

Basics of 3D-Vector Graphics

Copyright © by V. Miszalok, last update: 22-11-2005

- ↓ [Erweiterung der 2D-x,y-Koordinaten um eine z-Koordinate](#)
- ↓ [Drehungen um 3 Achsen: Pitch, Yaw, Roll](#)
- ↓ [Vertex und Vektor](#)
- ↓ [Normale](#)

Spricht man von 3D, ist so gut wie immer 3D-Vektorgraphik gemeint.

3D-Rastergraphik gibt es nur in exotischen Anwendungen in der Medizin, Materialforschung und Geologie, aber nicht bei Games und Unterhaltung. Auf dem PC des Normalbürgers existiert nur 2D-Rastergraphik.

Den Vorlesungstitel "3D-Vektorgraphik" könnte man folglich kürzer fassen: "3D-Graphik" würde ausreichen.

Erweiterung der 2D-x,y-Koordinaten um eine z-Koordinate

Jede Polygonecke bekommt zusätzlich zu seiner x- und y-Koordinate noch eine z-Koordinate.

In diesem Sinne war jede 2D-Vektorgraphik schon immer eine 3D-Vektorgraphik, nur hatten alle z-Koordinaten den Wert 0.0 und weil diese immer Null waren, waren sie nicht erwähnt worden.

Beispiel für die Erweiterung von Scroll, Zoom und Rot von 2D nach 3D:

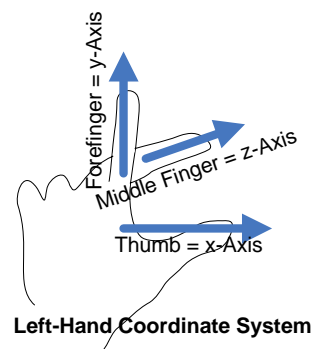
```
double arcus = alpha * 2.0 * Math.PI / 360.0;
float cosinus = (float)Math.Cos( arcus );
float sinus = (float)Math.Sin( arcus );
for ( i=0; i<n; i++ )
{
  p[i].X += dx;
  p[i].Y += dy;
  p[i].Z += dz; //new dimension
  p[i].X *= zoomx;
  p[i].Y *= zoomy;
  p[i].Z *= zoomz; //new dimension
  float help = p[i].X * cosinus - p[i].Y * sinus;
  p[i].Y = p[i].X * sinus + p[i].Y * cosinus;
  p[i].X = help; //Rot-operation around the z-axis doesn't change
}
```

Scroll und Zoom bekommen in 3D je eine neue z-Zeile, die 2D-Rotation bleibt identisch in 3D erhalten ohne neue z-Zeile. Begründung: Die Drehung in der x,y-Ebene ist eine Drehung um eine Achse senkrecht zur Ebene = z-Achse.

Bei einer solchen Drehung behalten alle Vertices ihre z-Koordinaten, nur ihre x- und y- Koordinaten ändern sich.

Man kann die z-Achse entweder vor oder hinter die x,y-Ebene stellen. Dadurch entstehen zwei verschiedene Koordinatensysteme, was viel Verwirrung stiftet. Default zeigt bei DirectX und OpenGL die z-Achse hinter die x,y-Ebene des Displays = Left-Hand Cartesian Coordinate System = x-Achse von links nach rechts, y-Achse von unten nach oben, z-Achse von vorn nach hinten.

Mehr Information: [3-D Coordinate Systems](#)



Drehungen um 3 Achsen: Pitch, Yaw, Roll

Es gibt in 3D neue, in 2D unbekannte Drehungen, nämlich die um die x-Achse und um die y-Achse. 3D kennt folglich nicht nur eine Art von Drehung sondern drei Arten mit jeweils eigenem Winkel, die nach Seemannsart benannt sind:

Pitch = Neigungswinkel = Drehwinkel um die x-Achse

Yaw = Gierwinkel = Drehwinkel um die y-Achse

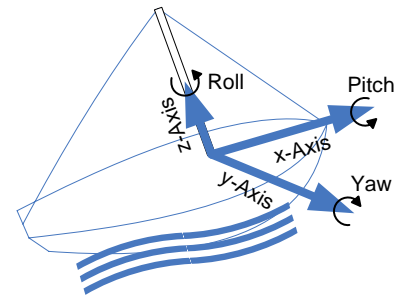
Roll = Rollwinkel = Drehwinkel um die z-Achse

Vorsicht: Rätselhafterweise nennt die DirectX-Funktion

`RotationYawPitchRoll` die 3 Drehungen in der Reihenfolge

y, x, z und erwartet die Winkelparameter auch so. siehe:

<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/...>



In 2D gibt es nur Rollwinkel, die meist **alpha** heißen.

Vertex und Vektor

Die Begriffe Vertex und Vektor scheinen zunächst identische Bedeutung zu haben.

Beide bezeichnen eine Datenstruktur vom Typ `{ float X; float Y; float Z; }`. Trotz dieser Identität stehen verschiedene Anschauungen hinter diesen Begriffen.

Anschauliche Interpretation von Vertex ist ein Raumpunkt ähnlich einem Stern im Weltraum. Ein Array von Vertices stellt man sich folglich als eine Punktwolke oder eine Art Sternwolke vor.

Vorteil: einfach. Nachteil: Man hat keinen Zugang zur Vektorrechnung.

Anschauliche Interpretation von Vektor ist ein Pfeil vom Ursprung des Koordinatensystems $(0,0,0)$, der an einer Stelle (x,y,z) seine Spitze hat. Ein Array von Vektoren stellt man sich folglich als ein Bündel von Pfeilen vor, die strahlenförmig von $(0,0,0)$ nach außen zeigen. Jeder Pfeil besitzt einen Raumwinkel und eine Länge.

Vorteil: Vektoren kann man addieren, subtrahieren und multiplizieren. Nachteil: komplizierter als Vertex.

1) Vektoraddition : $v_0 + v_1 = \{ v_0.X+v_1.X, v_0.Y+v_1.Y, v_0.Z+v_1.Z \}$ setzt v_1 an die Spitze von v_0 und zeigt dann vom Fußpunkt von v_0 an die Spitze von v_1 .

2) Vektorsubtraktion: $v_1 - v_0 = \{ v_1.X-v_0.X, v_1.Y-v_0.Y, v_1.Z-v_0.Z \}$ ist der Vektor von der Spitze von v_0 zur Spitze von v_1 .

3) Skalarprodukt : $v_0 * v_1 = v_0.X*v_1.X + v_0.Y*v_1.Y + v_0.Z*v_1.Z = \cos(\alpha)$ ist der Cosinus des Winkels zwischen v_0 und v_1 , wenn v_0 und v_1 beide die Länge 1.0 haben.

4) Kreuzprodukt : ist ein Vektor, der senkrecht steht auf der Ebene, die v_0 und v_1 aufspannt.

$$v_0 \times v_1 = \{ v_0.Y*v_1.Z - v_0.Z*v_1.Y, \\ v_0.Z*v_1.X - v_0.X*v_1.Z, \\ v_0.X*v_1.Y - v_0.Y*v_1.X \}$$

5) Betrag = Länge von v : $Länge = \text{Math.Sqrt}(v.X^2 + v.Y^2 + v.Z^2)$.

Anwendungen:

Für 1) gibt es nur selten eine Verwendung in der 3D-Graphik (wohl aber in der Physik).

Mit 2) berechnet man die Verbindungsvektoren zwischen den Vertices.

Mit 3) berechnet man den Auftreffwinkel des gerichteten Lichts auf eine Fläche = Face = meistens Dreieck.

Mit 4) berechnet man die Normale einer Fläche = Face = meistens Dreieck.

Mit 5) berechnet man die Länge und wenn die Länge eines Vektors keine Rolle spielen soll, kappt man diese auf 1.0 = normierter Vektor $v = \{ v.X/Länge, v.Y/Länge, v.Z/Länge \}$

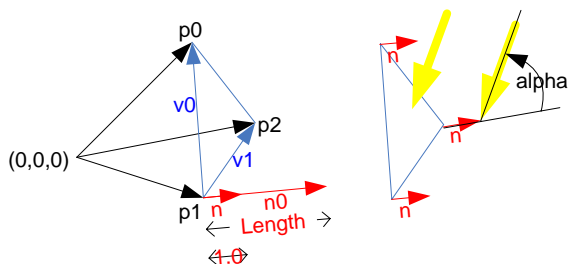
Zusammenfassung: Benutzen Sie zunächst die einfache Vertex-Denkweise und wenn Sie Winkel, Normale und Normierung brauchen, wechseln Sie zur komplizierteren Vektor-Denkweise.

Normale

2D-Flächen haben nur Vorderseiten, in 3D muss man sich auch um Rückseite und Orientierung kümmern.
3D-Flächen = Faces benötigen deshalb in der Regel eine Normale = "Face Normal" = senkrechter Lotvektor auf die Fläche. Richtung zur Außenwelt.

Berechnung der Normalen eines Dreiecks p_0, p_1, p_2 :

- 1) $v_0 = p_0 - p_1$ // Vektor von p_1 nach p_0
- 2) $v_1 = p_2 - p_1$ // Vektor von p_1 nach p_2
- 3) $n_0 = \text{Kreuzprodukt } v_0 \times v_1$ // Normale ohne Normierung
- 4) $\text{Length} = \text{Math.Sqrt}(n_0.X^2 + n_0.Y^2 + n_0.Z^2)$ berechnen // Länge von n_0
- 5) $n = \{ n_0.X/\text{Length}, n_0.Y/\text{Length}, n_0.Z/\text{Length} \}$ // normierte Normale



siehe: **Cross Product Applet**

www.phy.syr.edu/courses/java-suite/crosspro.html

Die Normale bestimmt, wo Vorder- und wo Rückseite ist und in welchem Winkel α Face zum Licht steht. Aus dem Skalarprodukt normierte Normale mal normiertem Lichtvektor erhält man $\cos(\alpha)$, was die Helligkeit bestimmt. Face ist am hellsten, wenn das Licht koaxial zur Normalen auftrifft. Beispiele:

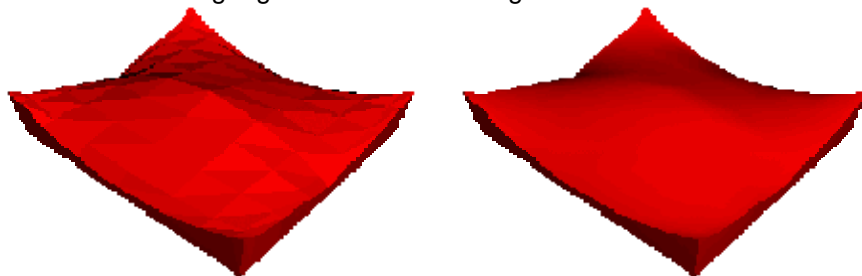
alpha [Grad]	0	30	45	60	90	120	180	360
cos(alpha)	1.0	0.87	0.71	0.5	0.0	-0.5	-1.0	1.0
Helligkeit	100%	87%	71%	50%	0%	0%	0%	100%

Bei Faces, die keine Normale haben, behilft man sich mit einer Konvention: Man stellt sich auf den zweiten Vertex $p[1]$ und richtet den linken Daumen nach dem ersten Vertex $p[0]$ und den Zeigefinger nach dem dritten Vertex $p[2]$. Das Face bekommt seine Normale auf die Seite auf die jetzt der angewinkelte Mittelfinger zeigt.

In DirectX3D und in OpenGL gibt man nicht jedem Face, sondern jedem Vertex eine Normale = "Vertex Normal". Ein Dreieck besitzt also drei parallele "Vertex Normals", was redundant und widersinnig ist bei einem einzelnen Dreieck.

Wenn jedoch drei oder mehr Dreiecke wie z.B. an einer Pyramidenspitze zusammenstoßen, dann kann man die am Punkt versammelten Normalen zu einer Durchschnittsnormalen mitteln, die die Ecke sehr gut repräsentiert. Der Vorteil zeigt sich bei `TriangleStrips` an Kanten, wo Dreiecke scharfkantig zusammenstoßen.

Anstatt des scharfen Farbsprungs bei schräger Beleuchtung → Flat Shading ergeben die gemittelten Normalen einen sanften Übergang → Gouraud-Shading.



Shading-Verfahren = Vortäuschen einer Tiefe durch Licht- und Schatteneffekte.

Gouraud-Shading von 1971 benannt nach dem Studenten Henri Gouraud (sprich: Guro).

Ähnlich: Phong-Shading, Metal Shading

Details siehe: [Face and Vertex Normal Vectors](#)