich Kante2 die abhängig von Kante1 ist wieder korrekt neu befüllt werden kann.

drawClipped (CG1Canvas canvas)

Zuerst überprüft man mit **if (numVertex == 0) return;** ob numVertex gleich 0 ist und wenn ja, hat man nichts zu „arbeiten“ und bricht deswegen ab. Ansonsten fährt man fort.

Ich speicherte zuerst in einer Variable hoehe die Canvasgröße via **canvas.getHeight();** ab und machte dann eine neues CG1EdgeList Array mit der Anzahl der horizontalen Pixel in dem ich **new CG1EdgeList[hoehe];** schrieb.

In einer **for Schleife**, die so lange läuft bis man alle Scanlines in horizontaler Richtung durchhat, sprich solange **i<hoehe;** true ist, speichert man pro Scanline die EdgeList ab. Dazu muss man mit **edgelist[i] = new CG1EdgeList();** zuerst eine anlegen, dann **mit buildEdgeList();** die Sachen Speichern (wie das geht haben wir schon oben bei der ersten Methode gehabt) und dann für die for Schleife die gleich danach kommt eine neue CG1EdgeList mit **activelist = new CG1EdgeList();** machen. Damit erzeugt man neue Werte der CG1EdgeList (der aktiven Scanline) an activelist.

Wie angekündigt kommt dann eine **for Schleife**, die so lange durchlaufen wird, bis man alle Scanlines durch hat (sprich so hoch Canvas ist und scan<hoehe true ist)

Darin führt man **buildActiveList(scan);** aus was nichts anderes macht als buildActiveList mit der aktuellen Scanline aufzurufen.

In einer **if Schleife** fragt man **!activelist.isEmpty()** ab um sicher zu sein das in der activelist etwas drinnen ist.

Danach sagt man mit **fillScan(scan, canvas);** welche Scanline und canvas man fillScan übergeben will. Konkret zeichnet er dann die Scanline vom ersten Schnittpunkt der ersten Kante bis zu der der zweiten Kante.

Dann führt man **activelist.update(scan);** aus, dass die Kante die schon bearbeitet wurde aus der Liste löscht. Mit **activelist.resort();** sortiert die Kanten in der Liste (falls Kanten sich kreuzen nach dem man sich zur nächsten Scanline bewegt hat).

Beispiel 5

CG1Canvas extends Canvas

Define a Z-Buffer

Der Z- Buffer wird in der Computergraphik verwendet um zu schauen welche Elemente einer Szene gezeichnet werden müssen und welche verdeckt sind und folglich nicht gezeichnet werden. Sprich was ist vorne, wird gezeichnet, was dahinter und wird nicht gezeichnet.

Hier an der Stelle des Programms braucht man nur die Variable anlegen, welches ich mit **private double[] z_Buffer;** machte.

clear ()

Mit **Arrays.fill (imageData, 0);** und **Arrays.fill (z_Buffer, Double.NEGATIVE_INFINITY);** füllt man alles mit 0 bzw. einer negativen unendlichen Zahl. (Weiß leider nicht mehr genau was imageData ist und so.)

resize ()

Zuerst liest man die Höhe und Breite aus mit **getWidth ();** und **getHeight ();**, multipliziert es anschließend (oder gleich in einem schritt) und speichert das Ergebnis in einer Variable ab. Nun weiß man wie viele Pixel man am Canvas hat.

Anschließend macht man ein neues **imageData** mit der Größe von **new int**[numPixel];

Danach füllt man den **Z- Buffer** mit dem numPixel (**new double[numPixel];**) und danach Arrays mit einer negativen unendlichen Zahl (**Arrays.fill (z_Buffer, Double.NEGATIVE_INFINITY);**)

setPixel (int x, int y, Color c, double z)

Zuerst wird mit **if (z <= z_Buffer[pos]) return;** geschaut, ob der übergebener Z Wert außerhalb der Z-Achse liegt und wenn es true ist, also außerhalb ist, wird **mit z_Buffer[pos] = z;** der Z- Buffer auf die Länge von Z beschränkt.

Anschließend wird mit imageData[i] = (c.getRed() << 16) | (c.getGreen() << 8) | c.getBlue(); die Farben gesetzt.

CG1FlatShadedPolygon extends CG1ScanfilledPolygon

Die beiden Methoden in der Datei heißen genauso wie die Methoden aus der Datei CG1ScanfilledPolygon aus der geerbt wird (wie man schon am extends sieht) und überschreiben sie (die buildEdgeList und fillScan). Da stellt sich natürlich die Frage „Wozu das alles?“.

Die Antwort ist sehr einfach: Zuvor arbeiteten wir mit X und Y Koordinaten (da wir 2D Objekte hatten), jetzt wollen wir aber auch mit 3D Objekten Arbeiten, und die haben eine Z Koordinate (nicht zu verwechseln mit dem Z- Buffer der was GANZ anderes ist!). Deswegen machen wir jetzt alles noch mal, nur mit einer zusätzlichen Koordinate und überspeichern die alte buildEdgeList und filScan.

drawClipped, welches fillScan aufruft, tut das weiterhin ohne Problem („extends“).

buildEdgeList ()

Da schon zuvor alles hoffentlich verständlich erklärt wurde, es sehr ähnlich ist, werde ich mich eher kurzfassen.

Wir legen wie zuvor **2 Arrays an mit [2]**, und dann, dass ist jetzt neu, auch **2 double Variablen für Z**.

Nun geben wir in X, Y und Z die Werte hinein, sprich für **X clipped[numVertex - 1][X];**, für **Y clipped[numVertex - 1][Y];** und für **Z depth[numVertex - 1];**.

Nun durchläuft man in einer **for Schleife** alle Punkte (**i<numVertex;**).

Dort speichert man dann ins zweite Array bzw. zweite double Variable die Werte an der aktuellen Stelle ab (für **X clipped[i][X];**, für **Y** folglich **clipped[i][Y];** und für **Z depth[i];**).

Anschließend überprüft man wieder ob der Inhalt ungleich ist via **if (vertexspeicher1[Y] != vertexspeicher2[Y])**.

Wie bei Beispiel 4 schaut man wieder in welche Richtung es geht (**if (vertexspeicher1[Y] < vertexspeicher2[Y])**) und mit **edgelist[vertexspeicher1[Y]].insert(vertexspeicher2[Y]-1, (double)vertexspeicher1[X], (double)(vertexspeicher2[X]-vertexspeicher1[X])/(double)(vertexspeicher2[Y]-vertexspeicher1[Y]), (double)vertexspeicher1Z, (double)(vertexspeicher2Z-vertexspeicher1Z)/(double)(vertexspeicher2[Y]-vertexspeicher1[Y]));** speichert man es richtig ab.

Wenn die if Abfrage false ist, tauscht man und macht folglich **edgelist[vertexspeicher2[Y]].insert(vertexspeicher1[Y]-1, (double)vertexspeicher2[X], (double)(vertexspeicher1[X]-vertexspeicher2[X])/(double)(vertexspeicher1[Y]-vertexspeicher2[Y]), (double)vertexspeicher2Z, (double)(vertexspeicher1Z-vertexspeicher2Z)/(double)(vertexspeicher1[Y]-vertexspeicher2[Y]));**

Genau wie bei Beispiel 4 gibt man **ins erste Array und** die **erste double Variable** den **Inhalt des zweiten Arrays/ double Variable**.

fillScan (int scan, CG1Canvas canvas)

Zuerst legen wir **2 CG1Edge Variablen an**. In die erste speichern wir via **activelist.getHead()**; von activelist den ersten Eintrag und prüfen dann in einer while Schleife ob es ungleich null ist (**while (ka1 != null)**).

Anschließend laden wir in die **zweite CG1Edge** das darauffolgende Element von ka1, sprich **ka1.next;**

Nun kommt **Z** ins Spiel, legen dafür **eine dobule Variable** an und speichern darin **ka1.zIntersect;** (weiß leider nicht mehr genau was das ist).

Jetzt legen wir noch eine andere Variable für **Z** an in der wir den Anstieg speichern. Den rechnen wir mit **(ka1.zIntersect - ka2.zIntersect)/((double)(ka1.xIntersect-ka2.xIntersect));** aus

In einer **for Schleife** die mit **i = (int)ka1.xIntersect;** beginnt und so lange rennt bis **man i < (int)ka2.xIntersect;** hat, zeichnet man mit **canvas.setPixel (i, scan, illuminatinFarbe, z);** und erhöht z mit **z += zSchritt;**

Außerhalb der for Schleife darf man nicht vergessen noch **ka1 = ka2.next;** zu setzen damit man aufs nächste Element kommt.

CG1IlluminationModelDiffuse extends CG1IlluminationModel

Color doIllumination (CG1Vector normalVector, Color color)

Zuerst legt man eine Variable **mit float diffus;** für den Lichteinfallswinkel an und dann eine für das Kreuzprodukt des normalVector und lightVector, speichert mit **CG1Vector.dotProduct (normalVector, lightVector);** das Ergebnis in der Variable ab. Man berechnet hier den Winkel zwischen Oberflächennormale und der Richtung aus der das Licht kommt.

Nun überprüft man mit **if (punkt < 0)** ob das Ergebnis der Berechnung kleiner 0 ist (ist glaube ich der Fall wenn das Licht sehr schräg einfällt). Wenn ja, überspeichert man den Wert für **diffus** mit **0**, ansonsten mit dem Ergebnis des Kreuzproduktes das man zuvor ausgerechnet hat (also **diffus = (float)punkt;**) .

Nun legen wir eine Farbe für ambient und diffuse an, speichern jeweils darin die Farbe weiß mit Hilfe von **Color.white;**.

Anschließend müssen wir für ambient und diffusion den (Reflexions-) Koeffizienten berechnen.

Mit **float ambientmaterial[] = color.getRGBColorComponents (null);** bekommen wir ein Array mit dem Namen "ambientmaterial" in dem wir dann auf die einzelnen RGB Komponenten zugreifen können. Auf das erste greifen wir zum Beispiel mit ambientmaterial[0] *= ambientIntensity; und mit **ambientmaterial[0] *= ambientIntensity;** berechnen wir es für das 0te Element korrekt. Das machen wir auch für **[1]** und **[2]**. [0] steht dabei für rot, [1] für grün und [2] folglich für blau.

Dasselbe mache wir nun für Diffusion (**float diffusematerial[] = color.getRGBColorComponents (null);**) und führen dementsprechend (**diffusematerial[0] *= diffuseIntensity;**) für [0] und auch für **[1]** und **[2]** aus.

Anschließend müssen wir für jede RGB Komponente (also rot, grün und blau) Ambient und Diffusion berechnen. Dazu nimmt man den jeweiligen Anteil, dividiert ihn durch 255, multipliziert es anschließend mit dem jeweiligen Koeffizienten. Wenn man zum Beispiel für rot ambient berechnen will muss man **((float)ambient.getRed()/255) * ambientmaterial[0];** berechnen und speichern. Für diffusion schaut das sehr ähnlich aus, nur das man da nicht die ambient Werte sondern die von diffusion nimmt, sprich **float rotdiffus = ((float)diffuse.getRed()/255) * diffusematerial[0] * diffus;**.

Nachdem man das auch für Grün und Blau gemacht hat, muss man den „Ergebnisswert" berechnen, da sich ja die einzelnen Werte nach der „Beleuchtung" aus ambient und diffusion zusammensetzen. Man braucht sie nur zusammen zählen, also für rot **rotambambient + rotdiffus;**. Dementsprechend geht man auch für grün und blau vor.

Am Schluss muss man nur noch die Werte normieren (unter LINK gab es ein Post dazu). Da die Ergebnisse nur zwischen 0 (entspricht RGB Wert 0)und 1 (entspricht RGB Wert 255, also Maximum) sein können, schaut man also was kleiner 0 ist und setzt das auf 0 (**if (rot < 0.0) rot = 0.0f;**). Wenn etwas größer 1 ist muss man das ebenfalls „abfangen", ändern und setzt es auf die Maximalzahl, also 1 (**if (rot > 1.0) rot = 1.0f;**)

Am Schluss muss man nur noch die Ergebnisse zurück geben. Ich machte es in zwei Schritten, in dem ich es zuerst über **Color illuminatinFarbe = new Color (rot, gruen, blau);** einer Variable übergab und die dann via **return (illuminatinFarbe);** zurück gab. Natürlich könnte man dies auch alles in einem Schritt machen.

CG1Object

calcFaceNormals ()

Wie Beispiel 4, nur das man diese Mal statt von transVertices sich die Infos von vertexTable holt.

Man nimmt also **CG1Vector.diff (vertexTable[faceTable[i][2]], vertexTable[faceTable[i][3]]);** bzw. **CG1Vector.diff (vertexTable[faceTable[i][2]], vertexTable[faceTable[i][4]]);** und am Schluss rechnet

man sich wieder das Kreuzprodukt aus via **faceNormals[i] = CG1Vector.crossProduct (vertexspeicher1, vertexspeicher2);**.

Am Schluss muss man aber nicht die <0 Überprüfung machen (weiß leider nicht mehr warum genau man das nicht mehr braucht) und da wir mit unbearbeiteten Punkten arbeiten, verwenden wir **faceNormals[i].normalize ();** und Beispiel 5 ist somit fertig.

Beispiel 6

CG1GouraudShadedPolygon extends CG1ScanFilledPolygon

buildEdgeList ()

TODO

fillScan (int scan, CG1Canvas canvas)

TODO

clipPoint (CG1Point point, int num, int edge)

TODO

closeClip ()

TODO

clip (CG1Canvas canvas)

TODO

CG1IlluminationModelPhong extends CG1IlluminationModel

Color doIllumination (CG1Vector normalVector, Color color)

Nun wollen wir die Illumination machen. Dafür legen wir 2 Color, ich nannte sie **Color illuminationAmbient = Color.*WHITE*;** für Ambient und Color **illuminationLight = Color.*WHITE*;** fürs licht. Wie man sieht wurde es gleich mit „weiß“ gefüllt da in der Angabe von ToDo steht das man weiß verwenden soll.

Dann brauchen wir noch 2 floats, die ich **mit float diffuseLight, specularLight;** machte und ein **double für das Kreuzprodukt**.

Da es ja 3 Farbanteile gibt (rot, grün, blau) und man am Schluss schauen muss das sie nicht größer als 1 sind (da man nur Sachen zwischen 0 und 1 anzeigen kann, und 1 deswegen Maximum ist), legte ich auch noch **3 floats für red, green und blue** an.

Anschließend rechnete ich mir das Kreuzprodukt aus mit **CG1Vector.*dotProduct*(normalVector, lightVector);** Ich schaute dann in einer **if Abfrage** ob **(Kreuzprodukt < 0)**und wenn das zutraf, setze ich **diffuseLight = 0; ansonsten diffuseLight = (float) Kreuzprodukt;**

Dann legte ich mir mit **CG1Vector way = CG1Vector.add(lightVector, viewVector);** einen Vektor an und addierte dort die beiden Vektoren.

Anschließend normalisierte ich sie via **way.normalize();**

Kreuzprodukt = CG1Vector.dotProduct(normalVector, way); rechnete dann das Kreuzprodukt aus

In einer if Abfrage wurde dann wie zuvor geschaut ob (Kreuzprodukt < 0) und wenn ja, **specularLight = 0;**, ansonsten **specularLight = (float) Math.pow(Kreuzprodukt, specularExponent);** gesetzt, wobei specularExponent schon in der Datei vorgegeben war.

Wie bei Color doIllumination bekommen wir mit **float ambient[] = color.getRGBColorComponents (null);** ein Array mit dem Namen "ambient" in dem wir dann auf die einzelnen RGB Komponenten zugreifen können. **Dasselbe machen wir für diffuse und specular**.

Auf das erste Element greifen wir wie bei Color doIllumination zum Beispiel mit ambientmaterial[0] zu und mit **ambientmaterial[0] *= ambientIntensity;** berechnen wir es für das 0te (also eigentlich erste) Element korrekt. Das machen wir auch für **[1]** und **[2]**. [0] steht dabei für rot, [1] für grün und [2] folglich für blau.

Dasselbe mache wir nun für Diffuse (**diffuse[0] *= diffuseIntensity;**) und führen dementsprechend (**diffuse[0] *= diffuseIntensity;**) und auch für **[1]** und **[2]** aus. Nicht vergessen es auch für Specular zu machen, also **specular[0] *= specularIntensity;** und auch für **[1]** und **[2]**.

Anschließend müssen wir wie bei Color doIllumination für jede RGB Komponente (also rot, grün und blau) berechnen. Die Berechnung erfolgt ähnlich wie dort, aber nicht gleich! Wir müssen zum Beispiel diffuse[0]*diffuseLight rechnnen, dann specular[0]*specularLight dazu addieren, das alles Mal illuminationLight.getRed()/255) nehmen und die ganze Rechnung dann plus dem Rechnen wie man es bei Color doIllumination machte. Sprich man rechnet zum Beispiel für **red = ((((float)illuminationAmbient.getRed()/255) * ambient[0]) + (((float)illuminationLight.getRed()/255) * (diffuse[0]*diffuseLight + specular[0]*specularLight)));**
Dasselbe macht man für die anderen Farbkomponenten auch.

Am Schluss schaut man wieder ob die einzelnen Farbkomponenten nicht größer als 1 sind und **wenn doch**, setzt man sie auf die maximale Größe, **also 1**.

Am Schluss verwendete ich illuminatedColor, welches in der Datei (kurz vor der ToDo Anweisung) schon angelegt war, speicherte dort die Farbkomponenten ab mit **illuminatedColor = new Color (red, green, blue);** und gab es mit Hilfe von **return (illuminatedColor);** zurück.

CG1PhongShadedPolygon extends CG1GouraudShadedPolygon

buildEdgeList ()

Da das schon die dritte BuildEdgeList() ist, werde ich mich eher kurzfassen.

Wir legen wie zuvor **2 Arrays an mit [2]** (nannte sie vec1 und vec2), und dann wieder **2 double Variablen für Z**.
Diese Mal brauchen wir aber auch **3 double Arrays** die ich **normal1**, **normal2** und **normalD** nannte.

Dann speicherte ich in vec 1 mit **vec1[*X*] = clipped[numVertex - 1][*X*];** die Werte, genauso für **[Y]**. Für Z mit **vec1Z = depth[numVertex - 1];**.

Im ersten double Array, also normal1 sagte ich zuerst das ich 3 Elemente geben möchte (**normal1 = new double[3];**) und mit **normal1[*X*] = clippedNormals[numVertex-1].x;** speicherte ich sie, genauso wie für **[Y]** und **[Z]**.

Nun durchläuft man in einer **for Schleife** alle Punkte (**i<numVertex;**) (wie beim Beispiel zuvor).

Dort speichert man dann ins zweite Array bzw. zweite double Variable die Werte an der aktuellen Stelle ab (für **[X] clipped[i][X];**, für **[Y]** folglich **clipped[i][Y];** und für **Z depth[i];**).

Jetzt wird es wieder anders, da wir noch normal2 (wie schon kurz davor bei normal1) mit **normal2 = new double[3];** sagen müssen das wir 3 Elemente haben wollen, dann mit **normal2[X] = clippedNormals[i].x;** uns das [X] Element speichern, genauso wie wir das mit **[Y]** und **[Z]** machen.

Anschließend überprüft man wieder ob der Inhalt ungleich ist via **if (vec1 [Y] != vec2[Y])**.

Jetzt kommt normalD ins Spiel (was bei den Beispielen davor ja nicht vorkam!). Wie bei normal1 und normal2 sagen wir **mit normalD = new double[3];** das wir 3 Elemente haben werden und mit **normalD[X] = (normal2[X] - normal1[X])/(vec2[Y] - vec1[Y]);** speichern wir das erste ab. Das gleiche folgt wieder mit **[Y]** und **[Z]**.

Wie bei Beispiel 4 und 5 schaut man wieder in welche Richtung es geht (**if (vec1[Y] < vec2[Y])**) und mit **edgelist[vec1[Y]].insert(vec2[Y]-1, (double)vec1[X], (double)(vec2[X]-vec1[X])/(double)(vec2[Y]-vec1[Y]), vec1Z, (vec2Z-vec1Z)/(vec2[Y]-vec1[Y]), normal1, normalD);** speichert man es richtig ab.

Wenn die if Abfrage false ist, tauscht man und macht folglich vec1 und vec2 und hat dann **edgelist[vec2[Y]].insert(vec1[Y]-1, (double)vec2[X], (double)(vec1[X]-vec2[X])/(double)(vec1[Y]-vec2[Y]), vec2Z, (vec1Z-vec2Z)/(vec1[Y]-vec2[Y]), normal2, normalD);**

Genau wie bei Beispiel 4 und 5 gibt man **ins erste Array und** die **erste double Variable** den **Inhalt des zweiten Arrays/ double Variable**. Nicht vergessen darf man jetzt, da man ja auch ein **vec1Z hat**, **dort vec2Z hinein zu geben**, genauso wie in **normal1 für [X]**, **[Y] und [Z] die Werte aus normal2 zu übergeben**.

fillScan (int scan, CG1Canvas canvas)

Sehr ähnlich dem fillScan von CG1FlatShadedPolygon.

Zuerst legen wir **2 CG1Edge Variablen an**. In die erste speichern wir via **activelist.getHead()**; von activelist den ersten Eintrag und prüfen dann in einer while Schleife ob es ungleich null ist (**while (p1 != null)**).

Anschließend laden wir in die **zweite CG1Edge** das darauffolgende Element von p1, sprich **p1.next;**

Nun kommt **Z** ins Spiel, legen dafür **eine double Variable** an und speichern darin **p1.zIntersect;** (weiß leider nicht mehr genau was das ist).

Jetzt legen wir noch eine andere Variable für **Z** an in der wir den Anstieg speichern (zstep). Den rechnen wir mit **(p1.zIntersect - p2.zIntersect)/((double)(p1.xIntersect - p2.xIntersect)** aus.

Nun wird es anders (bis hierher war es so wie bei CG1FlatShadedPolygon)! Wir legen ein double Array für n an und für nstep, also **double[] n = new double[3];** und **double[] nstep = new double[3];**

Mit **n[X] = p1.cIntersect[X];** speichert man für X es ins Array ab. Dasselbe macht man auch für **[Y]** und **[Z]**.

In nstep speichern wir jetzt die Steigung ab die wir mit **nstep[X] = (p1.cIntersect[X] - p2.cIntersect[X])/(double)(p1.xIntersect - p2.xIntersect);** ausrechnen und gleich in nstep[X] speichern. Für **[Y]** und **[Z]** muss man es auch machen.

In einer **for Schleife** die mit **i = (int)p1.xIntersect**; beginnt und so lange rennt bis man **i < (int)p2.xIntersect;** hat (also so wie beim Beispiel davor), legt man zuerst ein Mal **CG1Vector normalVector;** und Color illuminatedColor; an. Danach speichert man mit **normalVector = new CG1Vector (n[X], n[Y], n[Z]);** die 3 Koordinaten in normalVector ab, normalisiert sie mit **normalVector.normalize ();**

Nun müssen wir mit **illuminatedColor = illuminationModel.doIllumination (normalVector, color);** alles in illuminatedColor geben damit wir dann, wie beim Beispiel davor, mit **canvas.setPixel (i, scan, illuminatedColor, z);** zeichnen können. Genauso erhöhen wir mit **z += zSchritt;**

Zu beachten ist, da wir jetzt ja auch ein n Array haben im Gegensatz zum Beispiel davor, und dieses auch zu erhöhen. Also **n[X] += nstep[X];** bzw. dasselbe für **[Y]** und **[Z]**.

Außerhalb der for Schleife darf man nicht vergessen noch **p1 = p2.next;** zu setzen (genauso wie beim Beispiel davor) damit man aufs nächste Element kommt.

Praktisches

Backface Culling

Was ist Backface Culling? Wozu ist das gut?

Bei Backface Culling werden nicht sichtbare Dreiecke entfernt und dadurch wird die Darstellungsgeschwindigkeit erhöht. Bei einem Dreieck ist eine der zwei Seiten durch den Normalvektor als Vorderseite definiert.

Gibt es in der Szene nur geschlossene, massive Objekte, so blickt der Betrachter immer auf solche Vorderseiten. Es ist somit überflüssig die Rückseiten („Backface“) zu zeichnen (da diese ja von anderen Dreiecken verdeckt werden).

Ist das Ergebnis > 0, so ist die Fläche (von vorne) zu sehen, bei einem Ergebnis < 0 nicht (bzw. nur von hinten und wird somit auf jeden Fall von anderen Objekten verdeckt).

Wie geht das (programmiertechnisch)?

dot Product aus Normalenvektor und View Vector machen.

Was ist das dotprodukt?

(das braucht man ja zum Beispiel bei N*L). Der cosinus des aufgespannten winkels (falls die vektoren einheitsvektoren waren). Das is wichtig, weil es so ein maß für den winkel ist. stehen sie

normalaufeinenader (d.h. es wird gaaanz dunkel) ==> N*L = cos(alpha) = 0. Sind sie ident ==> ... = 1 (sehr helle spec. reflection!)

Aus meiner letzten Ausarbeitung: Das **Dot Product**, auf Deutsch **Skalarprodukt** (auch inneres Produkt) genannt, wird oft verwendet um **Winkel zwischen zwei Vektoren und die Länge von Vektoren zu bestimmen**. Man berechnet es durch komponentenweises Multiplizieren der Koordinaten der Vektoren und anschließendes Aufsummieren.

$$\begin{pmatrix}1\\2\\3\end{pmatrix} \cdot \begin{pmatrix}-7\\8\\9\end{pmatrix} = 1 \cdot (-7) + 2 \cdot 8 + 3 \cdot 9 = 36$$

Cross Product, auf Deutsch **Kreuzprodukt** (auch äußeres Produkt) genannt, entspricht einem **Vektor, der senkrecht auf der von den beiden Vektoren aufgespannten Ebene steht.** Die Länge dieses Vektors entspricht dem Betrage nach der Fläche des Parallelogramms mit den Seiten $\vec{a}$ und $\vec{b}$.

$$\begin{pmatrix}1\\2\\3\end{pmatrix} \times \begin{pmatrix}-7\\8\\9\end{pmatrix} = \begin{pmatrix}2 \cdot 9 - 3 \cdot 8\\3 \cdot (-7) - 1 \cdot 9\\1 \cdot 8 - 2 \cdot (-7)\end{pmatrix} = \begin{pmatrix}-6\\-30\\22\end{pmatrix}$$

Wieso ist im 4er bsp. der fehler im teapot, dass man neben dem deckel ins nix hineinsehen kann?

Weil die unteren Ebenen ausgeblendet werden und der z-Buffer noch nicht arbeitet (wird erst in Bsp. 5 implementiert).

Wieso gibt es einige "fehlerhafte" Artefakte?

- Durch das backfaceculling werden nur Faces angezeigt, dessen Normalvektor in gegenentsetzte Richtung des Betrachters blickt, was bei uns ein negativer Z-Wert des Normalvektors wäre, da wir ja in die negative Z-Achse blicken. Bei offenen Objekten sieht man deswegen den Boden nicht (bei „teapot“) da es als Backface erkannt (es blickt weg von uns, aber sollte sichtbar sein da wir ja durch die offene Stelle ins Gefäß schauen) und entfernt wird.
- Bei durchsichtigen Objekten gibt es Probleme (kommt bei uns nicht vor).
- Wenn ein Objekt vor einen Spiegel stehen würde, dann würde im Spiegelbild es Fehler geben(kommt bei uns nicht vor).
- 2D Objekte machen ebenfalls Probleme, da bei 2D Objekten nicht zwischen Vorder und Hinterseite unterschieden werden kann.

In welcher Datei wird Backface-Culling angewendet?

In CG1Object.

Scanline Fill

Was ist Scan filling und wie funktioniert der Scanfill Algorithmus ungefähr?

Zuerst werden alle Eckpunkte in edgelist gespeichert und in die active list kommen dann die aktuellen eckpunkte der kanten, durch die die Scanline geht.

Warum ist es schlecht wenn eine Scanline durch einen (Eck) Punkt geht? Warum y-1 ? Was ist bei konvexen Polygonen das Problem?

Antwort 1: In den Folien und im Buch gibt es eine Zeichnung für diesen Fall (siehe nächste Frage!). Es kann dann vorkommen, dass das Polygon falsch gefüllt wird. Das was außen ist würde dann gefüllt werden und das Innere nicht. Gefüllt wird immer vom 1. Schnittpunkt zum 2. Schnittpunkt, dann nicht bis zum 3. Schnittpunkt. Von da wird dann wieder gefüllt bis zum 4. usw.

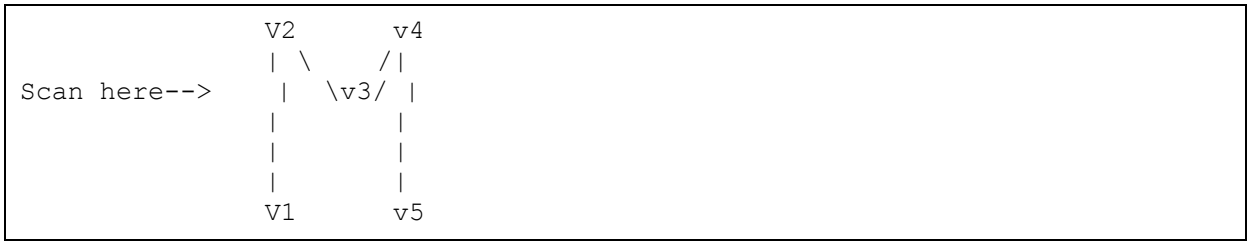
Bei uns im Code wird das durch yUpper -1 realisiert. Das yPrev oder die Methode yNext dient nur dazu festzustellen, ob benachbarte Kanten monoton steigend oder fallend sind, also ob ich überhaupt was abziehen muss.

Antwort 2: Der Klassiker mit der Scanline durch den Punkt. - Kommt bei uns nicht vor, da der Algorithmus den Punkt direkt darüber nimmt.

Na ja, du hast ja immer das Inside-Outside Problem. Wenn du die erste Kante schneidest, wird gezeichnet (inside), dann schneidest du die nächste Kante (outside) und es wird nicht mehr gezeichnet. Wenn die Scanline jetzt direkt durch den Punkt geht, kann es vorkommen, dass er nur als eine Kante erkannt wird. Daher verschiebst du den Punkt etwas um wieder zwei Kanten zu haben, sichtbar ist das (so gut wie) gar nicht.

Antwort 3: Erstmal zur Musterlösung: Dort hat man es sich einfach gemacht und einfach überall y-1 gerechnet. Damit spart man sich den Aufwand herauszufinden, ob es jetzt monoton steigend oder fallend ist, dafür nimmt man aber einen kleinen Fehler in Kauf, da auch dort abgezogen wird, wo es eigentlich nicht notwendig ist.

Scanline fill anhand eines beliebigen Polygons erklären/ Beispiel am Papier

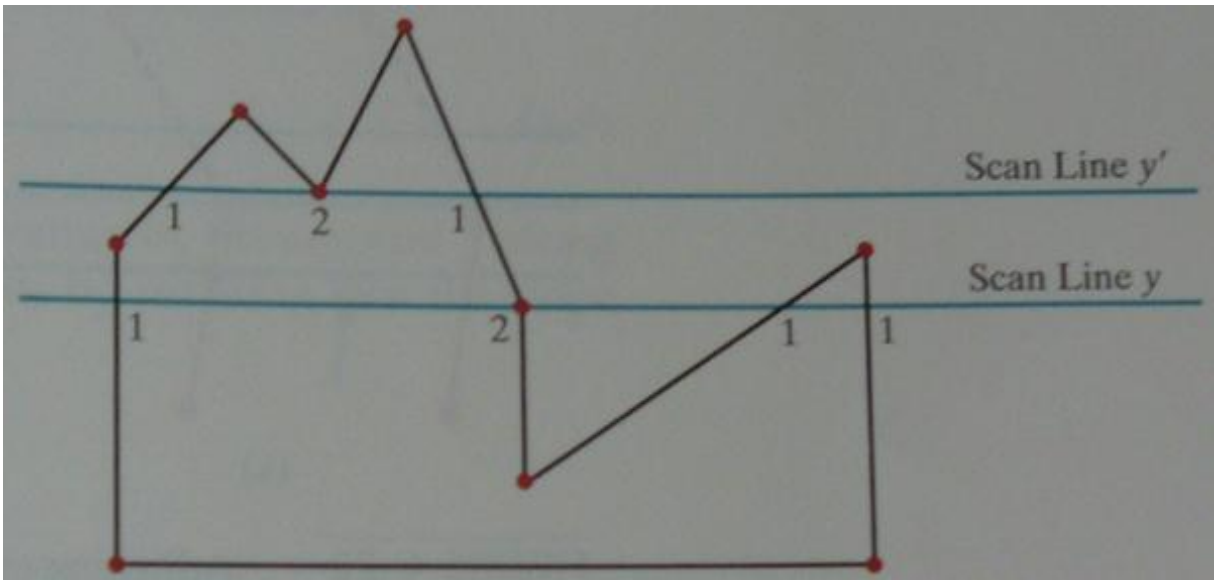


Scanline geht also genau durch v3, wobei der punkt v3 2x gezählt wird. Also ganze Scanline wird von erster Kante(links) bis v3 gefüllt (wobei v3 erstes Mal gezählt wird), dann wird wieder von v3 (zweites Mal gezählt) weiter gezeichnet bis zur Kante rechts und es gibt keine Probleme. Zu Problemen kommt es nur, wenn 2 Linien in dieselbe Richtung gehen.

Das wichtige ist, dass immer der kleine Y Wert in die edgelist gespeichert wird (also wo das insert passiert, daher die abfrage ob die Linie nach oben oder unten geht, der Eintrag passiert immer beim kleinen wert, also bei v2 wären das v1 und v3).

Allerdings kommt es bei v2 zu einem kleinen darstellungsfehler, der auch im Kommentar der Musterlösung erwähnt ist. Denn es wird immer nur v2[Y]-1 gespeichert (beim eintrag von v1 und v3), v2[Y] wird daher nicht ausgefüllt, es fehlt also ein Pixel.

Aus dem CG1 Buch Seite 197 Grafik 4-21:



Hier erkennt man gut, dass bei y es 1 + 2 + 1 + 1 Eckpunkte sind. Zusammen ergibt es 5, sprich man hat Probleme beim zeichnen. Er würde vom ersten Punkt zum Doppelpunkt zeichnen. Vor dort dann weiter zeichnen zum nächsten Punkt (zum 1er), und dann zum letzten Punkt (1er)auslassen und von dort bis zur Canvasgrenze zeichnen.

Oben bei y‘ hingegen hat man 1 + 2 +1 => 4 und hat keine Probleme beim zeichnen. Er würde von 1 bis Doppelpunk (erstes Mal gezählt) zeichnen, und von dort (zweites Mal gezählt) dann wieder zum 1ser nach rechts weiter zeichnen.

Z- Buffer

Was ist ein Z- Buffer? Wofür ist er gut?

Löst das Sichtbarkeitsproblem indem man für jeden Pixel neben der Farbe auch den Z- Wert speichert. Dann geht man Pixelweiße das Canvas durch, schaut welche Pixel dort gezeichnet werden sollen, und schaut auf Grund des Z- Wertes nach, welcher Pixel vorne liegt. Die Pixel die hinter dem zu zeichnenden Pixel liegen (was man aufgrund des Z- Wertes ja leicht feststellen kann), braucht man ja nicht zeichnen und speichert somit nur den Pixel und die Pixelfarbe vom Pixel der ganz vorne liegt.

Am Schluss zeichnet man dann nur die Pixel mit der entsprechenden Farbe die im Z- Buffer (Array) drinnen sind.

Wie funktioniert er?

(hab eine skizze gemacht und das mit den 2 arrays erklärt). wo ist er implementiert --> CG1Canvas. dann hat sie die CG1Canvas aufgemacht und ich hab den zBuffer herzeigen müssen (was wegen dem TODO 5: depthBuffer net so schwer zu finden war. und eben erklären was der code macht.

Wie werden die Z Werte der pixel berechnet?

Meiner Meinung nach werden die Z- Werte übergeben bei setPixel (int x, int y, Color c, double z) und dann schaut man, ob es kleiner ist als der im Z- Buffer gespeicherte Wert an der Stelle. Wenn ja, liegt es dahinter und man behält den alten Pixel. Wenn der übergebene Z- Wert aber größer ist, liegt es vor dem im Z- Buffer gespeicherten Wert und wird somit an der Stelle des „alten“ Z- Wertes gespeichert.

Was passiert, wenn der z Wert eines Pixels kleiner als der vorhandene ist?

Nicht zeichnen da er dahinter liegt=> kein Buffer Update.

Wie groß ist das Z- Buffer Array?

Das Array muss so groß wie das Canvas sein, also width*height. An sich sollte das ja logisch sein, da man beim Z-Buffer ja Pixel für Pixel vorgeht, somit jedes Pixel des Canvas untersucht (Was liegt vorne, muss gezeichnet werden, was liegt dahinter und kann folglich weggelassen werden?).

Wie schaut die Dimension des Z- Buffer Arrays aus und welchen Datentyp hat er?

Tutorin meinte bei der Abgabe das es ein Dimensional ist und double ist.

Mit was wird am Anfang das Z- Buffer Array gefüllt?

Mit negativen, unendlichen Zahlen.

faceNormals

Wo berechne ich die faceNormals?

In CG1Object. Braucht man für Berechnung der IlluminationColor in FlatShadedPolygon.
Skalarprodukt $\rightarrow_{xy} x \rightarrow_{xz}$ von vertexTable (unbearbeitete/ nicht transformierte Punkte + .normalzie());

Warum wird normalisiert?

Tutorin meinte bei der Abgabe, dass der Wert zwischen 0 und 1 sein muss, da damit dann ja noch weiter gerechnet wird und man dann Probleme hat.

Wozu berechne ich eigentlich die faceNormals?

Beim Flat Shading braucht man den Normalvektor jeder Fläche, um die beleuchtete Flächenfarbe zu berechnen.

```
public CG1FlatShadedPolygon (CG1Point[] _vertices, CG1Vector _faceNormal, Color _color,
CG1IlluminationModel _illuminationModel)
{
        super (_vertices, _color);
        illuminatedColor = _illuminationModel.doIllumination (_faceNormal, _color);
}
```

Illumination Model

Woraus setzt sich die diffuse illumination zusammen?

diffuses + ambiente (Umgebungslicht) Beleuchtung eines Objektes berechnet
=> I = Reflexionskoeffizient k2 * Lichtquellenintensität I, *N * L

Erklären/ berschreiben der phong illumination

diffusses + ambientes + specular (Glanzpunkte) Beleuchtungsmodell

Was ist specular und wie funktioniert es?

TODO

Lichtmodelle/ Beleuchtungsmodel (diffuse und phong)

Diffuse: diffuses + ambiente (Umgebungslicht) Beleuchtung eines Objektes berechnet
Phong: diffuses + ambiente **+ specular** Beleuchtungsmodell (+ Glanzpunkte)

Phong Illumination Model: Worin liegt der Hauptsächliche Unterschied?

Es gibt auch einen Specular Highlight (spiegeln).

Formeln für diffuse beleuchtung und phong beleuchtung

Diffuse: $\mathbf{I = k_d \cdot I_l \cdot \cos\theta = k_d \cdot I_l \cdot N \cdot L}$ [N·L ist skalares Produkt]
Phong: $\mathbf{I_{l,spec} = k_s \cdot I_l \cdot \cos_n\varphi = k_s \cdot I_l \cdot (R \cdot V)_n}$.

Was ist eine perfekte spekulare Reflektion?

Da kommt es drauf an wie hoch der Exponent bei der cos alpha Funktion ist. Ist dieser höher ist auch der Glanzpunkt kleiner und dadurch wirkt das objekt glänzender und wenn nun dieser Exponent ins unendliche tendieren würde dann wäre dies eine perfekte spekular-Reflektion.

Reflexionsvektor bei der phong illumination, wie berechnet?

R = (2N*L)N-L

Wie wird das Licht beim phong illumination model berechnet?

TODO

Was macht der Exponent bei der Phong Illumination?

Exponent (ist in unserem CG1Main Framwork das "E" ganz rechts unten): Der Exponent bei der Phong Illumination verstärkt den Glanzwert. Da man das Skalarprodukt vom Halfway Vektor und vom Normalvektor berechnet und das Ergebnis der Cosinus vom eingeschlossenen Winkel ist. Wenn man den Cosinus Wert potenziert, wird der Gipfel dünner und somit die Streuung des Lichtes geringer. Somit steuert der Exponent die Streuung der Lichtquelle aufs Material. --> Specular Light ist konzentrierter und schaut daher glänzender aus.

Was ist der Unterschied zwischen Phong SHADING und Phong ILLUMINATION MODEL?

Das illumination model ist das mit ambient, diffuse und specular light und das shading ist ja so ähnlich wie der goraud algorithmus nur komplizierter und rechen intensiver.....das heißt das eine ist ein Beleuchtungsverfahren und das andere ein Schattierungsverfahren.

Shading

Was passiert beim Flat Shading?

Berechnung des Normalenvektors mit Crossproduct, Farbe für das ganze Polygon anhand vom Normalvektor ermitteln

Wie entsteht der Lichtwert einer polygonfläche beim Flat shading?

Beim Flat Shading ist die Intensität (und damit der Farbwert) für jede Oberfläche konstant und hängt vom Winkel des Normalvektors der Oberfläche mit dem Lichtvektor ab. Je flacher das Licht auf die Fläche trifft, desto weniger Intensiv erscheint es.

Im Diffuse Illumination Model berechnet man den Farbwert der an der Stelle des Normalvektors steht durch den Ambienten Lichtanteil (Umgebungslicht) durch Materialkonstante* Intensität -> die in Color enthalten ist. Dazu kommt der diffuse Anteil, der wie gesagt vom Einfallswinkel abhängt.

Also Konstante*Intensität*cos(phi) wobei cos(phi) der Winkel ist der vom Lichtvektor und dem Normalvektor aufgespannt wird. Man bekommt diesen auch durch das innere Produkt beider Vektoren. Also ergibt sich insgesamt:

Farbwert = Materialkonstante*Intensität + Materialkonstante*Intansität*Lichtvektor_DOT_Normalvektor.

Dieser ist in jeder Surface Konstant und wird im Scan Fill einfach zwischen den Schnittpunkten konstant verwendet.

Woher kommt der Farbwert im Flat Shading

Aus dem Mittel der Farbwerte der Eckpunkte.

Wie errechne ich den Normalvektor beim Flatshading

Gegeben sind die VertexNormals, also die Normalvektoren der Eckpunkte. die normals auf die flaeche errechnet man sich einfach indem man 2 vektoren des polygons bildet und dann das Kreuzprodukt der vektoren berechnet.

Gouraud Shading erklären (Was ist es, wie funktioniert es?)

Beim Gouraud Shading wird die Beleuchtung auf die Vertices des Polygons angewendet, die Farbwerte der einzelnen Pixel jedoch aus den Farbwerten der Vertices interpoliert.

Am Papier die Berechnung von Interpolationswerten beim Gouraud Shading erklären.

TODO

Im Code zeigen wo Interpolation stattfindet.

TODO

Phong Shading erklären. Was ist es, wie funktioniert es?

Phong Shading interpoliert für jeden Pixel aus den Normalen der Vertices eine interpolierte Normale des Pixels und wendet das Beleuchtungsmodell für jeden Pixel mit einer neuen Normale an.

Farbwerte woher nehmen?

TODO

Was müsste mit dem Modell passieren oder welche Eigenschaft müsste das Modell aufweisen damit bei Gouroudshading dasselbe Resultat herauskäme wie bei Phongshading?

Das Modell müsste so detailiert modelliert werden, sprich so viele Polygone enthalten, dass sie nur noch als Pixel am Bildschirm angezeigt werden können und somit das gleiche Resultat bei Gouroud bieten würde als bei Phong.

Unterschied Gourad und Phong Shading (Vorteile und Nachteile)

Interpolation von Farben beim Gourad. Gourad ist schneller.

Normalenvektoren beim Phong. Schaut beim Phong Illumination Modell viel besser aus, da für jedes Pixel die Beleuchtung berechnet wird.

Wo spielt der Lichtvektor eine Rolle?

Bei beiden Modellen, da er den Diffusewert beeinflusst, der aber im Phong ebenfalls vorhanden ist

Wieso befindet sich auch abseits der Lichtquelle ein einigermaßen helles Objekt?

Ambient Light wird angewandt.

Wie sieht der Ausfallswinkel beim Diffusen Licht aus?

Beim Diffusen Licht wird in einem sogenannten Halbkreis (oder räumlich gesehen Hemisphäre heißt diese Form) das Licht ausgestrahlt.

Wie kann man das diffuse Licht einordnen?

Es wird gestreut.

Halfway Vektor

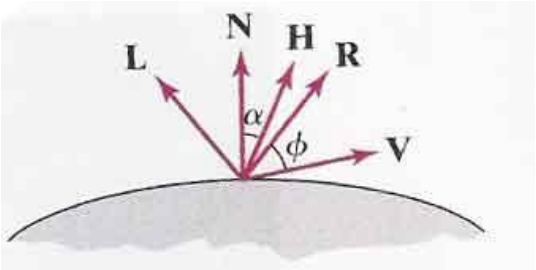


FIGURE 10-22 Halfway vector H along the bisector of the angle between L and V.

Gefragt war wie man sich den errechnet (view und lightVektor addieren und normalisieren) und warum man den nimmt und nicht den Vektor R (viel mühsamer zu errechnen, würde zu viel rechenzeit brauchen und da es sich ohnehin um ein empirisches modell handelt...)

Aus Buch: „A somewhat simplified Phong model is obtained using the halfway Vector H between L and V to calculate the range of specular reflections. If we replace V*R in the Phong model with the dot Product N*H, this simply replace the empirical cos ϕ calculation.

For nonplanar surfaces, N*H requires less computation than V*R because calculation of R at each surface point involve the variable vector N.“

Edgelist und Activelist

Was ist die activelist?

In die active list kommen die aktuellen Eckpunkte der Kanten, durch die die Scanline geht.

Wie funktioniert das verwalten der edgelist beim scanlineFilling

Erklären was in "drawClipped" bei Beispiel 4 passiert, wie der ganze Algorithmus der Reihe nach abläuft (vorher code genau anschauen). Ein paar Sonderfälle besprochen, was ist wenn sich zwei edges in einem punkt treffen, dass von oben nach unten eingetragen wird beachten...

Was musste bei dem Bsp 4 (z.B.) gemacht werden?

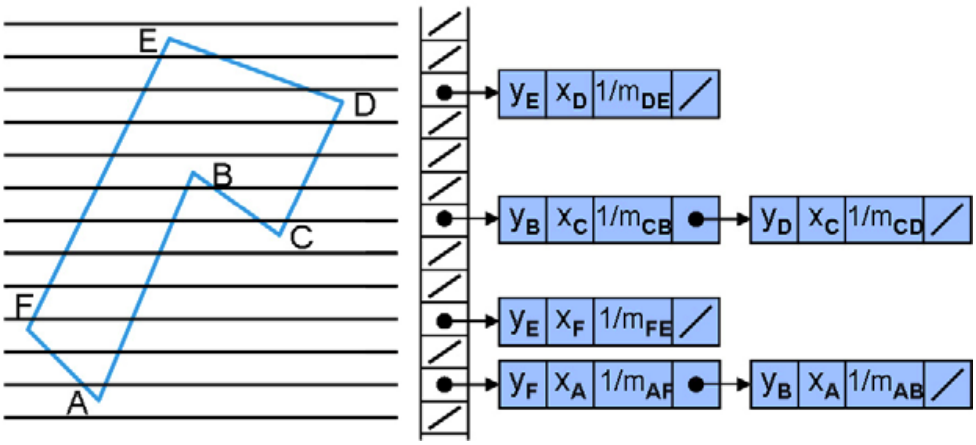
Ich musste noch im Code bei ScanFilledPolygon das speichern in die edgelist (in der for-Schleife) zeichnen und bei PhongShadedPolygon das erhöhen des Wertes der Normalen (auch in der for-Schleife in scanFill).

Aus PDFs von Werner Purgathofer

Graphikprim.-Attribute

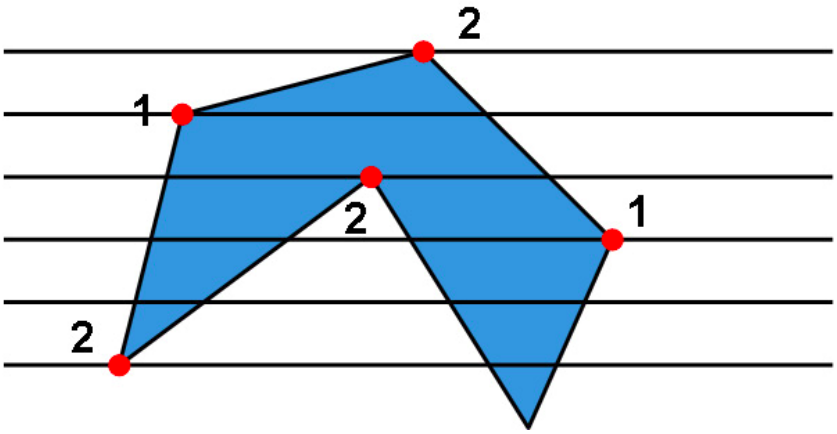
Scanlinien-Flächenfüllen

Alle Kanten des Polygons werden nach ihrem niedrigeren y-Wert sortiert (Bucketsort). Man speichert für jede Kante: [max. y-Wert, Anfangs-x-Wert, Steigung] Aus dieser Datenstruktur erzeugt man für jede Scanlinie von unten nach oben eine Liste der „aktiven“ Kanten, das sind jene, die die Scanlinie schneiden. Dann wird einfach die Scanlinie vom 1. zum 2. Schnittpunkt gefüllt, vom 3. zum 4., vom 5. zum 6. usw.



Die aktiven Kanten erzeugt man inkrementell. Am Beginn sind keine Kanten aktiv. Aus der sortierten Kantenliste sieht man für jeden y-Wert, ob dort eine neue Kante beginnt, diese wird zur aktiven Liste hinzugefügt. Gleichzeitig werden alle Kanten, deren maximales y überschritten wurde, entfernt. Die Liste selbst ist immer nach ihren Schnittpunkten von links nach rechts sortiert, sodass das Zeichnen unmitelbar erfolgen kann.

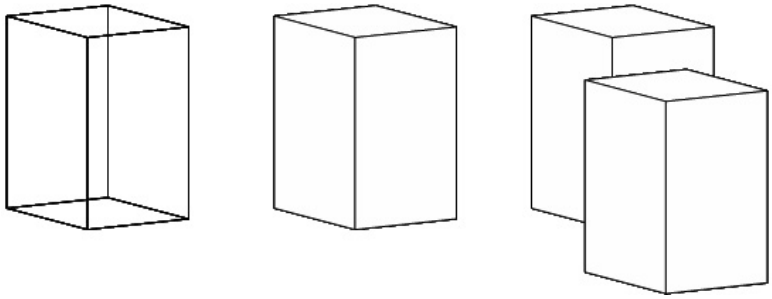
Die Schnittpunkte kann man ebenfalls inkrementell erzeugen. Ausgehend vom (exakten) Schnittpunkt (xk,yk) einer Scanlinie mit einer Kante erhält man den nächsthöheren Schnittpunkt (xk+1,yk+1) durch xk+1= xk+1/m und yk+1= yk+1. Man sieht nun, warum der Anstieg 1/m in den Knoten mitgespeichert wird.



Wenn ein Polygoneckpunkt genau auf einer Scanlinie zu liegen kommt, dann muss darauf geachtet werden, dass die Anzahl der Schnittpunkte an dieser Stelle korrekt ist. Manche Punkte müssen dann einfach gezählt werden, andere doppelt (siehe Abbildung). Um diesem Problem aus dem Weg zu gehen werden häufig die Punktkoordinaten um einen kleinen Wert ε (Anmerkung von mir: Epsilon) nach oben oder unter verschoben. Dieser Wert ist so klein, das man es nicht sieht, aber groß genug, dass der Punkt neben der Scanlinie liegt.

Sichtbarkeitsverfahren

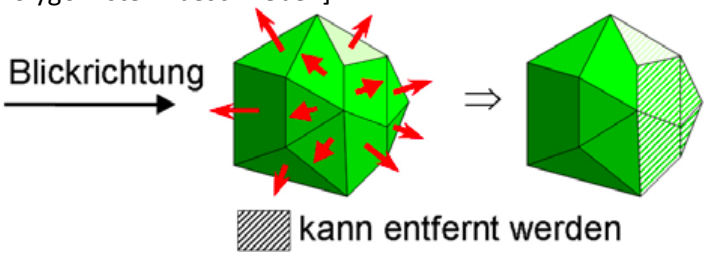
Um Szenen glaubwürdig und korrekt darzustellen, müssen alle Teile, die aus der Blickrichtung nicht sichtbar sind, weggelassen werden. Das sind insbesondere die Rückseiten von Objekten und Objektteile, die von anderen Objekten verdeckt werden. Man spricht von Hidden-Line- oder Hidden-Surface Eliminierung.



Abhängig von der Szenenkomplexität, den Objekttypen und –datenstrukturen, der verfügbaren Hardware und Anwendungsanforderungen verwendet man dazu verschiedene *Sichtbarkeitsverfahren*. Bei den Objektraum-Methoden wird die Lage der Objekte miteinander verglichen und nur vordere (sichtbare) Teile gezeichnet, bei den Bildraum-Methoden wird für jeden Bildteil getrennt berechnet, was dort sichtbar ist. Die folgenden Erklärungen erfolgen ohne Berücksichtigung von transparenten Objekten.

Backface Detection (Backface Culling)

Backface Culling ist ***kein*** vollständiges Sichtbarkeitsverfahren. Es werden lediglich alle Polygone, deren Oberflächennormale vom Betrachter wegzeigt und die daher ganz sicher nicht sichtbar sein können, eliminiert, um den Aufwand nachfolgender Arbeitsschritte zu reduzieren. Dadurch werden im Durchschnitt 50% der Polygone entfernt. Dies berechnet man entweder mit dem Skalarprodukt des Blickrichtungsvektors mit der Oberflächennormale ($V_{view} \cdot N > 0 \longrightarrow$ unsichtbar) oder durch Einsetzen des Blickpunktes (x,y,z) in die Ebenengleichung ($Ax + By + Cz + D < 0 \longrightarrow$ unsichtbar). [Annahmen wie bei „Polygonlisten“ beschrieben]

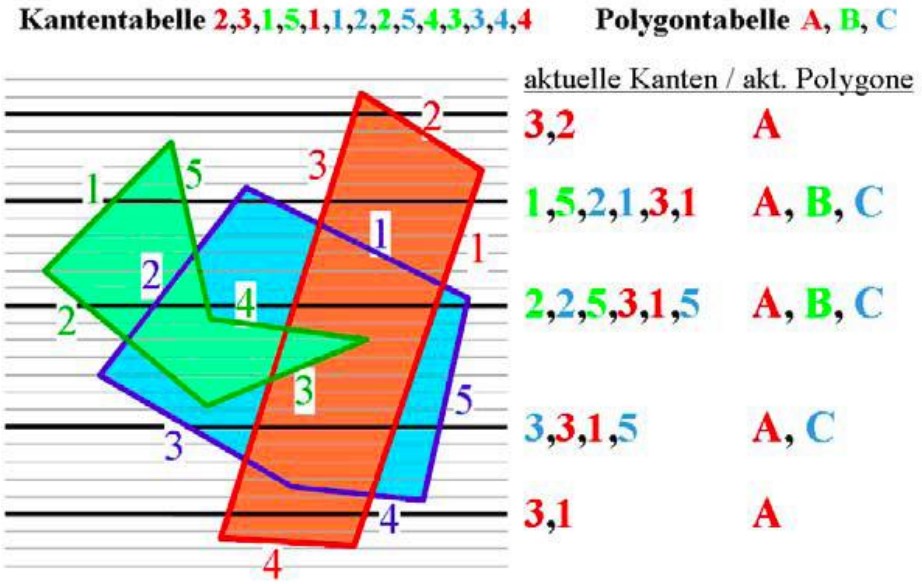


Z-Puffer-Verfahren (Depth Buffer)

Der z-Puffer-Algorithmus löst das Sichtbarkeitsproblem für eine bestimmte Bildauflösung folgendermaßen: Für jeden Bildpunkt merkt man sich zusätzlich zur Farbinformation in einem eigenen Speicher die Position des dargestellten Objektes. Da als Blickrichtung normalerweise die z Richtung verwendet wird, entsprechen x- und y-Werte dieser Position denen der Abbildungsebene und es braucht nur der z-Wert gespeichert zu werden. Man braucht also zusätzlich zum Bildpuffer (frame buffer) einen weiteren Speicherbereich, der für jeden Pixel einen Koordinatenwert (z-Wert) aufnehmen kann, diesen Speicher nennt man z-Puffer oder Tiefenpuffer (z-buffer, depth buffer). Nun kann man alle Objekte in beliebiger Reihenfolge zeichnen. Die z- Werte des nächsten zu zeichnenden Objektes (meist ein Polygon) werden berechnet und mit den z-Werten der Pixel verglichen, in die das Objekt gezeichnet werden soll. Ist der neue z-Wert näher zum Betrachter (also normalerweise größer), dann wird das Objekt an dieser Stelle über den alten Bildwert darüber gezeichnet und der z-Wert im z-Puffer ebenfalls ersetzt. Andernfalls ist das neue Objekt verdeckt und wird an dieser Stelle nicht gezeichnet: Für ebene Polygone lassen sich die z-Werte natürlich wieder inkrementell effizienter berechnen. **Der große Vorteil des z-Puffer-Verfahrens ist, dass die Objekte (Polygone) nicht sortiert werden brauchen**.

Scanline-Methode

Die korrekte Sichtbarkeit wird beim Scanline-Verfahren zeilenweise berechnet (im Beispiel von oben nach unten, also y fallend). Dabei nutzt man aus, dass sich zwei übereinander liegende Pixelreihen (Scanlines) in ihrem Sichtbarkeitsverhalten oft nur unwesentlich unterscheiden.



Ausgehend von einer Tabelle aller nach ihrem größten y-Wert sortierten Polygonkanten und einer zugehörigen Polygonliste wird für jede Scanline jeweils eine Liste der aktuellen Kanten erstellt. Dies passiert inkrementell aus den Kanten der letzten Scanline: Kanten, die geendet haben, werden eliminiert, und die nächsten Kanten der sortierten Tabelle werden überprüft, ob sie schon angefangen haben. Nachdem solcherart alle Schnittpunkte einer Scanline mit allen Kanten errechnet sind, werden diese nach x-Wert (von links nach rechts) sortiert. Zwischen je zwei Schnittpunkten muss nun noch eruiert werden, welches Polygon dort am nächsten zum Betrachter liegt, dieses ist sichtbar und wird entlang dieser einen Scanline gezeichnet.

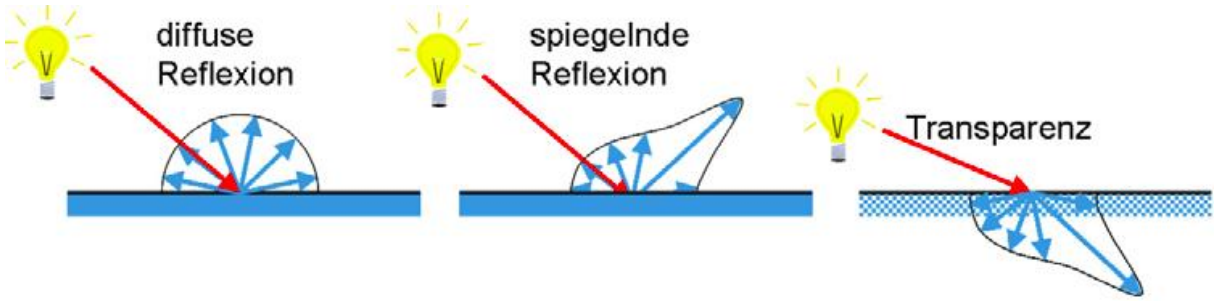
Licht und Schattierung

Ein Beleuchtungsmodell oder auch Schattierungsmodell (Illumination Model oder Lighting Model oder Shading Model) dient dazu, aus der Beschreibung der Lichtverhältnisse und den Oberflächeneigenschaften eines Objektes zu berechnen, welche Farbe/Helligkeit der Betrachter wahrnimmt, also welche Farbe das zugehörige Pixel erhalten soll. Zusammen mit der perspektivischen Projektion ist das der wichtigste Beitrag für das realistische Aussehen von Computergraphik-Bildern.

Alle folgenden Betrachtungen und Formeln beziehen sich der Einfachheit halber nur auf die Helligkeit der Beleuchtung. Um Farben zu behandeln, müssen diese Berechnungen in mehreren Wellenlängen durchgeführt werden, im einfachsten Fall in Rot, Grün und Blau.

Objektoberflächen

Oberflächen können das einfallende Licht entweder diffus reflektieren, d.h. es wird in jede Richtung gleich viel Licht reflektiert (z.B. Papier, Kreide), oder spiegelnd reflektieren, d.h. in die Spiegelungsrichtung wird mehr Licht reflektiert als in die anderen Richtungen (z.B. Lack, Metall), oder transparent sein, d.h. das Licht geht durch die Oberfläche durch und kommt auf der anderen Seite wieder heraus (z.B. Glas, Wasser). Reale Oberflächen besitzen meist eine Mischung aus diesen Eigenschaften. Weiters darf nicht übersehen werden, dass Licht nicht nur von den Lichtquellen auf Flächen auftrifft, sondern dass auch reflektiertes Licht von anderen Flächen mitspielt.



Ein einfaches Beleuchtungsmodell

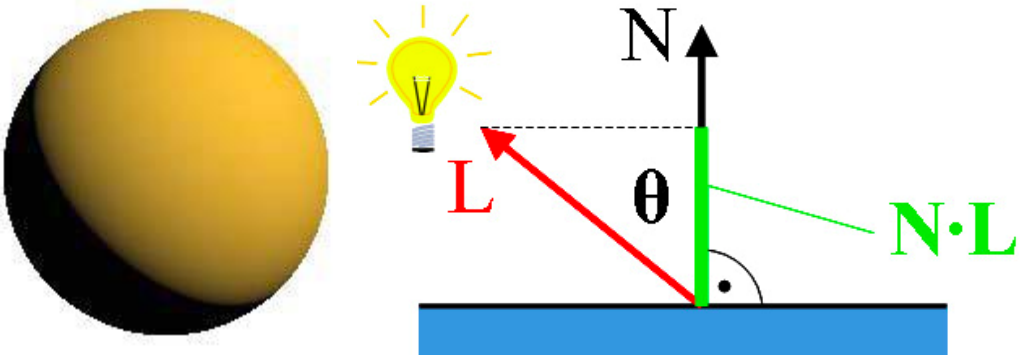
Die physikalisch exakte Simulierung von Licht und dessen Interaktion mit Objektoberflächen ist sehr komplex. Daher verwendet man in der Praxis vereinfachte, empirische Beleuchtungsmodelle. Diese sind etwa so aufgebaut.

Hintergrundlicht (ambientes Licht)

Da jedes Objekt einen Teil des auf ihn auftreffenden Lichtes auch wieder abstrahlt, ist es auch dort nicht ganz dunkel, wo keine Lichtquelle direkt hinleuchten kann. Dieses **überall vorhandene Basislicht** wird ambientes Licht genannt, auch Hintergrundlicht. Einfache Beleuchtungsmodelle inkludieren dazu einen **konstanten Wert I_a zu jeder Beleuchtungsberechnung**.

Lambert'sches Gesetz

Dieses Gesetz besagt, dass je flacher Licht auf eine Oberfläche auffällt, desto dünkler erscheint diese Oberfläche. Erst durch diesen Effekt erhalten wir den Eindruck einer räumlichen Form.



Sei **I_l die Helligkeit der relevanten Lichtquelle**, und sei **k_d der *diffuse Relexionskoeffizient*** der beleuchteten Oberfläche, der also angibt, **wieviel Prozent des einfallenden Lichtes in alle Richtungen gleichmäßig wieder abgestrahlt wird**. Natürlich gilt $0 \leq k_d \leq 1$. Weiters sei **θ der Winkel zwischen der Oberflächennormale und der Richtung zur Lichtquelle**, also der Lichteinfallsrichtung. Dann gilt für die resultierende Intensität I an der Oberflächenstelle:

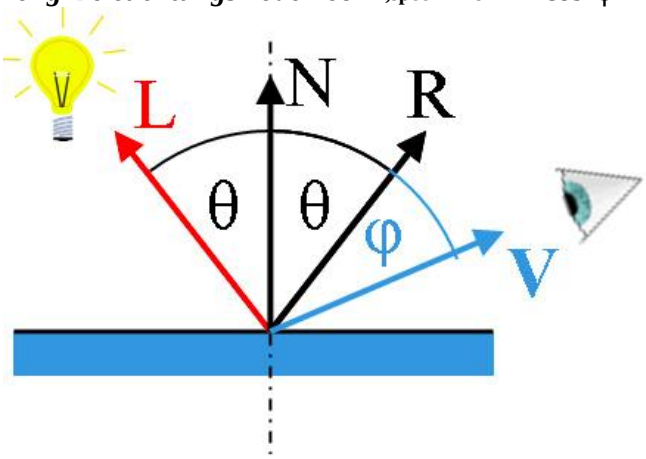
$\mathbf{I = k_d \cdot I_l \cdot \cos\theta = k_d \cdot I_l \cdot N \cdot L}$ [N·L ist skalares Produkt]

Wenn man nun auch noch das ambiente Licht hinzufügt, dann erhält man schon eine recht schöne Kugel (obere Kugel = nur diffuse Beleuchtung, untere Kugel = diffus + ambient).

Glanzpunkte (Specular Highlights)

Fast jede Oberfläche ist auch etwas spiegelnd. Wenn man diesen Aspekt nicht mitmodelliert, dann wirken alle Materialien gleich stumpf. Da die exakte spiegelnde Reflexion äußerst kompliziert zu berechnen ist, behilft man sich mit einer einfachen Funktion, die einen ähnlichen Verlauf hat wie das Highlight: $\cos_n$. Mit dem freien Parameter n lässt sich dabei die „Poliertheit" der Oberfläche steuern**: je größer n ist, desto kleiner wird der Glanzpunkt und desto glatter wirkt die Oberfläche** (linke Kugel), je kleiner das n ist, desto matter wirkt die Oberfläche (rechte Kugel).

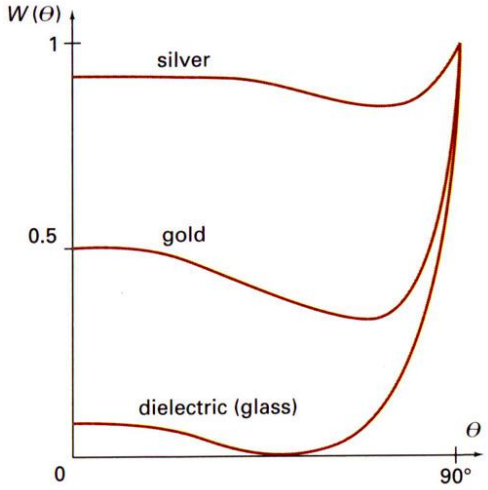
Um diesen Effekt im richtigen Ausmaß zur Beleuchtung hinzufügen zu können, wird noch ein weiterer Faktor eingeführt, der ***spiegelnde Reflexionskoeffizient*** k_s. Der Glanz berechnet sich dann nach diesem sogenannten ***Phong-Beleuchtungsmodell*** so: $I_{l,spec} = k_s \cdot I_l \cdot \cos_n\varphi = k_s \cdot I_l \cdot (R \cdot V)_n$.



Der Winkel ϕ ist dabei der Unterschied zwischen dem exakten Reflexionsstrahl und der Richtung zum Auge.

Etwas näher an der Wahrheit ist die Verwendung der Fresnel'schen Reflexionsgesetze, die beschreiben, dass der Spiegelungsgrad auch vom Lichteinfallswinkel abhängt, dass also der Koeffizient k_s eigentlich eine Funktion W(θ) der Lichteinfallsrichtung ist. Für die meisten Materialien ist dieser Wert aber fast konstant. Daher wird auf diesen Aufwand verzichtet, wenn man nicht gerade ein Material darstellen will, bei dem der Effekt auffällt. Das Bild rechts zeigt die Abhängigkeit dieser

Funktion W(θ) vom Winkel zwischen Lichteinfall und Normale auf die Oberfläche für drei verschiedene Materialien.



Bei der Berechnung von R muss man noch bedenken, dass es sich hier um Vektoren im 3D-Raum handelt, wo L, N und R in einer Ebene liegen müssen und alle Länge 1 haben sollen. R ergibt sich zu R = (2N·L)N – L Weil die Glanzfunktion sowieso nur eine grobe Näherung ist, verwendet man auch häufig eine einfache Formel, in der R·V durch N·H ersetzt wird. Der Winkel zwischen N und der Halbierenden H zwischen L und V ist ϕ sehr ähnlich. Wenn wir alle bisherigen Kompenenten zusammensetzen, erhalten wir ein

einfaches komplettes Beleuchtungsmodell: $\mathbf{I = k_a \cdot I_a + \Sigma_{l=1-n} (k_d \cdot I_l \cdot N \cdot L + k_s \cdot I_l \cdot (N \cdot H_l)_n)}$

Es gibt noch viele weitere Aspekte, die man berücksichtigen muss, um der Realität näher zu kommen, aber diese werden hier nicht näher beschrieben: Farbverschiebungen in Abhängigkeit der Blickrichtung, Einfluss der Entfernung der Lichtquelle, anisotrope Oberflächen und Lichtquellen, Transparenz, atmosphärische Effekte, Schatten, und so weiter.

Schattierung von Polygonen

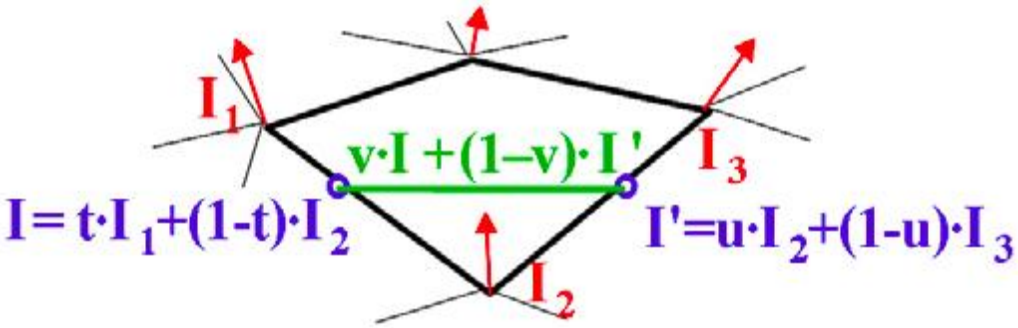
Flat-Shading

Beim Schattieren eines Polygons hat klarerweise jeder Punkt die gleichen Oberflächeneigenschaften, vor allem auch den gleichen Normalvektor. Beim einfachen Ausfüllen jedes Polygons mit einer Farbe werden die Grenzen zwischen den Polygonen deutlich störend erkennbar. Der sogenannte Mach-Band-Effekt, das ist ein kantenverstärkender Mechanismus des Auges, macht das Problem dabei noch ärger als es ist. Dieser Effekt lässt uns an Kanten die dunklere Seite dunkler wahrnehmen als sie ist, und die hellere Seite heller als sie ist. Die einfachste Lösung dieses Problems ist das Interpolieren der Schattierung zwischen den Polygonen. Dazu sind zwei Verfahren üblich: Gouraud-Schattierung und Phong-Schattierung.

Gouraud-Schattierung

Die Gouraud-Schattierung interpoliert die berechneten Helligkeitswerte über die Polygonflächen. Dazu werden an den Eckpunkten der Polygone Helligkeitswerte berechnet und von diesen aus wird durch lineare Interpolation jedes Polygon gefüllt. Konkret geht das so:

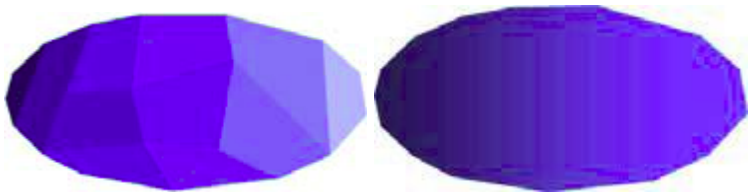
(1) An jedem Eckpunkt wird eine Normale als Mittelwert der Normalen aller angrenzenden Polygone berechnet. Das ist natürlich nur ein Näherungswert der Normale der echten zugrundeliegenden Fläche.



(2) Aus den Eigenschaften der Oberfläche, der Normale und der Lichteinfallsrichtung wird ein Helligkeitswert („Schattierung“) für jeden Eckpunkt berechnet. Man beachte, dass dadurch angrenzende Polygone an diesen Eckpunkten alle die gleichen Werte erhalten.

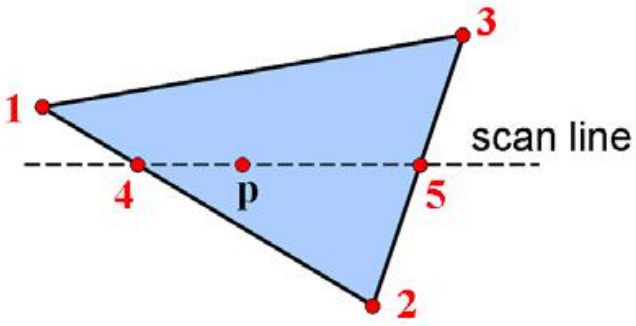
(3) Entlang der Polygonkanten werden die Helligkeitswerte linear interpoliert, d.h. es wird für jeden Schnittpunkt mit einer Scanline ein Wert ermittelt. Man beachte, dass dadurch für aneinander grenzende Polygone entlang der gemeinsamen Kante die gleichen Werte entstehen.

(4) Entlang jeder Scanline wird von der linken bis zur rechten Polygongrenze wieder linear interpoliert. Dadurch haben nebeneinander liegende Pixel immer eine sehr ähnliche Helligkeit und es kommt zu keinen sichtbaren Kanten.

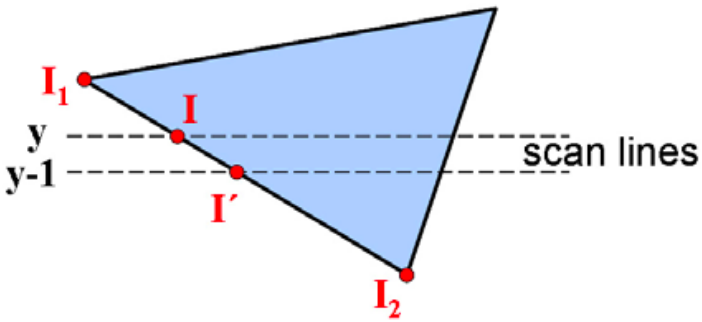


Dennoch verbleiben Fehlerquellen. So wird die **Silhouette** natürlich **nicht verändert**, dadurch verbleiben **störende Polygonkanten** sichtbar (siehe Bild (oben) rechts). Weiters kommt es im Bereich von **Glanzpunkten zu zufälligen Interpolationsergebnissen**, je nachdem ob es **zufällig eine Normale gibt, die genau einen Glanzpunkt erzeugt oder nicht**. Dies stört besonders bei bewegten Objekten.

Die (lineare!) *Interpolation der Intensitäten* kann natürlich wieder inkrementell erfolgen.



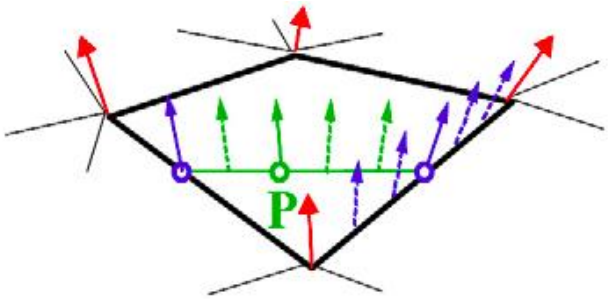
$$I_4 = \frac{y_4 - y_2}{y_1 - y_2} I_1 + \frac{y_1 - y_4}{y_1 - y_2} I_2 \qquad I_p = \frac{x_5 - x_p}{x_5 - x_4} I_4 + \frac{x_p - x_4}{x_5 - x_4} I_5$$



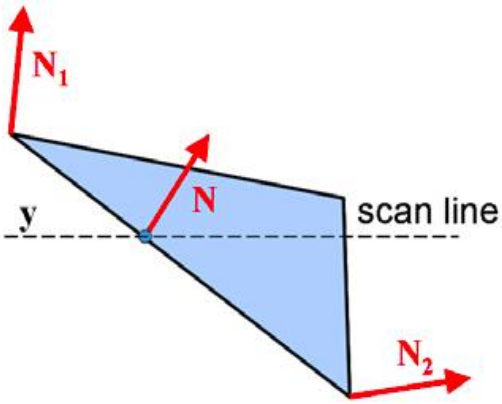
$$I = \frac{y - y_2}{y_1 - y_2} I_1 + \frac{y_1 - y}{y_1 - y_2} I_2 \qquad I' = I + \frac{I_2 - I_1}{y_1 - y_2}$$

Phong-Schattierung

Als Alternative zur Gouraud-Schattierung erzeugt die Phong- Schattierung (nicht mit dem Phong-Beleuchtungsmodell verwechseln!) viel konsistentere Glanzeffekte. Ebenso wie bei der Gouraud-Schattierung werden in den Polygoneckpunkten die Normalen berechnet, aber nun werden diese Normalen entlang der Polygonkanten interpoliert, und anschließend entlang der Scanlines. Dann wird für jedes Pixel extra die Helligkeit nach einen Beleuchtungsmodell berechnet. Dies verursacht zwar einen höheren Aufwand, führt aber auch zu schöneren Ergebnissen.



Normalvektorinterpolation: $\mathbf{N = N_1(y-y_2)/(y_1-y_2) + N_2\,(y_1-y)/(y_1-y_2)}$



ACHTUNG: Das Phong-Beleuchtungsmodell (auch Phong-Schattierungsmodell) und die Phong- Schattierung (auch Phong-Interpolation) sind zwei vollkommen unabhängige Dinge!

Aus Wikipedia.de

Scanline Rendering

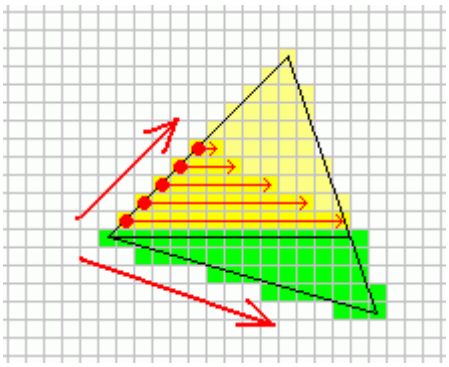
oder auch **Tastlinienrendern** ist ein Verfahren der 2D-Computergrafik zur Darstellung von grafischen Primitiven auf Rastergeräten. Es wird auch für 3D-Computergrafik eingesetzt, dies ist aber erst nach der projektiven Abbildung möglich.

Das Verfahren wird häufig in modernen Grafikkarten zur Darstellung von 3D-Grafik in Echtzeit verwendet und wird in der Regel über OpenGL oder DirectX umgesetzt.

Das Scanline-Rendering ist eine Form des Renderns, bei dem für jedes Primitiv ein in Frage kommender Bildbereich zeilenweise auf Überdeckung geprüft wird. Eine solche Zeile nennt man **Scanline** oder auch **Tastlinie**. Einen Bereich zusammenhängender Pixel in einer Scanline, die zum Primitiv gehören, nennt man **Span**.

Bei dem Verfahren werden meist nur Polygone (Dreieck, Rechteck, Fächer, Band, ...) als Primitive eingesetzt. In diesem Fall werden alle nicht horizontalen Kanten betrachtet und ihre Schnittpunkte mit jeder Zeile des Bildes berechnet. Das kann zwar schnell und einfach mit dem Mittelpunktsalgorithmus erfolgen. Dieser Algorithmus führt allerdings gerne zu ungeliebten Überschneidungen, weswegen man beim Scanline Rendering eher darauf verzichtet. Zur Bestimmung der Schnittpunkte dienen deswegen meist andere, inkrementelle Verfahren. Hierbei wird z.B. der Kehrwert der Steigung bei Iteration in y-Richtung für Linien mit einer Steigung m>1 auf den aktuellen x-Wert (basierend auf schneller Integer-Arithmetik) aufsummiert. Division wird in der Regel

vermieden, da sie entweder langsamer oder bei Hardwarelösungen nur mit unverhältnismässig hoher Gatterzahl zu realisieren ist.



Beim Abarbeiten einer Scanline werden dann die bereits überschrittenen Schnittpunkte mit den Objektekanten gezählt und die Parität betrachtet. Ist diese ungerade. so befindet man sich in einem Span, also zwischen zwei Objektkanten, ansonsten nicht. Somit sind auch konkave oder sich selbst überschneidende Objekte zulässig, jedoch sollten die Objekte einen geschlossenen Linienzug bilden. Die Pixel innerhalb eines Spans werden üblicher Weise ausgefüllt. Dabei wird zwischen den Attributen (z.B. gewünschte Farbe, Tiefe, Nebel-Parameter, Texturkoordinaten) der beiden Randpunkte interpoliert. Zum Setzen der finalen Pixelfarbe an einem bestimmten Punkt des Spans werden unter anderem die Pixelfarbe, Texturdaten, Tiefeninformationen, Überblendfaktoren und die zuvor schon für diesen Punkt ermittelten Pixeldaten herangezogen.

Z-Buffer

Das **Z-Buffering** (deutsch „Z-Pufferung“) wird in der Computergraphik angewendet, um die verdeckten Flächen in einer dreidimensionalen Computergrafik zu ermitteln. Durch die Informationen im **Z-Buffer** (deutsch „Z-Puffer“) stellt das Verfahren pixelweise fest, welche Elemente einer Szene gezeichnet werden müssen und welche verdeckt sind. Heutige Grafikkarten unterstützen Z-Buffering als Standardverfahren zur Lösung des Sichtbarkeitsproblems in Hardware. Die Erfindung des Z-Buffers wird zu meist Edwin Catmull zugeschrieben, obwohl Wolfgang Strasser das Konzept bereits 1974 in seiner Dissertation beschrieben hat.

Funktionsweise

Wenn ein Objekt von einer 3D-Grafikkarte gerendert wird, wird die Tiefeninformation der erzeugten Pixel (die Z-Koordinate) im so genannten Z-Buffer abgelegt. Dieser Puffer, gewöhnlich als zweidimensionales Array (mit den Indizes X und Y) aufgebaut, enthält für jeden auf dem Bildschirm sichtbaren Punkt des Objekts einen Tiefenwert. Wenn ein anderes Objekt im selben Pixel dargestellt werden soll, vergleicht der Renderalgorithmus die Tiefenwerte beider Objekte und weist dem Pixel den Farbwert des Objekts zu, das dem Beobachter am nächsten liegt. Die Tiefeninformation des ausgewählten Objekts wird dann im Z-Buffer gespeichert und ersetzt den alten Wert. Durch den Z-Buffer kann die Grafikkarte die natürliche Tiefenwahrnehmung nachbilden: ein nahe gelegenes Objekt verdeckt ein fernes Objekt.

Die Speichertiefe des Z-Buffers hat einen großen Einfluss auf die Qualität der Szene: Wenn zwei Objekte sehr eng beieinander liegen, können bei einem 8 Bit tiefen Z-Buffer leicht Artefakte entstehen. Ein Z-Buffer mit 16 Bit, 24 Bit oder 32 Bit Speichertiefe erzeugt weniger Artefakte.

Da die Abstandswerte nicht gleichmäßig im Z-Buffer abgelegt werden, werden nahe Objekte besser dargestellt als ferne, da ihre Werte genauer abgespeichert sind. Allgemein ist dieser Effekt erwünscht, er kann aber auch zu offensichtlichen Artefakten führen, wenn sich Objekte voneinander

entfernen. Eine Variation des Z-Bufferings mit ausgeglicheneren Entfernungswerten ist das so genannte **W-Buffering**.

Zum Erstellen einer neuen Szene muss der Z-Buffer gelöscht werden, in dem er mit einem einheitlichen Wert (üblicherweise Eins) überschrieben wird.

Auf aktuellen Grafikkarten (1999-2003) beansprucht der Z-Buffer einen bedeutenden Teil des verfügbaren Speichers und der Datenübertragungsrate. Mit verschiedenen Methoden wird versucht, den Einfluss des Z-Buffers auf die Leistung der Grafikkarte zu reduzieren. Dies ist zum Beispiel durch die verlustfreie Kompression der Daten möglich, da das Komprimieren und Dekomprimieren der Daten kostengünstiger ist als die Erhöhung der Datenübertragungsrate einer Karte. Ein anderes Verfahren spart Löschvorgänge im Z-Buffer: die Tiefeninformation wird mit alternierendem Vorzeichen in den Z-Buffer geschrieben. Ein Bild wird mit positiven Vorzeichen gespeichert, das nächste Bild mit negativem, erst dann muss gelöscht werden.

Die Mathematik des Z-Buffering

Der Bereich der Tiefeninformation im Kameraraum, der zu rendern ist, wird häufig durch den *nah*-Wert und *fern*-Wert von *z* definiert. Nach einer Perspektivtransformation wird der neue Wert von *z*, hier als *z*' bezeichnet, wie folgt berechnet:

$$z' = \frac{\text{fern} + \text{nah}}{\text{fern} - \text{nah}} + \frac{1}{z}\left(\frac{-2 \cdot \text{fern} \cdot \text{nah}}{\text{fern} - \text{nah}}\right)$$

Dabei ist *z*' der neue Wert von *z* im Kameraraum. Manchmal werden auch die Abkürzungen *w* und *w*' verwendet.

Die resultierenden Werte von *z*' werden auf Werte zwischen -1 und 1 normiert, wobei die Fläche bei *nah* den Wert -1 und die Fläche bei *fern* den Wert 1 erhält. Werte außerhalb dieses Bereichs stammen von Punkten, die sich nicht im Sichtbereich befinden, und sollten nicht gerendert werden.

Bei der Implementierung eines Z-Buffers werden die Werte der Scheitelpunkte eines Polygons linear interpoliert und die *z*'-Werte einschließlich der Zwischenwerte im Z-Buffer gespeichert. Die Werte von *z*' sind wesentlich enger an der Nah-Fläche verteilt und wesentlich mehr zur Fern-Fläche hin verstreut, was zu einer höheren Genauigkeit der Darstellung nahe dem Kamerastandpunkt führt. Je enger die Nah-Fläche an die Kamera gesetzt wird, desto geringer ist die Präzision im Fernbereich. Eine häufige Ursache für unerwünschte Artefakte bei entfernten Objekten ist, dass die Nah-Fläche zu eng an die Kamera gesetzt wurde.

Um einen W-Buffer zu implementieren, werden die unveränderten Werte von *z* bzw. *w* in den Buffer gespeichert, im allgemeinen als Fließkommazahlen. Diese Werte können nicht linear interpoliert werden – sie müssen invertiert, interpoliert und wieder invertiert werden. Die resultierenden *w*-Werte sind, im Gegensatz zu *z*, gleichmäßig zwischen *nah* und *fern* verteilt.

Ob ein Z-Buffer oder ein W-Buffer zu besseren Bildern führt, hängt vom jeweiligen Anwendungszweck ab.

Beleuchtungsmodell

Als **Beleuchtungsmodell** bezeichnet man in der 3D-Computergrafik allgemein ein Verfahren, das das Verhalten von Licht simuliert. Im Speziellen wird zwischen *globalen* und *lokalen* Beleuchtungsmodellen unterschieden. Meist ist mit dem Begriff „Beleuchtungsmodell“ jedoch ein lokales Beleuchtungsmodell gemeint.

Globale Beleuchtungsmodelle

Globale Beleuchtungsmodelle simulieren die Ausbreitung von Licht in einer Szene. Generell sind damit Radiosity oder Raytracing, beziehungsweise ihre Varianten gemeint. Der Begriff „globales Beleuchtungsmodell“ ist nicht mit dem Begriff *globales Beleuchtungsverfahren* zu verwechseln, der nur eine bestimmte Klasse globaler Beleuchtungsmodelle bezeichnet.

Lokale Beleuchtungsmodelle

Lokale Beleuchtungsmodelle simulieren das Verhalten von Licht auf Oberflächen. Dabei wird die Helligkeit bzw. Farbe eines von einem Punkt auf dieser Oberfläche in eine bestimmte Richtung reflektierten Lichtstrahls berechnet. Dies geschieht unter ausschließlicher Zuhilfenahme der Blickrichtung, des Lichteinfallswinkels, der Materialeigenschaften des Objektes und der Lichtquellen. Die indirekte Beleuchtung bleibt hier zunächst unberücksichtigt. Für die Simulation dieser Effekte sind globale Beleuchtungsmodelle zuständig; Raytracing beispielsweise sendet zu diesem Zweck weitere Strahlen aus.

Im Gegensatz zu den anderen Verfahren zur Darstellung eines Materials wie Bump Mapping simulieren Beleuchtungsmodelle nicht die Mesostruktur, sondern die Mikrostruktur eines Materials. Je nach Art der Darstellung müssen hierfür verschiedene Shading-Methoden verwendet werden.

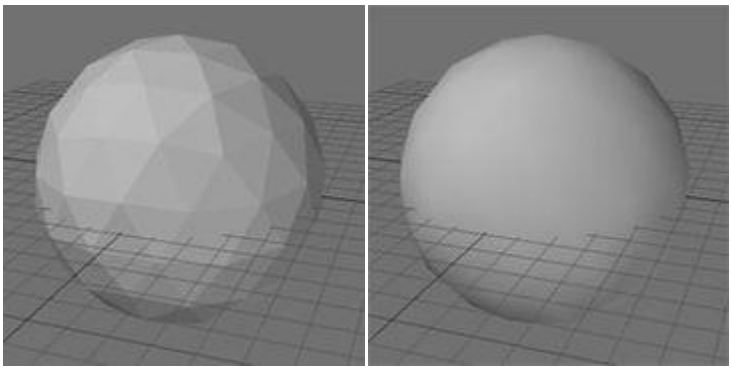
Die bekanntesten lokalen Beleuchtungsmodelle sind:

- Lambert-Beleuchtungsmodell (1760)
- Phong-Beleuchtungsmodell (1975)
- Torrance-Sparrow-Beleuchtungsmodell (1977)
- Schlick-Beleuchtungsmodell (1994)

Physikalisch gesehen sind lokale Beleuchtungsmodelle nichts anderes als die Interpretation und Umsetzung der bidirektionalen Reflektanzverteilungsfunktion (kurz BRDF).

Shading

Shading (engl. Schattierung, von *to shade* „schattieren“) bezeichnet in der 3D-Computergrafik im allgemeinen Sinne die Simulation der Oberfläche eines Objekts. Dies wird unter anderem durch Beleuchtungsmodelle ermöglicht.



Flat-Shading Ball Gouraud-Shading Ball

Im Spezialfall von Polygongeometrie bezeichnet Shading auch das Interpolationsverfahren, mit dem der Normalenvektor auf beliebigen Punkten der Oberfläche berechnet wird:

- Flat Shading führt keine Interpolation durch, sondern greift auf die Farbe des ersten Vertex des gerade zu zeichnenden Polygons zurück. Diese Farbe wird für alle Punkte des Polygons verwendet.
- Beim Gouraud Shading wird die Beleuchtung auf die Vertices des Polygons angewendet, die Farbwerte der einzelnen Pixel jedoch aus den Farbwerten der Vertices interpoliert.
- Phong Shading interpoliert für jeden Pixel aus den Normalen der Vertices eine interpolierte Normale des Pixels und wendet das Beleuchtungsmodell für jeden Pixel mit einer neuen Normale an.

Bei den vorgestellten Verfahren haben jeweils alle Lichtquellen Einfluss auf das gesamte Polygon. Das Beleuchten eines Polygonteils, etwa mit einem Spotlight, ist nicht möglich.

Flat Shading

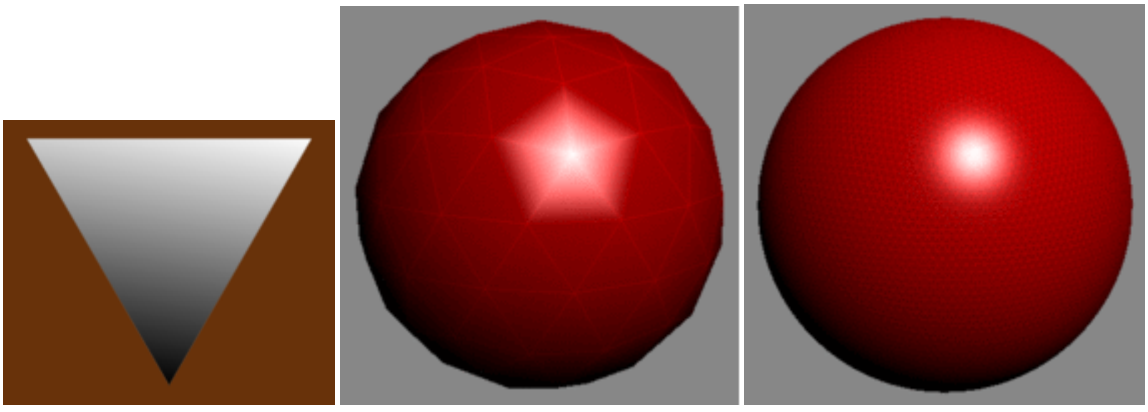
Flat Shading, auch **Constant Shading**, ist im Gegensatz zum Gouraud Shading oder Phong Shading ein sehr einfaches Schattierungsverfahren. Mit dieser Methode erhält jedes Pixel eines Polygons anhand der Flächennormale die gleiche Farbe und den gleichen Lichtwert. Dies hat eine abgestufte, eckige und unrealistische Erscheinung der Objekte besonders bei gekrümmten Oberflächen zur Folge (z. B. Mach Banding).

Um die facettenartige Darstellung zu vermindern, müssten die Polygone verkleinert werden, was einen erhöhten Rechenaufwand zur Folge hätte. Flat Shading findet daher besonders bei Objekten mit ebenen Flächen (Quader, Würfel, Pyramide, Prisma ...) Anwendung. Für Objekte mit gekrümmten Flächen (Zylinder, Kugel, Kegel ...) sollte jedoch das Gouraud- oder Phong Shading verwendet werden.

Polygone, die der Lichtquelle zugewandt sind, werden heller dargestellt als solche, die der Beleuchtungsquelle abgewandt sind. Eine entfernungsabhängige Lichtabnahme ist nicht darstellbar.

Nach der Drahtgitter-Methode ist Flat-Shading der schnellste Algorithmus zur Visualisierung einer 3D-Szene.

Gouraud Shading



Links: Gouraud-schattiertes Polygon.
Mitte: Gouraud-schattierte Kugel - man beachte die Ungenauigkeiten an den Polygonseiten.
Rechts: Ein unbewegtes Bild derselben Kugel, die durch eine größere Polygonanzahl dargestellt wird.

Das **Gouraud Shading** (auch als Intensitätsinterpolations- oder Farbinterpolations-Shading/Schattierung bekannt) ist ein Verfahren in der 3D-Computergrafik, um Polygon-Flächen zu

schattieren (engl. *to shade*). Als Scanline-Rendering-Verfahren wird es ausschließlich auf Raster-Ausgabegeräten verwendet. Benannt wurde es nach seinem Entwickler Henri Gouraud, der es erstmals 1971 vorstellte.

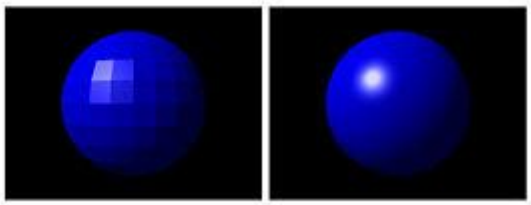
Beim Gouraud Shading werden zunächst die Farben des darzustellenden Polygons an seinen Eckpunkten (Vertices) berechnet und die Vertices auf die Bildebene projiziert. Das so entstandene zweidimensionale Abbild des Polygons wird anschließend zeilenweise abgearbeitet. Dabei werden die Farben an den Schnittpunkten der Kanten mit der Abtastzeile aus den Farben der Eckpunkte interpoliert. Die Farbwerte der Bildpunkte der Abtastzeile werden wiederum aus den Farben der Kanten interpoliert.

Als Beleuchtungsverfahren wird beim Gouraud Shading lediglich diffuse Reflexion nach dem Modell eines Lambert-Strahlers verwendet. Zur Berechnung der Farbwerte eines Vertex werden die Flächennormalen der anliegenden Flächen gemittelt. Die von der Fläche wiedergegebene Farbe wird aus dem diffusen Reflexionskoeffizienten, aus der Lichtquelle sowie aus dem Winkel zwischen Flächennormale und Lichtstrahl zur Lichtquelle gebildet.

Durch die Interpolation der Flächennormalen erscheinen so facettierte Oberflächen eines dargestellten Objekts nicht kantig wie beim Flat Shading, sondern weich. Die Silhouette des Objekts hingegen bleibt weiterhin kantig. Das Gouraud Shading ist eines der schnellsten Verfahren in der 3D-Computergrafik zur Darstellung räumlicher Objekte.

Nachteile dieser Art der Schattierungen sind Sprünge im Farbverlauf, Machsche Streifen und das Auftreten des Moiré-Effekts.

Phong Shading



Phong Shading wird auch Normalenvektor-Interpolations-Shading genannt und ist ein Verfahren aus der 3D-Computergrafik, um Polygon-Flächen mit Farbschattierungen zu versehen. Benannt wurde es nach seinem Entwickler Phong Bui-Tuong, der es erstmals 1975 vorstellte.

Beim Phong Shading werden an den Eckpunkten (Vertices) eines Polygons die Normalen berechnet und dann erweitert durch eine Berechnung von interpolierten Normalen entlang der Kanten für jede Projektion des Polygons auf einen Pixel. Farbstärken berechnen sich aus den interpolierten Normalen. Die Ergebnisse des Phong Shadings sind qualitativ besser als die des Gouraud Shadings, allerdings sind die mathematischen Berechnungen aufwändiger. Durch die vermehrte Interpolation der Normalen erscheinen so facettierte Oberflächen eines dargestellten Objekts sehr weich.

Phong-Beleuchtungsmodell

Das **Phong-Beleuchtungsmodell** ist ein Beleuchtungsmodell in der 3D-Computergrafik, das dazu verwendet wird, die Beleuchtung von Objekten zu berechnen. Das Modell wurde nach seinem Entwickler Bui-Tuong Phong benannt und erstmals 1975 vorgestellt (Bui-Tuong Phong : *Illumination for Computer Generated Pictures*, Juni 1975).

Das Phong-Modell ist zur Darstellung von glatten, plastikähnlichen Oberflächen geeignet. Dabei wird das Glanzlicht der Oberfläche durch den Term $cos^n(\theta)$ beschrieben, wobei der Parameter *n* die „Rauhigkeit“ der Oberfläche bestimmt.

Es handelt sich um ein vollständig empirisches Modell, das auf keinerlei physikalischer Grundlage aufbaut. So erfüllt es z.B. den Energieerhaltungssatz nicht, der vorschreibt, dass eine Oberfläche nicht mehr Licht, als von der Lichtquelle zur Verfügung gestellt wird, reflektieren kann. Zudem ist es relativ langsam zu berechnen. Das alternative Schlick-Beleuchtungsmodell vermeidet beide Nachteile.

Trotz seiner Mängel ist es noch häufig wesentlicher Bestandteil vieler gängiger 3D-Darstellungsverfahren.

Abstraktionen des Phong-Beleuchtungsmodelles

- Lichtquellen sind punktförmig
- nur Oberflächengeometrie wird berücksichtigt
- diffuse und spiegelnde Reflexion wird nur lokal modelliert
- ambiente Reflexion wird global modelliert

Zusammensetzung des reflektierten Lichtes

Im Phong-Beleuchtungsmodell wird die Reflexion von Licht als Kombination aus ambienter, ideal diffuser und ideal spiegelnder Reflexion beschrieben.

$$I_{out} = I_{ambient} + I_{diffus} + I_{specular}$$

Die Beschreibung der einzelnen Komponenten wird nachfolgend gezeigt.

Ambiente Komponente des reflektierten Lichtes

Die ambiente Komponente des reflektierten Lichts ist unabhängig vom Einfallswinkel des Lichtstrahls der Punktlichtquelle und vom Blickwinkel des Beobachters der Szene. Sie ist abhängig von dem für alle Punkte auf allen Oberflächen konstanten Umgebungslicht, und einem empirisch bestimmten Reflexionsfaktor (Materialkonstante).

$$I_{ambient} = I_a \cdot k_{ambient} \text{ mit}$$

- I_a ... Intensität des Umgebungslichts
- $k_{ambient}$ ... Materialkonstante

Diffuse Komponente des reflektierten Lichtes

Bei diffuser Reflexion wird das Licht unabhängig vom Standpunkt des Betrachters in alle Richtungen reflektiert (Lambertsches Gesetz). Die Lichtstärke des reflektierten Lichts der Punktlichtquelle hängt dennoch vom Einfallswinkel ab, da sich die Beleuchtungsstärke der Oberfläche mit dem Einfallswinkel ändert. Somit ist die Lichtstärke der diffusen Komponente vom Einfallswinkel des Lichtstrahls der Punktlichtquelle und von einem empirisch bestimmten Reflexionsfaktor (Materialkonstante) abhängig, jedoch vom Blickwinkel des Beobachters der Szene unabhängig.

$$\begin{aligned} I_{diffus} &= I_{in} \cdot k_{diffus} \cdot \cos\phi \\ &= I_{in} \cdot k_{diffus} \cdot (\vec{L} \cdot \vec{N}) \end{aligned} \text{ mit}$$

- I_{in} ... Lichtstärke des einfallenden Lichtstrahls der Punktlichtquelle
- k_{diffus} ... empirisch bestimmter Reflexionsfaktor für diffuse Komponente der Reflexion

- ϕ ... Winkel zwischen Normalenvektor der Oberfläche $\vec{N}$ und einfallendem Lichtstrahl $\vec{L}$

Spiegelnde Komponente des reflektierten Lichtes

Bei spiegelnder Reflexion wird das Licht in einer gewissen Umgebung der idealen Reflexionsrichtung reflektiert. Die Lichtstärke des reflektierten Lichtes ist vom Einfallswinkel des Lichtstrahls der Punktlichtquelle, von einem empirisch bestimmten Reflexionsfaktor (Materialkonstante) sowie der Oberflächenbeschaffenheit und vom Blickwinkel des Beobachters der Szene abhängig.

$$\begin{aligned} I_{specular} &= I_{in} \cdot k_{specular} \cdot \cos^n \theta \\ &= I_{in} \cdot k_{specular} \cdot (\vec{R} \cdot \vec{V})^n \end{aligned}$$ mit

- I_{in} ... Lichtstärke des einfallenden Lichtstrahls der Punktlichtquelle
- $k_{specular}$ ... empirisch bestimmter Reflexionsfaktor für spiegelnde Komponente der Reflexion
- θ ... Winkel zwischen idealer Reflexionsrichtung des ausfallenden Lichtstrahls $\vec{R}$ und Blickrichtung des Betrachters $\vec{V}$
- *n* ... konstanter Faktor zur Beschreibung der Oberflächenbeschaffenheit (rau kleiner 32, glatt größer 32, $n = \infty$ wäre ein perfekter Spiegel)

Vollständige Formel

Und hier die zusammengesetzte Formel für das Phong-Beleuchtungsmodell:

$$\begin{aligned} I_{out} &= I_a \cdot k_{ambient} + I_{in} \cdot k_{diffus} \cdot \cos\phi + I_{in} \cdot k_{specular} \cdot \cos^n \theta \\ &= I_a \cdot k_{ambient} + I_{in} \cdot (k_{diffus} \cdot \cos\phi + k_{specular} \cdot \cos^n \theta) \\ &= I_a \cdot k_{ambient} + I_{in} \cdot (k_{diffus} \cdot (\vec{L} \cdot \vec{N}) + k_{specular} \cdot (\vec{R} \cdot \vec{V})^n) \end{aligned}$$

mit $k_{ambient} + k_{diffus} + k_{specular} = 1$

Von I-Light Webseite

Ambiente Beleuchtung

Das Modell, in dem es keine äußere Lichtquelle gibt, bei jedem Objekt die Intensität sozusagen eingebaut ist, kann man sich als ziemlich unrealistische Welt nicht reflektierender, selbstleuchtender Objekte vorstellen. Die Objekte erscheinen als einfarbige Silhouette, es sei denn, seine einzelnen Teile sind unterschiedlich schattiert.

Mit einer *Beleuchtungsgleichung*, deren Variablen dem jeweiligen Punkt auf der Oberfläche des Objektes zugeordnet sind, wird ein Beleuchtungsmodell beschrieben. Die Auswertung der Beleuchtungsgleichung an einem oder mehreren Punkten eines Objekts nennt man *Beleuchten* des Objekts. Die Gleichung für dieses einfache Modell lautet:

$$I = k_i$$

I ist die resultierende Intensität des Punktes, der Koeffizient k_i bezeichnet die dem Objekt eigene konstante Intensität. Diese Gleichung enthält keine Ausdrücke, die von der Lage des jeweiligen Punktes abhängen, weshalb man sie nur einmal für jedes Objekt auswerten muß.

Geht man von einer diffusen Lichtquelle ohne bestimmte Richtung aus, entsteht ambientes Licht, das durch viele Lichtreflexionen an den Flächen der dargestellten Szene entsteht. Ambientes Licht nennt man auch indirekte Beleuchtung oder Hintergrundbeleuchtung. Gleichung (1) wird zu

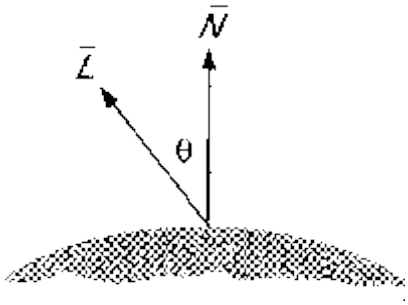
$$I = I_a k_a$$

I_a ist die Intensität des indirekten Lichts und wird für alle Objekte als konstant angenommen. Der *ambiente Reflexionskoeffizient* k_a liegt zwischen 0 und 1. Er gibt den Anteil des indirekten Lichts an, den die Oberfläche reflektiert. Damit zählt der ambiente Reflexionskoeffizient zu den *Materialeigenschaften* eines Objekts. Zusammen mit anderen Eigenschaften, die noch behandelt werden, charakterisiert er das Material, aus dem die Oberfläche des Objekts besteht. Der ambiente Reflexionskoeffizient steht nicht in direkter Verbindung zu einer physikalischen Eigenschaft des Materials, er ist eine empirische Größe.

Die ambiente Beleuchtung allein ist nicht besonders interessant und sieht auch keineswegs natürlich aus. Jedoch behandelt diese Komponente der resultierenden Intensität eines Punktes all die komplexen Möglichkeiten der Beleuchtung, die nicht durch andere Methoden in der Gleichung zum Ausdruck kommen werden.

Lambert Reflexion

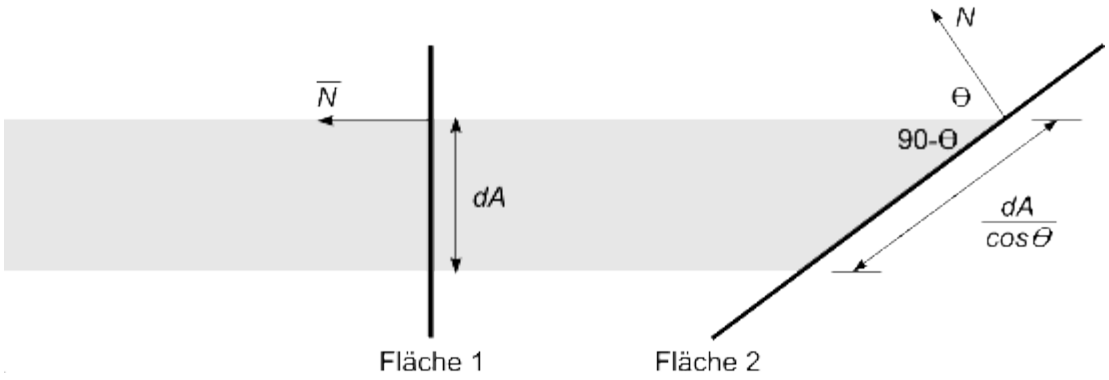
Indirekt beleuchtete Objekte sind proportional zur Intensität des Umgebungslichts mehr oder weniger hell, ihre Flächen sind jedoch gleichmäßig beleuchtet. Mit dem Vorhandensein einer punktförmigen Lichtquelle, die von einem Punkt ausgehend das Licht gleichmäßig in alle Richtungen strahlt, ändert sich die Helligkeit des Objekts von Punkt zu Punkt in Abhängigkeit vom Abstand und Richtung zur Lichtquelle.



Lambert Reflexion

Bei stumpfen, matten Flächen tritt *Lambert-Reflexion* oder *diffuse Reflexion* auf (Abbildung 1). Solche Flächen erscheinen aus allen Richtungen betrachtet gleich hell, da sie das Licht mit gleicher Intensität in alle Richtungen gleichförmig reflektieren. Die Helligkeit eines Punktes hängt nur vom Winkel Θ zwischen der Richtung $\bar{L}$zur Lichtquelle und der Flächennormalen $\bar{N}$ab.

Zur Bestimmung der Intensität pro Flächeneinheit betrachtet man die Fläche, die ein Lichtstrahl bedeckt (siehe Abbildung 2). Für Winkel 0° $\leq\Theta<$ 90° ist die Größe dieser Fläche proportional zu Θ. Für alle Winkel also umgekehrt proportional zu cos Θ. Die Größe der Fläche ist *dA*/cos Θ, wobei *dA* der Querschnitt des Lichtstrahls ist. Die Energie des Lichtstrahls ist konstant, pro Flächeneinheit jedoch proportional zu cos Θ, also umgekehrt proportional zu Θ. Bei 0° ist die Intensität am größten, dann trifft der Lichtstrahl senkrecht auf die Oberfläche.



Zusammenhang zwischen Fläche und Einfallswinkel

Wieviel Licht sieht aber ein Betrachter, der sich irgendwo im Raum befindet? Das Lambertsche Gesetz besagt, daß der Betrag des Lichts, den der differentielle Bereich *dA* zum Betrachter reflektiert, direkt proportional zum Kosinus des Winkels zwischen Blickrichtung und der Normalen ist. Da die Größe der sichtbaren Fläche umgekehrt proportional zum Kosinus dieses Winkels ist, heben sich diese beiden Faktoren gegenseitig auf. Die Intensität eines Punktes hängt also nur vom Winkel θ und nicht von der Blickrichtung ab. Die Gleichung für die diffuse Beleuchtung, mit normalisierten Vektoren, lautet:

$$I = I_p k_d \cos\theta \quad \text{bzw.} \quad I = I_p k_d \left(\overline{N} \cdot \overline{L}\right) \tag{3}$$

Es wird hier und in den folgenden Gleichungen davon ausgegangen, daß der Winkel im Bereich $0° \leq \Theta < 90°$ liegt, damit sich die Lichtquelle auf den untersuchten Punkt auswirkt. Lichtquellen hinter der Fläche erreichen den Punkt also nicht. Bei der Berechnung der Normalen im voraus und Transformation der Polygoneckpunkte ist zu beachten, daß man manche Transformationen, wie Scherung oder ungleichförmige Skalierung, nicht durchführen darf, da sie den Winkel nicht erhalten. Das hat zur Folge, daß die im voraus berechnete Normale nicht mehr senkrecht auf der Fläche steht. Es existieren jedoch Methoden zur Transformation von Normalen, wenn beliebige Transformationen auf einem Objekt angewandt werden.

Wenn eine punktförmige Lichtquelle weit genug von dem zu schattierenden Objekt entfernt ist, schneiden ihre Lichtstrahlen alle Flächen unter annähernd dem gleichen Winkel. In diesem Fall spricht man von einer *gerichteten Lichtquelle*, $\overline{L}$ ist dann in jedem Punkt konstant.

Bei der Darstellung von Objekten nur mit dem Lambert Modell, sehen sie sehr grell aus, als würden sie in einem dunklen Raum von einem Blitzlicht beleuchtet. Zur Steigerung des realistischen Eindrucks erweitert man die Gleichung (3) um einen Anteil für die Hintergrundbeleuchtung. Ambiente und diffuse Reflexion werden kombiniert

$$I = I_a k_a + I_p k_d \left(\overline{N} \cdot \overline{L}\right)$$

Abschwächung der Lichtquelle

Wenn sich die Projektionen zweier paralleler Flächen gleichen Materials, die vom Blickpunkt aus beleuchtet werden, überlappen, so kann man sie mit Gleichung (4) nicht voneinander unterscheiden. Die Entfernung zur Lichtquelle oder zum Betrachter ist kein Faktor, der die Gleichung beeinflußt. Daher wird ein Abschwächungskoeffizient f_{ab} für die Lichtquelle eingeführt:

$$I = I_a k_a + f_{ab} I_p k_d \left(\overline{N} \cdot \overline{L} \right)$$

Die Energie einer punktförmigen Lichtquelle nimmt mit dem Kehrwert des Quadrats des Abstands d_L zwischen Fläche und Lichtquelle ab. Dies muß bei der Wahl von f_{ab} berücksichtigt werden:

$$f_{ab} = 1 / d_L^2$$

Für große d_L ändert sich der Abschwächungskoeffizient nur sehr gering, während er bei naher Lichtquelle stark schwankt. Dadurch werden die Flächen bei gleichem Winkel sehr unterschiedlich schattiert. Das ist zwar für punktförmige Lichtquellen theoretisch korrekt, aber in der Natur werden Objekte typischerweise nicht von punktförmigen Lichtquellen beleuchtet und schon gar nicht mit vereinfachten Beleuchtungsmodellen der Computergrafik schattiert. Dieser Abschwächungskoeffizient sollte sowohl die atmosphärische Abschwächung, als auch die Abnahme der Energiedichte von der Lichtquelle zum Objekt hin approximieren. Die Gleichung (7) stellt einen Kompromiß dar und ermöglicht darüber hinaus mehr Effekte als die quadratische Abschwächung.

$$f_{ab} = \min\left(\frac{1}{c_1 + c_2 d_L + c_3 d_L^2}, 1 \right)$$

Gleichung 7 bietet neben der quadratischen Abhängigkeit von der Entfernung der Lichtquelle (Gleichung 6) die Möglichkeit einer feineren Abstimmung durch lineare Abhängigkeit und einer zusätzlichen Konstante. c1 verhindert, daß der Nenner zu klein und damit die Abschwächung zu groß wird, wenn sich die Lichtquelle sehr nahe am Objekt befindet. Die Parameter c_n werden vom Benutzer durch Testen bestimmt, bis eine subjektiv gute Darstellung erfolgt. Der Ausdruck wird immer unter 1 gezwungen, um Abschwächung sicherzustellen.

Farbiges Licht und farbige Flächen

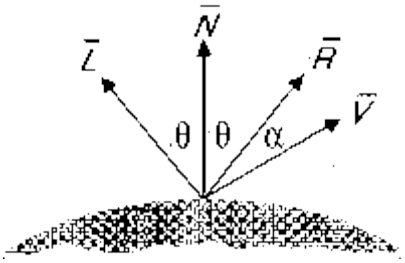
Farbe wird üblicherweise durch separate Gleichungen für die Komponenten des Farbmodells berücksichtigt. Die diffuse Farbe eines Objekts wird durch je einen Wert O_d für jede Komponente dargestellt. Das Tripel (O_{dR}, O_{dG}, O_{dB}) definiert beispielsweise die Anteile eines Objekts im RGB-Farbsystem. Für den roten Anteil wird I_{pR} der Lichtquelle proportional zu $k_d O_{dR}$ reflektiert. Mit Gleichung (8) kann der Anwender den Anteil der ambienten und diffusen Reflexion steuern, ohne die Farbzusammensetzung zu ändern, da es nur einen einzigen Koeffizienten für die reflektierten Anteile gibt:

$$I = I_{a\lambda} k_a O_{d\lambda} + f_{ab} I_{p\lambda} k_d O_{d\lambda} \left(\overline{N} \cdot \overline{L} \right)$$

Anstatt sich auf ein bestimmtes Farbmodell zu beschränken, werden die Terme der Beleuchtungsgleichung, die von der Wellenlänge abhängen, mit dem Index λ gekennzeichnet. Denn die vereinfachte Annahme, daß ein Farbmodell mit drei Komponenten das Zusammenspiel von Licht und Objekten vollständig modellieren kann, ist falsch. Sie ist aber einfach zu implementieren und liefert oft akzeptable Ergebnisse. Man müßte die Gleichung theoretisch im gesamten modellierten Spektralbereich auswerten, in der Praxis wird dies aber nur für diskrete Spektralwerte gemacht.

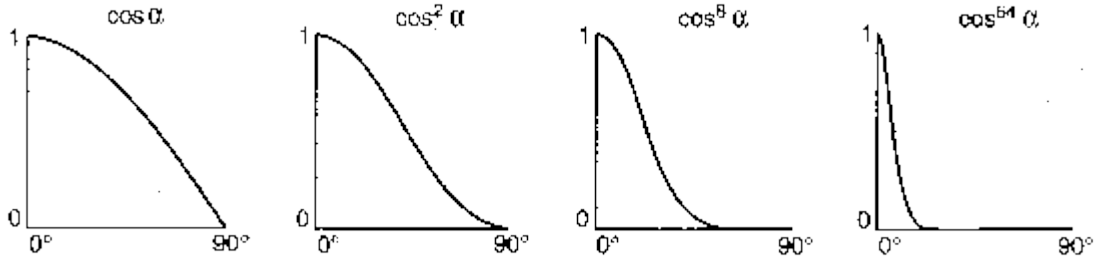
Spiegelnde Reflexion

Die spiegelnde oder gerichtete Reflexion kann man an jeder glänzenden Fläche beobachten. Wird z.B. ein Apfel mit einem hellen weißen Licht beleuchtet, entsteht die Spiegelung durch gerichtete Reflexion. Das Licht, was der Rest des Apfels reflektiert, entsteht durch diffuse Reflexion. Der Apfel erscheint an der spiegelnden Stelle nicht grün, sondern in der Farbe der Lichtquelle, nämlich weiß. Objekte wie ein gewachster Apfel oder glänzender Kunststoff haben eine transparente Oberfläche. Kunststoffe bestehen zumeist aus Pigmentteilchen, die in ein transparentes Material eingebettet sind. Das Licht, das durch die Reflexion an der farblosen Oberfläche entsteht, hat die Farbe der Lichtquelle.



Spiegelnde Reflexion

Wird nun die Position des Beobachters, also die Blickrichtung verändert, bewegt sich das Glanzlicht ebenfalls. Glänzende Flächen reflektieren das Licht ungleichmäßig in verschiedene Richtungen. Ein absolut perfekter Spiegel reflektiert das Licht nur in der Richtung $\bar{R}$, die durch Spiegeln von $\bar{L}$an $\bar{N}$ entsteht. Ein Betrachter würde nur dann reflektiertes Licht sehen, wenn seine Blickrichtung mit -$\bar{R}$ übereinstimmt. In dem Fall wird in Abbildung 2 der Winkel gleich Null. $\bar{V}$ist die Richtung zum Blickpunkt.



Intensitätsabfall vom Lichtpunkt aus für verschiedene n

Das Modell von Phong

Von Phong Bui-Tuong stammt ein weit verbreitetes Beleuchtungsmodell für nicht perfekte Reflektoren (z.B. den Apfel). Dabei wird davon ausgegangen, daß die gerichtete Reflexion bei α=0 maximal ist und mit steigendem αrapide abnimmt. Diese rapide Abnahme wird durch $\cos^n \alpha$ approximiert. *n* ist ein Materialkoeffizient, dessen typische Werte zwischen 1 und mehreren Hundert liegen. Kleine Werte erzeugen eine langsame Intensitätsabnahme, große Werte simulieren ein kleines, scharfes Glanzlicht. Das Phong Modell basiert auf den Arbeiten von Warnock [Foley94], berücksichtigt jedoch zudem unterschiedliche Positionen für Lichtquelle und Betrachter.

Der Betrag des spiegelnd reflektierten Lichts *W(*Θ*)* hängt auch wieder vom Einfallswinkel Θab. Gleichung (8) ist ein Term dafür hinzuzufügen:

$$I = I_{a\lambda} k_a O_{d\lambda} + f_{ab} I_{p\lambda} \left[k_d O_{d\lambda} \cos\theta + W(\theta) \cos^n \alpha \right]$$

Der Term in eckigen Klammern steht für den Anteil des reflektierten Lichts, sowohl durch diffuse, als auch durch spiegelnde Reflexion. Sind $\bar{R}$und $\bar{V}$normalisiert, gilt auch hier wieder cos α= $\bar{R}$*$\bar{V}$. *W(*Θ

) wird durch eine Konstante k_s ($0 \leq k_s \leq 1$) ersetzt, den *spiegelnden Reflexionskoeffizienten*. In der Praxis wird der Wert für k_s experimentell so bestimmt, daß das Resultat zufriedenstellend aussieht, etwa die Realität für den Anwendungsfall befriedigend nah nachbildet.

$$I = I_{a\lambda} k_a O_{d\lambda} + f_{ab} I_{p\lambda} \left[k_d O_{d\lambda} \left(\overline{N} \cdot \overline{L} \right) + k_s \left(\overline{R} \cdot \overline{V} \right)^n \right]$$

Man beachte, daß die Farbe des Glanzpunkts nicht von den Materialeigenschaften abhängt ($O_d\lambda$ ist nur Faktor in den Komponenten ambienter und diffuser Reflexion). Das Phong Modell eignet sich daher gut zum Beleuchten von Kunststoffkörpern. Andererseits kann man auch sagen, daß Phong-beleuchtete Körper sehr künstlich aussehen.

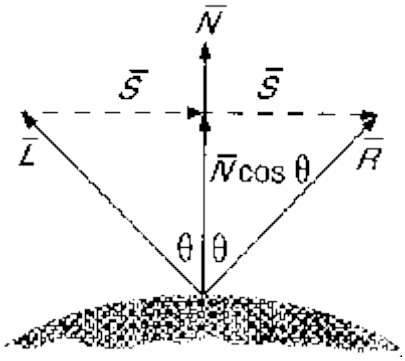
Diffuse und ambiente Reflexion haben die Farbe des Objekts, bzw. der Fläche. Gerichtete Reflexion nimmt die Farbe der Lichtquelle an, die *spiegelnde Farbe* des Objekts ($O_s\,\lambda$). Diese sind nicht zwangsläufig gleich. Ein kleiner Zusatz zu Gleichung (10) berücksichtigt den Effekt:

$$I = I_{a\lambda} k_a O_{d\lambda} + f_{ab} I_{p\lambda} \left[k_d O_{d\lambda} \left(\overline{N} \cdot \overline{L} \right) + k_s O_{s\lambda} \left(\overline{R} \cdot \overline{V} \right)^n \right]$$

Berechnung des Reflexionsvektors

Aus einfachen geometrischen Überlegung (siehe Abbildung 5) geht hervor, daß $\overline{R}$ durch Spiegelung von $\overline{L}$ an $\overline{N}$ entsteht. Da $\overline{L}$ und $\overline{N}$ normalisiert sind, hat die Projektion den Wert $\overline{N}\cos\theta$. Es gilt $\overline{R} = \overline{N}\cos\theta + \overline{S}$, wobei $\overline{S} = \overline{N}\cos\theta - \overline{L}$ ist. Daraus folgt:

$$\overline{R} = 2\overline{N}\cos\theta - \overline{L} = 2\overline{N}\left(\overline{N} \cdot \overline{L}\right) - \overline{L}$$

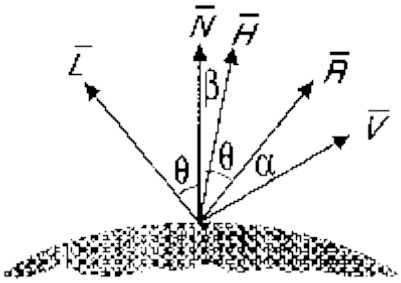


Berechnung des Reflexionsvektors

Für unendlich weit entfernte Lichtquellen ist $\overline{N} * \overline{L}$ für ein gegebenes Polygon konstant, während sich $\overline{R} * \overline{V}$ auf der Oberfläche ändert. Bei gekrümmten Flächen ändern sich beide Winkel an jedem Punkt.

Der Halfway-Vektor

Eine andere Formulierung des Phong Beleuchtungsmodells arbeitet mit dem sogenannten Halfway-Vektor $\overline{H}$. Er liegt zwischen der Richtung zur Lichtquelle und der Richtung zum Betrachter (siehe Abbildung 6). $\overline{H}$ ist die Richtung des maximalen Glanzlichts. Der Betrachter sieht das Glanzlicht am hellsten, wenn $\overline{R}$ und $\overline{V}$ zusammenfallen. Genau dann sind auch $\overline{N}$ und $\overline{H}$ gleich orientiert. Für die spiegelnde Reflexion wird nun nicht der Winkel α sondern β benutzt: $(\overline{N} * \overline{H})n$ mit $\overline{H} = (\overline{L} + \overline{V}) / |\overline{L} + \overline{V}|$ ($\overline{H}$ normalisiert). Liegen Lichtquelle und Betrachterposition bei unendlich, so ist $\overline{H}$ konstant.



Der Halfway-Vektor

Achtung, da αund βkeineswegs gleich sind, erzeugt der gleiche Exponent n mit beiden Formulierungen unterschiedliche Ergebnisse! Man sollte außerdem nicht vergessen, daß der Term cosn zwar erkennbar glänzende Flächen erzeugt, dieser jedoch nicht auf einem theoretischen Modell, sondern auf empirischen Erfahrungen beruht.

Aus „CG1 Buch“

Diffuse reflection: surface appears equally bright from any viewing angle
Specular reflection: only when we view the surface from a particular direction
Ground light/ ambient light: macht alles heller, wie bei GI. General brightness Level for a scene, produce a uniform ambient lighting that is the same for all objects
Diffuse Reflexion: Every surface is to be treated as an ideal diffuse reflector. Stark 1, absorbierend 0

Ausarbeitung von ska

Bsp 4

Backface Culling (in CG1Object)

Die Polygone, deren Oberflächennormale von Betrachter wegzeigen werden nicht gezeichnet.

Skalarprodukt Blickrichtungsvektors mit Oberflächennormale V_{vier}*N>0

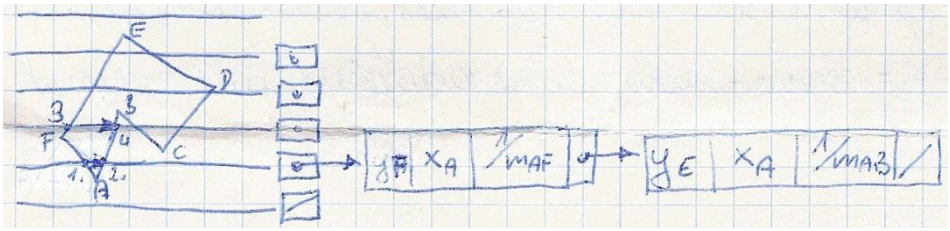
Einsetzen des Blickpunktes in Ebenengleichung Ax + By + z + D <0

Kreuzprodukt von $\underset{xy}{\rightarrow}$ und $\underset{xz}{\rightarrow}$ = Normale zum Polygon < 0 = unsichtbares continue();

ScanFill

(in CG1ScanFilledPolygon) Jede Scanline wird mit Polygon geschnitten und innere Teile werden gefüllt. => Kanten nach niedrigeren y- Wert sortieren und in edgeliste für jede Kante [max y-Wet, Anfangs-x Wert, Steigung] gespeichert. In actuellist werden die aktuellen (Eckpunte) der Kanten, die durch die Scanline gehen gespeichert (alle Kanten, deren max. y überschritten wurde, entfernt (und sortiert))

fillScan () + draw Clipped()



Polygoneckpunkt genau auf einer Linie => Punktkoordinaten um ε verschieben um 2 Kanten zu erkennen.

Bsp 5

Z- Buffer/ depth B

(in CG1Canvas initialisiert/ rein geschrieben: setPixel (int x, int y, Color c, double z)) = Sichtbarkeitsbuffer.

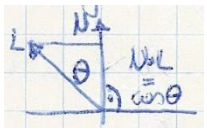
Vergleicht Z- Werte des nächsten zu zeichnenden Objektes mit z- Buffer Wert. Wenn z größer als Z Buffer, dann z Buffer = z und (z) Polygon gezeichnet

FaceNormaleCalculation

(in CG1Object) für Berechnung der IlluminationColor in FlatShadedPolygon.
Skalarprodukt $\underset{xy}{\rightarrow} x \underset{xz}{\rightarrow}$ von vertexTable (unbearbeitete/ nicht transformierte Punkte + .normalzie());

FlatShading

(in CG1 FlatShadedPolygon) = Schattieren eines Poylgons, wobei Oberflächeneigenschaften/ Farbwiedergabe konstant. Hängt von Winkel des Normalvektors und Lichtvektor ab, Mittelfarbwert der Eckpunkte.



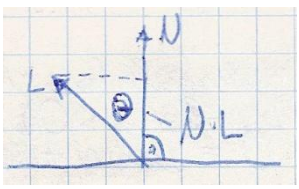
Je flacher das Licht auf die Fläche trifft, desto dunkler.

Grenzen der ausgefüllten Polygone sichtbar da Mittel der Eckpunkte verwendet werden=> Lösung: Interpolieren der Schattierung zwischen den Polygonen=> Gouraud Phong

Diffuse Illumination Model

(in CG1D ????) = diffuses + ambiente (Umgebungslicht) Beleuchtung eines Objektes berechnet => I = Reflexionskoeffizient k2 * Lichtquellenintensität I, *N * L

Normale x Lichteinfallsvektor cos Θ



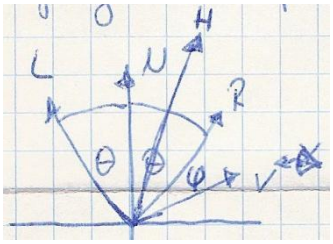
Bsp 6

PhongIlluminationModel

(in CG1IlluminationModelPhong) =specular + diffusses + ambientes Beleuchtungsmodell (+ Glanzpunkte)

Ambient + diff + spec cos Θ

⇨ $I = ka + Ia + (kd * Istrich * N \times L + ks * Istrich * (N \times Hstrich)^n)$



Spiegelnder Reflexionskoeffizient ks * Istrich * Winkel zwischen Reflexionsstrahl und Richtung zum Auge (Winkel zwischen Normale und Halber zwischen L. und V.)

perfekte spekular Reflektion, wenn cos Θ im unendlichen liegt

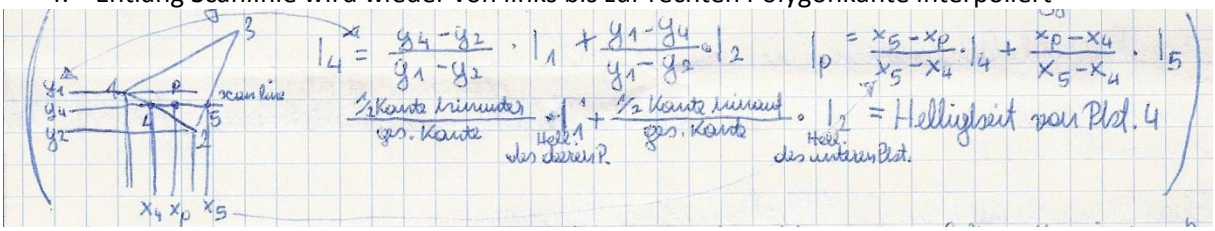
H = view + light vector addieren und normalisieren

N = normal vector

Gouraud Shading

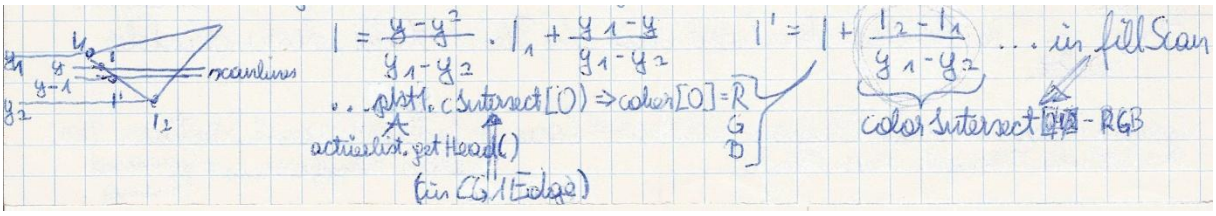
(in CG1) Helligkeitswerte an den Eckpunkten der Polygone berechnet und von diesen aus durch lineare Interpolation jedes Polygon gefüllt.

1. Normale als Mittelwert der Normalen aller angrenzenden Polygone berechnet.
2. Aus Oberflächeneigenschaft, Normale und Lichteinfallsrichtung wird Schattierung für jeden Eckpunkt berechnet.
3. Entlang der Polygonkante werden die Helligkeitswerte linear interpoliert, daher für jeden Schnittpunkt einer Scanlinie wird ein Wert ermittelt.
4. Entlang Scanlinie wird wieder von links bis zur rechten Polygonkante interpoliert



Wenn Normalen genau einen Glanzpunkt erzeugt zufällig und da Silhouette nicht verändert wird= Fehlerquellen.

Für dasselbe Ergebnis bei Gouraud Shading wie bei Phong Shading müssten Polygone 1 Pixel groß sein ~



Phong Shading

erzeugt konsistentere Glanzeffekte, da die mittleren Normalen der Polygoneckpunkte entlang Polygonkanten (interpoliert) und anschließend entlang der Scanlines interpoliert werden.

Man rechnet nicht mit Intensity Werten (I) , sondern Normalen (N).

$$N = \frac{N_1(y - y_2)}{(y_1 - y_2)} + \frac{N_2(y_1 - y)}{(y_1 - y_2)}$$

Und berechnet dann die Helligkeit für jedes Pixel extra nach=> in fillScan: for Schliefe: illColor = do Ill.=> set Pixel (…. illColor)

Infos

Die meisten Sachen bei „Praktisches“ sind aus dem Forum 1:1 kopiert (siehe Quellen) oder ein wenig umgeschrieben worden. Der Rest ist von mir geschrieben worden, genauso wie Beispiel 4, 5 und 6.

„Ausarbeitung von ska“ ist eine Ausarbeitung die ska für sich schrieb, und ich abtippte, da ich sie gut und hilfreich fand. Vielen Dank noch Mal an ska für die Scans!

Bei einigen Sachen war ich mir nicht ganz sicher ob sie so stimmen. Deswegen habe ich diese Sätze rot geschrieben (zum leichten erkennen das man da „aufpassen“ sollte, da es womöglich nicht stimmt).

Ich habe die Ausarbeitung so gut es geht gemacht, aber trotzdem können sich Fehler einschleichen! Falls man welche findet, bitte per E- Mail oder PM an mich weiter leiten damit ich sie ausbessere!

Version: 0.9

Zusammenfassung: Martin Tintel (mtintel)

Quellen und weiterführende Links:

http://www.informatik-forum.at/showthread.php?t=38720
http://www.informatik-forum.at/showthread.php?t=59650
http://www.informatik-forum.at/showthread.php?t=61647
http://www.informatik-forum.at/showthread.php?t=14901
http://www.cg.tuwien.ac.at/courses/CG/textblaetter.html (“Aus PDFs von Werner Purgathofer”)
http://olli.informatik.uni-oldenburg.de/i-light/tutorial.html („Von I-Light Webseite“)
http://de.wikipedia.org/wiki/Beleuchtungsmodell
http://de.wikipedia.org/wiki/Shading
http://de.wikipedia.org/wiki/Flat_Shading
http://de.wikipedia.org/wiki/Gouraud_Shading
http://de.wikipedia.org/wiki/Phong_Shading
http://de.wikipedia.org/wiki/Phong-Beleuchtungsmodell

CG1LU Abgabe 1 Ausarbeitung (Ausarbeitung von mir für erste Abgabe)
http://de.wikipedia.org/wiki/Epsilon
http://de.wikipedia.org/wiki/Z-Buffer
http://de.wikipedia.org/wiki/Scanline_Rendering
Scans von ska („Ausarbeitung von ska “)
http://de.wikipedia.org/wiki/Lambertsches_Gesetz
http://de.wikipedia.org/wiki/Mach_Banding
Computer Graphics with OpenGL Third Edition