Computer Grafik Vorlesung

Dr. Arno Formella

gehalten vier-stündig mit zwei-stündigen Übungen im SS98

Dies ist das etwas unvollständige Vorlesungsskript, so wie es auch im Internet zur Verfügung gestellt wurde. Einige Abschnitte wurden von mir nicht in elektronischer Form erstellt, sondern liegen nur handschriftlich vor. Die Vorlesung wurde als Tafel-Kreide-Vorlesung gehalten. Kapitel 18 (RayTracing) gegen Ende der Vorlessung wurde aus meiner 93er Vorlesung recycled und ist deshalb hier nicht enthalten.

Contents

1 Motivation **5**
1.1 Was ist Computer Grafik? 5
1.2 Wer verwendet Computer Grafik? 5
1.3 Womit befasst sich Computer Grafik? 6
1.4 kurze Geschichte der Computer Grafik 6
1.5 Toy Story: einige Daten 7

2 Virtuelles Rastergrafik-Gerät: **7**
2.1 Mögliche Realisierung 7
2.2 Möglicher Strahlengang 8
2.3 andere Anzeigetechniken: 8
2.4 andere Rastergrafik-Ausgabegeräte: 9
2.5 wichtige Begriffe 9
2.6 Einfache Rastergrafik-Einheit (Grafik-Karte) 9
2.7 Pixel-Kodierung 9
2.8 Darstellung mit Farbtabelle 10

3 Darstellung von Liniensegmenten **10**
3.1 einige ¨vorhandene¨ Funktionen 10
3.2 Anforderungen 11
3.3 Welche Pixel sollen gesetzt werden? 11
3.4 Naives Programm: 11
3.5 Mögliche Beschreibungsformen für ein Segment 12
3.6 Diskretisierung der parametrisierten Form 12
3.7 Was ist mit Anforderung 3)? 12
3.8 Was ist mit Anforderung 5)? 13
3.9 Mittelpunkt-Entscheidungsalgorithmus (*Bresenham*) 13
3.10 Bresenham-Algorithmus 14

4 Darstellung von Kreisen **14**
4.1 Mögliche Beschreibungsformen für einen Kreis 14
4.2 Diskretisierung der parametrisierten Form 15
4.3 Mittelpunkt-Entscheidungsalgorithmus 16
4.4 Bresenham-Kreis-Algorithmus 17

5 Darstellung von Ellipsen **18**
5.1 Mögliche Beschreibungsformen für eine Ellipse 18
5.2 Achsenparallele Ellipse mit Zentrum im Koordinatenursprung 19
5.3 Diskretisierung der parametrisierten Form 20
5.4 Mittelpunkt-Entscheidungsalgorithmus 21
5.5 Bresenham-Ellipsen-Algorithmus 23
5.6 Weitere Ellipsendarstellungsmethoden 25

6 Zusammenfassung **25**

7 Darstellung von Polygonen **25**
7.1 Was ist ein Polygon? 25
7.2 Klassifizierung von Polygonen 26
7.3 Einfachheit 26
7.4 Orientierung 26
7.5 Schnittpunkt von Segmenten bzw. Strahl und Segment 27
7.6 Innen und Außen 29
7.7 Konvexität 30
7.8 Scan-Line-Prinzip 30
7.9 Einfachheit-Test 30
7.10 Konvexität-Test 30
7.11 Innen-Außen-Korrektheit 31
7.12 Konvexität-und-Einfachheit-Test 31

8 Zusammenfassung **32**

9 Füllen **32**
9.1 Füllen von Polygonen 32
9.2 Abtastlinien-Methode 32
9.3 Saatkorn-Methode 33

10 Clipping **34**
10.1 Clipping von Punkten 35
10.2 Clipping von Geradensegmenten 35
10.3 Algorithmus nach Cohen-Sutherland 35
10.4 Algorithmus nach Liang-Barsky 37
10.5 Clipping von Polygonen 39
10.5.1 Algorithmus nach Sutherland-Hodgeman 39
10.5.2 Algorithmus nach Weiler-Atherton 41

11 Geometrische Transformationen **41**
11.1 Analytische Geometrie 41
11.2 Transformationen 41
11.3 Affine Kombination 41
11.4 Affine Abbildungen 42
11.5 Homogene Koordinaten 43
11.6 Translation 43
11.7 Skalierung 43
11.8 Scherung 44
11.9 Spiegelung 45
11.10 Rotation 45
11.11 Trigonometrische Additionstheoreme 45
11.12 Rotation mit Quaternionen 45
11.13 Drehformel nach Hamilton 46
11.14 Nachweis der Korrektheit der Drehformel 46

12 Objektmodellierung **50**
12.1 Einfache geometrische Objekte . . . 50
12.2 Polygonale Objekte . . . 50
12.3 Oberfläche als Graph . . . 51
12.4 Innen und Außen . . . 51
12.5 Normalenvektorberechnung von fast planaren Polygonen . . . 52
12.5.1 Quadriken . . . 54
12.6 Kugel . . . 54
12.7 Ellipsoid . . . 55
12.8 Torus . . . 56
12.9 Superquadriken . . . 56
12.10 2-dimensionale Superquadriken . . . 56
12.11 3-dimensionale Superquadriken . . . 57

13 Kurven und gekrümmte Oberflächen **57**
13.1 Kurven . . . 57
13.2 Polynominterpolation . . . 58
13.3 Zusammengesetzte Kurven . . . 58
13.4 Eigenschaften zusammengesetzter Kurven . . . 58
13.5 C-Eigenschaft . . . 58
13.6 G-Eigenschaft . . . 58
13.7 Worin liegt der Unterschied? . . . 59
13.8 Splines . . . 59
13.9 Spline-Repräsentation . . . 59
13.10 kubische Splines . . . 60
13.11 Hermite-Splines . . . 60
13.12 Cardinal-Splines . . . 60
13.13 Bézier-Splines . . . 61
13.14 Kubische Bézier-Splines . . . 62
13.15 Konstruktionstechniken: . . . 62
13.16 Matrix-Schreibweise: . . . 63
13.17 B-Splines . . . 63
13.18 Klassifizierung der B-Splines . . . 63
13.19 uniforme B-Splines . . . 63
13.20 Beispiel: quadratischer B-Spline . . . 63
13.21 Bemerkungen . . . 63
13.22 Beispiel: kubischer B-Spline . . . 64
13.23 Beta-Splines . . . 64
13.24 Rationale Splines . . . 64
13.25 Beispiel für einen NURBS . . . 64
13.26 Darstellung von Splines . . . 65
13.27 Polynomauswertung . . . 65
13.28 Vorwärtsdifferenzen . . . 65
13.29 Unterteilungsmethode . . . 66
13.30 Umrechnung von kubischen Splines . . . 66

14 Komplexe Objekte **67**
14.1 Translationsobjekte . . . 67
14.2 Rotationsobjekte . . . 67
14.3 Gekrümmte Flächen . . . 67

15 Licht und Farbe **67**
15.1 Licht . . . 67
15.2 Menschliches Wahrnehmungssystem . . . 68
15.3 Intensität und Farbe . . . 69
15.4 Klassifizierung von Farbe . . . 69
15.5 RGB-Modell . . . 70
15.6 HSV-Modell . . . 70
15.7 YIQ-Modell . . . 71
15.8 CMY-Modell . . . 71
15.9 CNS-Modell . . . 72
15.10 Darstellung von Bildern . . . 72
15.11 Intensitätsdiskretisierung . . . 73
15.12 Gamma-Korrektur . . . 73
15.13 Anpassung an Ausgabegeräte . . . 74
15.14 Halftoning . . . 74
15.15 Dithering . . . 75
15.16 Reduktion der Auflösung . . . 75
15.17 Fehlerkorrekturverfahren . . . 76
15.18 Fehlerverteilungsverfahren . . . 76
15.19 Fehlerdiffusionsverfahren . . . 76

16 3-dimensionale Darstellung **77**
16.1 Koordinatentransformationen . . . 77
16.2 Kamera-Koordinatensystem . . . 78
16.3 Spezifikation einer Kamera . . . 78
16.4 Welt-Koordinaten in Kamera-Koordinaten . . . 78
16.5 Klassifizierung der Projektionen . . . 79
16.6 Kanonische Sichtvolumen . . . 80

17 Sichtbarkeitsberechnungen **82**
17.1 Elimination der Rückseiten . . . 82
17.2 Elimination nicht sichtbarer Kanten . . . 82
17.3 Elimination nicht sichtbarer Flächen . . . 82
17.4 Z-Puffer-Methode . . . 82
17.5 A-Puffer . . . 84
17.6 Tiefensortier-Verfahren . . . 84
17.7 Unterteilungsverfahren . . . 84
17.8 Zusammenfassung der Darstellungsarten . . . 84

18 Ray Tracing **85**
18.1 Prinzip . . . 85
18.2 Strahlberechnung . . . 85
18.3 Effekte . . . 85
18.4 Beschleunigungsmethoden . . . 85

1 Motivation

1.1 Was ist Computer Grafik?

Computer Grafik ist ein Teilbereich der grafischen Datenverarbeitung:

- Computer Grafik (*computer graphics*)
  Beschreibung von Objekten
  –¿ **2-dimensionale Bilder**

- (*computer vision*)
  2-dimensionale Bilder
  –¿ **Beschreibung von Objekten**

- Digitale Bildverarbeitung (*image processing*)
  2-dimensionale Bilder
  –¿ **2-dimensionale Bilder**

- Computer Animation (*computer animation*)
  Beschreibung von Objekten mit zeitlicher Veränderung
  –¿ **Folge von 2-dimensionalen Bildern**

1.2 Wer verwendet Computer Grafik?

- Unterhaltung
  - Film-Industrie
  - Computer-Spiele
  - Computer-Kunst
  - Virtual-Reality
- Technik
  - Computer Aided Design (CAD)
  - Visualisierung Messdaten
- Ausbildung
  - Visualisierung
  - Fahr- und Flugsimulatoren
- Werkzeug
  - Grafische Benutzeroberflächen
  - Werbeindustrie
  - (Geo-)Grafische Informationssysteme (GIS)
- Wissenschaft
  - Medizin (CT,NMR)
  - Visualisierung, Simulation
  - Forschungsgegenstand

1.3 Womit befasst sich Computer Grafik?

- Modellierung
  - Geometrie
  - Kamera
  - Licht
  - Material
  - Oberflächen
- Rendering
  - Effiziente Algorithmen
  - Darstellungsarten (*shading*)
    (sprites, wire-frame, flat shading, Gauroud shading, Phong shading, ray tracing, radiance, radiosity)
  - Spezialeffekte
- Interaktion
  - Ein/Ausgabe-Geräte
  - Werkzeuge

1.4 kurze Geschichte der Computer Grafik

2000 vChr orthografische Projektion

17. Jhd. Koordinatensysteme (Descartes)
Numerik, Physik, Optik (Newton)

1897 Oszilloskop (Braun)

1950-1970 Computer mit Vektor-Displays
(Kathodenstrahlröhre)

1964 Anfänge von CAD bei GM

1966 erstes Rastergrafik-Display

1980 Personal-Computer von IBM und Apple Macintosh

1993 1200×1200, 500K Dreiecke/Sekunde
36-bit Farbkodierung
Stereobilder
Texture mapping
alles bei 60 Hz

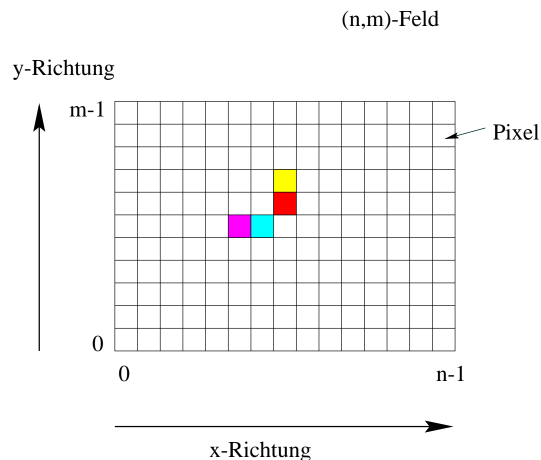
1995 vollständige Kinofilme

Zukunft? Holodeck ...

1.5 Toy Story: einige Daten

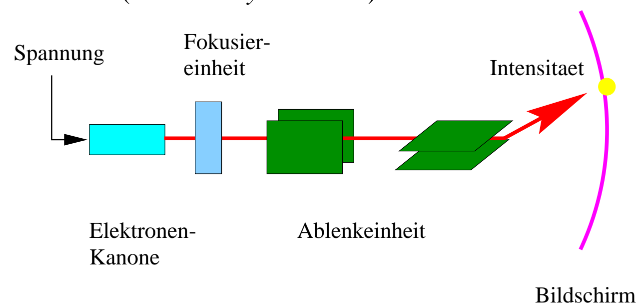
- ca. 79 min fertiger Film
- 114240 Einzelbilder (24 Bilder pro Sekunde)
- Auflösung: 1526×922 Pixel
- $114240 \times 1526 \times 922 = 161$ Mrd. Pixel
- ca. 600 GByte
- mehr als 400 Objektmodelle
- 34 TByte Renderman Eingabe-Dateien
- 1300 Renderman Shader
- 5.5 Mill. Zeilen Code
- ca. 25000 Storybord-Zeichnungen
- 300 Workstations (Sun)
- ca. 800000 CPU-Stunden Rechenzeit
- fast 4 Monate bloßes Rechnen

2 Virtuelles Rastergrafik-Gerät:



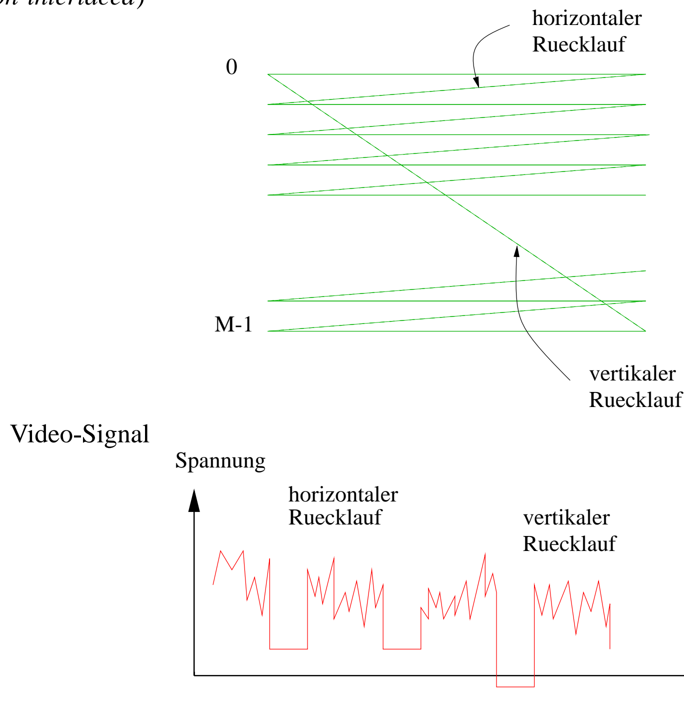
2.1 Mögliche Realisierung

als Elektronenstrahlröhre (*cathode-ray-tube CRT*):

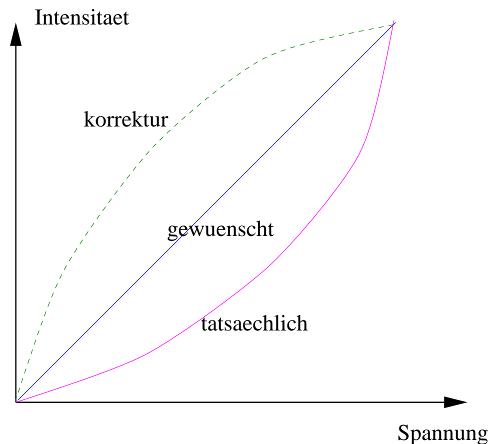


2.2 Möglicher Strahlengang

(*non-interlaced*)



Der Zusammenhang zwischen anliegender Spannung und Leuchtintensität des Monitors ist nicht linear:



was durch die sogenannte Gamma-Korrektur ausgeglichen werden kann.

1 Elektronenstrahl

–¿ monochromes (Schwarz/Weiß,Schwarz/Grün,Schwarz/Bernstein) Bild

3 Elektronenstrahlen

–¿ Farbbild

Additive Farbmischung: **Rot, Grün, Blau**

übliche Standards in der Fernsehwelt: NTSC, PAL, SECAM, HDTV

2.3 andere Anzeigetechniken:

- passive Liquid-Crystal-Displays (LCD: STN, DSTN)
- aktive Liquid-Crystal-Displays (TFT)

- Plasma-Bildschirme (PDP)
- Plastik-Bildschirme ??

Info-Punkte:

- Display-Workshop 1996[1]
- LC-Technologie[2]

2.4 andere Rastergrafik-Ausgabegeräte:

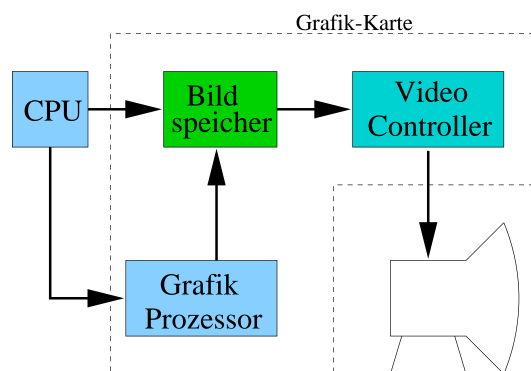
- Nadeldrucker (Matrixdrucker)
- Thermo-Drucker
- Tintenstrahldrucker
- Farb-Sublimationsdrucker
- Laserdrucker
- Fest-Tinte-Drucker

Subtraktive Farbmischung: **Gelb, Magenta, Cyan (Schwarz)**

2.5 wichtige Begriffe

- Auflösung (*resolution*)
- Breite-Höhen-Verhältnis (*aspect ratio*)
- Punktgröße (*spot size*)
- Bandbreite (*bandwidth*)
- Bildwiederholrate (*refresh rate*)

2.6 Einfache Rastergrafik-Einheit (Grafik-Karte)



2.7 Pixel-Kodierung

Einheit [bpp] = Bits per Pixel

- (*bit-map*): 1 bpp
  2 ¨Farben¨

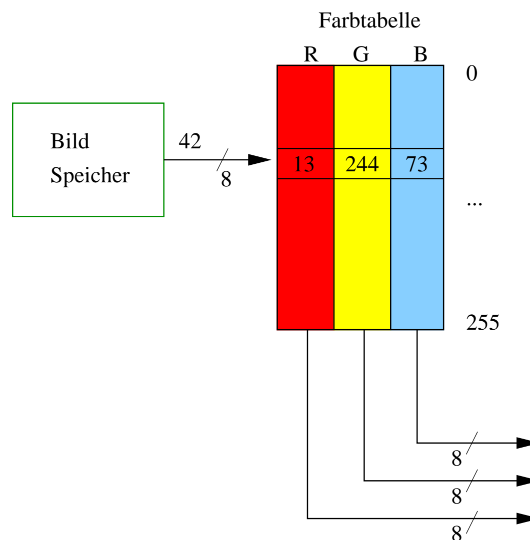
[1] See URL http://cordis.lu/esprit/src/tcsdispl.htm
[2] See URL http://www.garlic.com/sid/sid95/lctech.htm

- Graustufen (*grey scale*): 8 bpp
  256 ¨Farben¨

- Farbtabelle (*color map*): 8–16 bpp, indirekt
  256-65536 Farben

- (*true color*): 24 bpp
  16777216 Farben

2.8 Darstellung mit Farbtabelle



Auswahl einer ¨guten¨ Belegung der Farbtabelle?

3 Darstellung von Liniensegmenten

3.1 einige ¨vorhandene¨ Funktionen

Wir verfügen über die folgenden Funktionen:

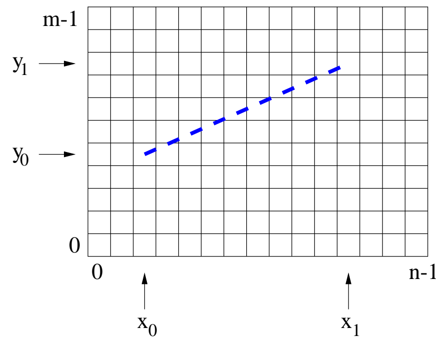
- `int Round(double x)`
  - rundet zur nächst gelegenen ganzen Zahl
  - x.5 wird sowohl bei positiven als auch bei negativen Zahlen aufgerundet,
    d.h. `Round(17.5)==18` und `Round(-17.5)==-17`

- `void SetPixel(int x, int y, int color)` setzt Pixel an Koordinate (x,y) auf die Farbwert `color`

- `type Abs(type x)` liefert Absolutbetrag vom entsprechenden Typ

- `type Max(type x, type y)` liefert Maximum vom entsprechenden Typ

3.2 Anforderungen



`DrawLine(x0,y0,x1,y1,color)`

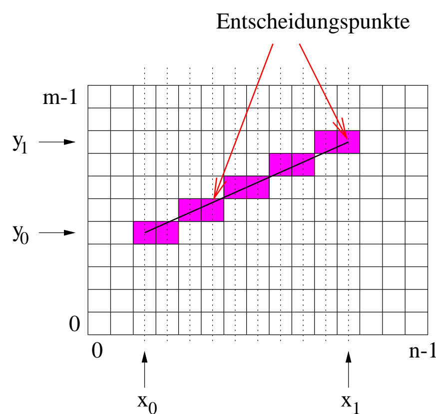
erwünschte Eigenschaften der Darstellung:

1. möglichst ¨gerade¨
2. Verlauf durch die Endpunkte
3. gleiche Helligkeit unabhängig von der Steigung

erwünschte Eigenschaften des Algorithmus:

1. robust
2. effizient
3. unabhängig von der Reihenfolge der Punkte

3.3 Welche Pixel sollen gesetzt werden?



3.4 Naives Programm:

```
DrawLine(
  int x0, int y0,
  int x1, int y1,
  int color
) {
  double y;
  int    x;

  for(x=x0; x!=x1+1; x++) {
    y = y0 + (x-x0)*(y1-y0)/(x1-x0)
```

```
    SetPixel(x, Round(y), color);
  }
}
```

- `x0==x1`, und tschüss ...
- wie sieht die Darstellung für |`x1-x0`|<|`y1-y0`| aus? ...
- brauchen wir wirklich floating point Operationen? ...

3.5 Mögliche Beschreibungsformen für ein Segment

- explizite Form
- parametrisierte Form
- implizite Form

3.6 Diskretisierung der parametrisierten Form

- Zerteilen des Intervalles [0,1] in `Max(|x1-x0|,|y1-y0|)` Intervalle.
- Setzen der Pixel an den diskreten Punkten.

```
DrawLine(
  int x0, int y0,
  int x1, int y1,
  int color
) {
  double x,y;
  double len;
  int    i;

  i = Max(Abs(x1-x0), Abs(y1-y0));

  SetPixel(x0, y0, color);
  len = i;
  while( i!=0 ) {
    x = x0 + i/len *(x1-x0);
    y = y0 + i/len *(y1-y0);
    i = i-1;
    SetPixel(Round(x), Round(y), color);
  }
}
```

Verfahren erfüllt zumindest Anforderungen 1, 2, 4 und 6

3.7 Was ist mit Anforderung 3)?

Verhältnis zwischen tatsächlicher Länge und Anzahl gesetzter Pixel liegt zwischen 1 und Wurzel(2),

also erscheinen Segmente verschiedener Steigung nicht gleich hell.

Da kann man bei 1 bpp nicht viel tun, bei Graustufen und Farbdisplays allerdings schon, siehe Aliasing.

Bemerkung: Durch eine andere Schrittweite im Parameterraum können auch weniger Punkte gezeichnet werden.

3.8 Was ist mit Anforderung 5)?

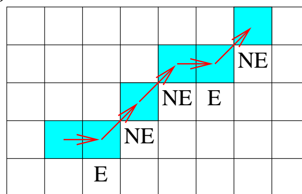
Das kommt darauf an ...!!

- Nachdenkzeit?
- Programmierzeit?
- Debugzeit?
- Laufzeit?

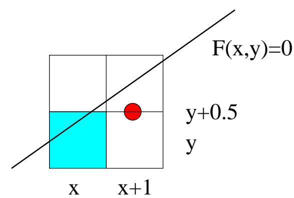
3.9 Mittelpunkt-Entscheidungsalgorithmus (*Bresenham*)

- Steigung liegt im Intervall [0,1]
- Verwenden nur integer-Arithmetik

Andere Steigungen: Übung.



Sukkzessive Entscheidungen nächstes Pixel im Osten (E) oder im Nordosten (NE) ?
Mittelpunkt zwischen diesen beiden Pixel – Einsetzen in die implizite Form




```
RestrictedDrawLine(
  int x0, int y0,
  int x1, int y1,
  int color
) {
  int x,y;

  y = y0;
  for( x=x0; x!=x1+1; x++) {
    SetPixel(x, y, color);
    if( F(x+1, y+0.5) > 0) {
      y = y+1;
    }
  }
}
```

Statt F(x,y) immer wieder komplett zu berechen nutze bereits berechnete Werte aus vorhergehender Iteration!

3.10 Bresenham-Algorithmus

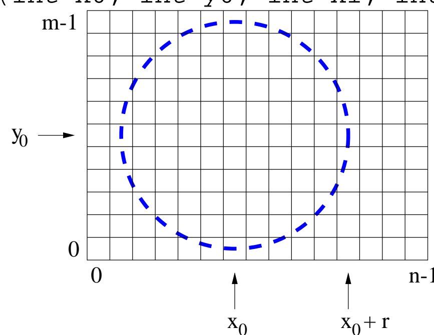
```
RestrictedDrawLine(
  int x0, int y0,
  int x1, int y1,
  int color
) {
  int dx, dy, d, incE, incNE, x, y;

  dx    = x1 - x0;
  dy    = y1 - y0;
  d     = 2*dy - dx;
  incE  = 2*dy;
  incNE = 2*(dy - dx);
  y     = y0;
  for (x=x0; x!=x1+1; x++) {
    SetPixel(x, y, color);
    if (d>0) {
      d = d + incNE;
      y = y + 1;
    }
    else {
      d = d + incE;
    }
  }
}
```

Implementierung der Fälle für andere Steigungen liefert kompletten Darstellungsalgorithmus.

4 Darstellung von Kreisen

Wir können bereits Liniensegmente darstellen:

`void DrawLine(int x0, int y0, int x1, int y1, int color)`



`DrawCircle(x0,y0,r,color)`

Gleiche Vorgehensweise wie bei Liniensegmenten!

4.1 Mögliche Beschreibungsformen für einen Kreis

- explizite Form

$$y = f(x)$$

$$f(x) = \begin{cases} y_0 + \sqrt{r^2 - (x - x_0)^2} \\ y_0 - \sqrt{r^2 - (x - x_0)^2} \end{cases}$$
$$\text{und} \quad |x - x_0| <= r$$

- parametrisierte Form

$$\begin{aligned} x &= x(t) \\ y &= y(t) \end{aligned}$$

$$\begin{aligned} x(t) &= x_0 + r \cdot \cos(t) \\ y(t) &= y_0 + r \cdot \sin(t) \\ & \text{und} \quad t \in [0, 2\pi[\end{aligned}$$

- implizite Form

$$F(x,y) = 0$$

$$F(x,y) = (x - x_0)^2 + (y - y_0)^2 - r^2$$

und es gilt:

$$F(x,y) \quad \begin{cases} < 0 & \text{innerhalb} \\ = 0 & \text{auf} \\ > 0 & \text{außerhalb} \end{cases}$$

4.2 Diskretisierung der parametrisierten Form

- Zerteilen des Intervalles $[0{,}2\pi[$ in **Wieviele?** Intervalle.
- Setzen der Pixel an den diskreten Punkten.

Wieviele?

- Übergabe als Parameter `n`
- `n` zu groß: Pixel werden doppelt gesetzt
  `n` zu klein: der Kreis hat Lücken

```
DrawCircle(
  int x0, int y0,
  int r,  int n,
  int color
) {
  double x,y;
  int    i=0;

  while( i < n ) {
    x = x0 + r*cos(2.0*pi*i/n);
    y = y0 + r*sin(2.0*pi*i/n);
```

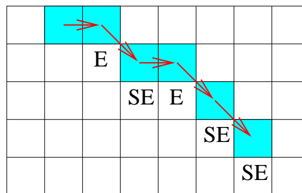
```
    SetPixel(Round(x), Round(y), color);
    i++;
  }
}
```

Bemerkung: Statt einzelne Pixel zu setzen, kann man zwischen aufeinanderfolgenden Pixeln Liniensegmente zeichnen (–¿ Lücken werden vermieden).

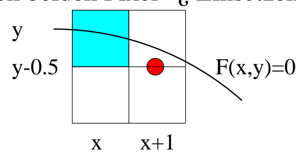
4.3 Mittelpunkt-Entscheidungsalgorithmus

- Steigung liegt im Intervall [0,-1]
  d.h. nur ein Oktant wird gezeichnet
- Verwenden nur integer-Additionen und -Subtraktionen in der inneren Schleife
- Nehmen Kreis liegt im Ursprung an und verschieben bei `SetPixel()`

andere Oktanten: Übung.



Sukkzessive Entscheidungen nächstes Pixel im Osten (E) oder im Südosten (SE) ?
Mittelpunkt zwischen diesen beiden Pixel –¿ Einsetzen in die implizite Form



Wann beträgt die Steigung -1?

wenn `x==y`!

```
DrawCircleOctant(
  int x0, int y0,
  int r
  int color
) {
  int x,y;

  x = 0;
  y = r;
  SetPixel(x+x0, y+y0, color);
  while( x < y ) {
    if( F(x+1, y-0.5) >= 0) y--;
    x++;
    SetPixel(x+x0, y+y0, color);
  }
}
```

Statt F(x,y) immer wieder komplett zu berechen nutze bereits berechnete Werte aus vorhergehender Iteration!

- Fall E:

$$\begin{aligned} & F(x+2, y-0.5) - F(x+1, y-0.5) \\ = \quad & (x+2)^2 + (y-0.5)^2 - r^2 \\ & - (x+1)^2 + (y-0.5)^2 - r^2 \\ = \quad & 2x+3 \\ = \quad & 2(x+1)+1 \end{aligned}$$

- Fall SE:

$$\begin{aligned} & F(x+2, y-1.5) - F(x+1, y-0.5) \\ = \quad & (x+2)^2 + (y-1.5)^2 - r^2 \\ & - (x+1)^2 + (y-0.5)^2 - r^2 \\ = \quad & 2x-2y+5 \\ = \quad & 2(x+1) - 2(y-1) + 1 \end{aligned}$$

- Zu Beginn:

$$\begin{aligned} & F(1, r-0.5) \\ = \quad & 1^2 + (r-0.5)^2 - r^2 \\ = \quad & 5/4 - r \end{aligned}$$

Wählen 1-r (statt mit 4 zu multiplizieren).

Macht man dabei einen groben Fehler?: Übung.

4.4 Bresenham-Kreis-Algorithmus

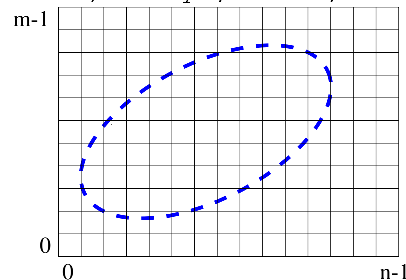
```
DrawCircleOctant(
  int x0, int y0,
  int r,
  int color
) {
  int d, incX, incY, x, y;

  d    = 1 - r;
  incX = 0;
  incY = -2*r;
  x    = 0;
  y    = r;
  SetPixel(x+x0, y+y0, color);
  while ( x < y ) {
    if (d >= 0) {
      y    -= 1;
      incY += 2;
      d    += incY;
    }
```

```
    x    += 1;
    incX += 2;
    d    += incX + 1;
    SetPixel(x+x0, y+y0, color);
  }
}
```

5 Darstellung von Ellipsen

Wir können bereits Linien und Kreise darstellen:

```
   void DrawLine(int x0, int y0, int x1, int y1, int color)
void DrawCircle(int x0, int y0, int r, int color)
```

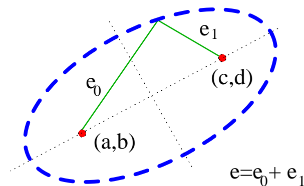



```
DrawEllipse(..., color)
```

Welche Parameter übergibt man?

5.1 Mögliche Beschreibungsformen für eine Ellipse

- implizite Form



Gegeben: die beiden Fokuspunkte (a,b) und (c,d) sowie die Fokusdistanz e:

$$F(x,y) = 0$$

$$F(x,y) = \sqrt{(x-a)^2 + (y-b)^2} + \sqrt{(x-c)^2 + (y-d)^2} - e$$

und $(c-a)^2 + (d-b)^2 < e^2$

und es gilt:

$$F(x,y) \begin{cases} < 0 & \text{innerhalb} \\ = 0 & \text{auf} \\ > 0 & \text{außerhalb} \end{cases}$$

- explizite Form

welche ¨etwas¨ komplex ist, am besten man verwendet ein Algebrasystem (z.B. Maple Waterloo Maple Inc.[3]), um die explizite Form aus der impliziten Form zu bestimmen.

[3] See URL http://www.maplesoft.com/home.html

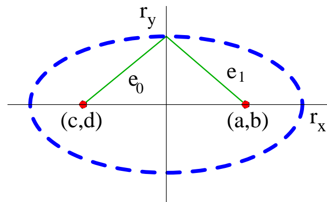
$$y \quad = \quad f(x)$$

$$f(x) = 1/(2\cdot(-4e^2+4d^2+4b^2-8db))\cdot$$
$$\big[-4be^2+4d^3+8xcb+8dxa-8xab-8xcd+$$
$$4b^3+4c^2d-4c^2b-4da^2+4a^2b-4db^2-4de^2-4d^2b$$
$$\pm 4\cdot\big(4e^2x^2c^2-4e^2xa^3-4be^2d^3-2e^2c^2a^2-4e^2xc^3+$$
$$4e^2x^2a^2-4b^3e^2d+4be^4d+6b^2e^2d^2+$$
$$2c^2d^2e^2+2d^2a^2e^2+4e^4xa+4d^2x^2e^2+4b^2x^2e^2+$$
$$e^6-2b^2e^4+b^4e^2+d^4e^2-2d^2e^4-4e^4x^2-2e^4a^2-2e^4c^2+$$
$$e^2c^4+e^2a^4+8be^2xcd+2b^2e^2a^2+2b^2e^2c^2-4b^2e^2xc+$$
$$8be^2dxa-4b^2e^2xa-4be^2c^2d-4be^2da^2-8e^2x^2ca+$$
$$4e^4xc-4d^2xae^2-4xcd^2e^2+4e^2xca^2+4e^2c^2xa-8dbx^2e^2\big)^{1/2}\big]$$

und $\sqrt{(c-a)^2+(d-b)^2} < e$

- parametrisierte Form
  siehe Rotationstransformationen

5.2 Achsenparallele Ellipse mit Zentrum im Koordinatenursprung



Mit $b = 0$, $c = -a$, und $d = 0$ und wegen

$$e = 2\cdot\sqrt{a^2+r_y^2} \quad \text{sowie} \quad e = (r_x - a) + r_x + a$$

ergibt sich

$$1/4\cdot e^2 = a^2 + r_y^2 \quad \text{sowie} \quad 1/4\cdot e^2 = r_x^2$$

Und man kann a angeben als

$$a \quad = \quad \sqrt{r_x^2 - r_y^2}$$

und alles in die implizite Gleichung einsetzen:

$$\sqrt{(x-\sqrt{r_x^2-r_y^2})^2+y^2}+\sqrt{(x+\sqrt{r_x^2-r_y^2})^2+y^2}-2r_x=0$$

Diese Gleichung vereinfacht man leicht (mit Maple!) zu:

$$x = \sqrt{(r_y^2-y^2)}\cdot r_x/r_y$$

oder auch in anderer Form

$$x^2/r_x^2 + y^2/r_y^2 = 1$$

Damit erhalten wie die implizite Form zu:

$$F(x,y) = r_y^2 x^2 + r_x^2 y^2 - r_x^2 r_y^2$$

Und eine Verschiebung liefert:

$$(x-x_0)^2/r_x^2 + (y-y_0)^2/r_y^2 = 1$$

Hier können wir nun auch die parametrisierte Form schreiben:

$$\begin{aligned} x &= x(t) \\ y &= y(t) \end{aligned}$$

$$\begin{aligned} x(t) &= x_0 + r_x \cdot \cos(t) \\ y(t) &= y_0 + r_y \cdot \sin(t) \\ & \text{und} \quad t \in [0, 2\pi[\end{aligned}$$

Stimmt diese Gleichung?: Übung.
Als Aufrufparameter zum Zeichnen einer achsenparallelen Ellipse haben wir also:
`DrawEllipse(x0,y0,rx,ry,color)`

5.3 Diskretisierung der parametrisierten Form

- Zerteilen des Intervalles [0,2π[in **Wieviele?** Intervalle.
- Setzen der Pixel an den diskreten Punkten.

Wieviele?

- Übergabe als Parameter `n`
- `n` zu groß: Pixel werden doppelt gesetzt
  `n` zu klein: der Ellipse hat Lücken

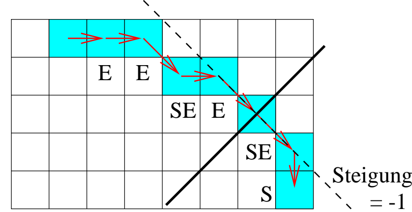
```
DrawEllipse(
  int x0, int y0,
  int rx, int ry,
  int n,
  int color
) {
  double x,y;
  int    i=0;

  while( i < n ) {
    x = x0 + rx*cos(2.0*pi*i/n);
    y = y0 + ry*sin(2.0*pi*i/n);
    SetPixel(Round(x), Round(y), color);
    i++;
  }
}
```

Bemerkung: Statt einzelne Pixel zu setzen, kann man zwischen aufeinanderfolgenden Pixeln Liniensegmente zeichnen (–¿ Lücken werden vermieden).

Bemerkung: Da man sowieso schon mit sin und cos hantiert, kann man eine Drehung der Ellipse durch entsprechende Rotationstransformationen erreichen.

5.4 Mittelpunkt-Entscheidungsalgorithmus

Ähnlich wie beim Zeichnen eines Kreises!

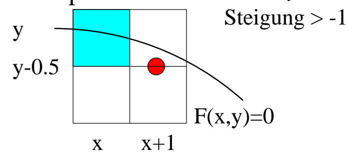


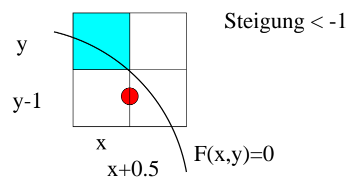
Sukkzessive Entscheidungen bis die Steigung -1 erreicht! nächstes Pixel im Osten (E) oder im Südosten (SE) ?

Dann:

Sukkzessive Entscheidungen nächstes Pixel im Südosten (SE) oder im Süden (S) ?

Mittelpunkt zwischen den entsprechenden Pixeln –> Einsetzen in die implizite Form





Wann beträgt die Steigung -1?

Bemühen wir die Tangentengleichung:

$$(x - x_0) \cdot \underbrace{\frac{\partial F(x_0, y_0)}{\partial x}}_{= dx} + (y - y_0) \cdot \underbrace{\frac{\partial F(x_0, y_0)}{\partial y}}_{= dy} = 0$$

Transformation dieser impliziten Geradengleichung in die explizite Form:

$$\begin{aligned} 0 &= (x - x_0) \cdot dx + (y - y_0) \cdot dy \\ 0 &= xdx - x_0 dx + ydy - y_0 dy \\ y &= y_0 - x_0 \cdot \frac{dx}{dy} - x \cdot \frac{dx}{dy} \end{aligned}$$

Die Steigung ist also

$$-\frac{dx}{dy} = -\frac{\frac{\partial F(x_0, y_0)}{\partial x}}{\frac{\partial F(x_0, y_0)}{\partial y}}$$

d.h. die Steigung ist gleich -1, wenn für die partiellen Ableitungen gilt:

$$\partial F / \partial x = \partial F / \partial y$$

Berechnen wir dies:

$$\begin{aligned} F(x,y) &= r_y^2 x^2 + r_x^2 y^2 - r_x^2 r_y^2 \\ \partial F(x,y)/\partial x &= 2 r_y^2 x \\ \partial F(x,y)/\partial y &= 2 r_x^2 y \end{aligned}$$

Überraschung: wenn wir `rx=ry=r` wählen, entspricht dies genau der Bedingung beim Kreiszeichnen: `x==y`

```
DrawEllipseQuadrant(
  int x0, int y0,
  int rx, int ry,
  int color
) {
  int x,y;

  x = 0;
  y = ry;
  SetPixel(x+x0, y+y0, color);
  while( 2*ry*ry*x /lt; 2*rx*rx*y ) {
    if( F(x+1, y-0.5) >= 0) y--;
    x++;
    SetPixel(x+x0, y+y0, color);
  }
  while( y >= 0 ) {
    if( F(x+0.5, y-1) <= 0) x++;
    y--;
    SetPixel(x+x0, y+y0, color);
  }
}
```

Statt F(x,y) immer wieder komplett zu berechen nutze bereits berechnete Werte aus vorhergehender Iteration!

- erster Oktant:
  - Fall E:

$$\begin{aligned} & F(x+2, y-0.5) - F(x+1, y-0.5) \\ &= r_y^2(x+2)^2 + r_x^2(y-0.5)^2 - r_x^2 r_y^2 \\ &\quad - r_y^2(x+1)^2 - r_x^2(y-0.5)^2 + r_x^2 r_y^2 \\ &= 2r_y^2 x + 3r_y^2 \\ &= 2r_y^2(x+1) + r_y^2 \end{aligned}$$

  - Fall SE:

$$\begin{aligned} & F(x+2, y-1.5) - F(x+1, y-0.5) \\ &= r_y^2(x+2)^2 + r_x^2(y-1.5)^2 - r_x^2 r_y^2 \end{aligned}$$

$$
\begin{aligned}
& \quad - r_y^2(x+1)^2 - r_x^2(y-0.5)^2 + r_x^2 r_y^2 \\
= & \quad 2r_y^2 x + 3r_y^2 - 2r_x^2 y + 2r_x^2 \\
= & \quad 2r_y^2(x+1) + r_y^2 - 2r_x^2(y-1)
\end{aligned}
$$

- zweiter Oktant:
  - Fall SE:

$$
\begin{aligned}
& F(x+1.5, y-2) - F(x+0.5, y-1) \\
& \quad = \quad r_y^2(x+1.5)^2 + r_x^2(y-2)^2 - r_x^2 r_y^2 \\
& \quad \quad \quad - r_y^2(x+0.5)^2 - r_x^2(y-1)^2 + r_x^2 r_y^2 \\
& \quad = \quad 2r_y^2 x + 2r_y^2 - 2r_x^2 y + 3r_x^2 \\
& \quad = \quad 2r_y^2(x+1) - 2r_x^2(y-1) + r_x^2
\end{aligned}
$$

  - Fall S:

$$
\begin{aligned}
& F(x+0.5, y-2) - F(x+0.5, y-1) \\
& \quad = \quad r_y^2(x+0.5)^2 + r_x^2(y-2)^2 - r_x^2 r_y^2 \\
& \quad \quad \quad - r_y^2(x+0.5)^2 - r_x^2(y-1)^2 + r_x^2 r_y^2 \\
& \quad = \quad - 2r_x^2 y + 3r_x^2 \\
& \quad = \quad - 2r_x^2(y-1) + r_x^2
\end{aligned}
$$

- Zu Beginn:

$$
\begin{aligned}
& F(1, r_y - 0.5) \\
& \quad = \quad r_y^2 + r_x^2(r_y - 0.5)^2 - r_x^2 r_y^2 \\
& \quad = \quad r_y^2 - r_x^2 r_y + 1/4 \cdot r_x^2
\end{aligned}
$$

  Auch hier runden wir statt mit 4 zu multiplizieren.

  Macht man dabei einen groben Fehler?: Übung.

- Übergang vom ersten zum zweiten Quadranten:
  Werten `F(x+0.5,y-1)` bezüglich der zuletzt gesetzten Position aus.

$$
\begin{aligned}
& F(x+0.5, y-1) \\
& \quad = \quad r_y^2(x+0.5)^2 + r_x^2(y-1)^2 - r_x^2 r_y^2
\end{aligned}
$$

  Bemerkung: hier könnten wir die Zeichenrichtung umdrehen und an der x-Achse zu zeichnen beginnen. Dies würde die Berechnungen zwischen den Quadranten etwas vereinfachen.

5.5 Bresenham-Ellipsen-Algorithmus

```
DrawEllipseQuadrant(
  int x0, int y0,
  int rx, int ry,
```

```
  int color
) {
  int d;
  int x,    y;
  int rx2,  ry2;
  int incX, incY;

  rx2  = rx*rx;
  ry2  = ry*ry;
  d    = Round(ry2-rx2*ry+0.25*rx2);
  incX = 0;
  incY = 2*rx2*ry;
  x    = 0;
  y    = ry;

  SetPixel(x+x0, y+y0, color);
  /* while ( 2*ry2*x < 2*rx2*y ) { */
  while ( incX < incY ) {
    if (d >= 0) {
      y    -= 1;
      incY -= 2*rx2;
      d    -= incY;
    }
    x    += 1;
    incX += 2*ry2;
    d    += incX + ry2;
    SetPixel(x+x0, y+y0, color);
  }
  d = Round(ry2*(x+0.5)*(x+0.5)+
            rx2*(y-1)  *(y-1)-
            rx2*ry2);
  while ( y > 0 ) {
    if (d <= 0) {
      x    += 1;
      incX += 2*ry2;
      d    += incX;
    }
    y    -=1;
    incY -= 2*rx2;
    d    += rx2 - incY;
    SetPixel(x+x0, y+y0, color);
  }
}
```

Es ist sehr mühsam diesen Bresenham-Ellipsen-Algorithmus auf rotierte Ellipsen anzuwenden, obwohl prinzipiell möglich.

Es treten folgende Schwierigkeiten auf:

- Einteilung in acht Oktanten erfordert Bestimmung der Punkte, an denen die Steigung 0, 1, -1 oder unendlich beträgt,
  diese liegen i.A. nicht alle genau auf Pixelwerten (–¿ Runden).
- Die impliziten Gleichungen (Ellipse sowie partielle Ableitungen) enthalten Wurzeln, und können deshalb nicht mehr (einfach) mit ganzzahliger Arithmetik inkrementell berechnet werden.

5.6 Weitere Ellipsendarstellungsmethoden

- Abtastlinien-Algorithmus (scanline-Algorithmus) (*Übung!*)
- Differentielle diskretisierte Parametermethode erster sowie zweiter Ordnung (kommt sobald Rotationstransformationen bekannt sind).

  benötigt (einfache) Gleitkommaarithmetik in der innerern Schleife, zeichnet aber mindestens zwei Nachbarpixel zu jedem Pixel (was für Füllalgorithmen u.U. wichtig ist).

6 Zusammenfassung

Die vorgestellten Algorithmen heißen DDA-Algorithmen (*digital differential analyser*).

Prinzipielle Form:

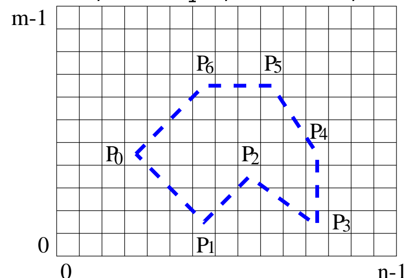
```
Initialisierung;

while ( Bedingung==TRUE ) {

  Aktion;

  if ( Entscheidung==SO ) {

     Inkrementelle Bedingungsaktualisierung;
     Inkrementelle Entscheidungsaktualisierung;
  }
  else {

     Inkrementelle Bedingungsaktualisierung;
     Inkrementelle Entscheidungsaktualisierung;
  }
}
```

Korrektheitsbeweise z.B. durch Induktion (siehe *Übung!*).

7 Darstellung von Polygonen

Wir können bereits Liniensegmente darstellen:

```
void DrawLine(int x0, int y0, int x1, int y1, int color)
```




```
DrawPolygon(point_list,color)
```

7.1 Was ist ein Polygon?

- zyklisch geordnete Liste von Punkten
  d.h. jeder Punkte hat genau einen Vorfahren und genau einen Nachfolger

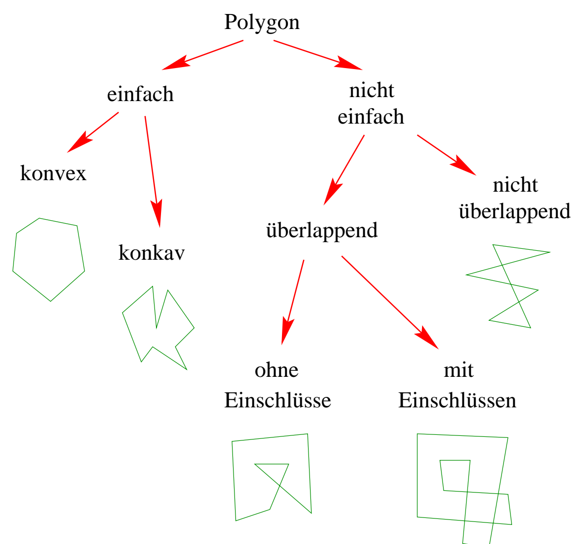
- Länge mindestens 3
- Punkte werden mit Geradensegmenten verbunden

Problem: Was passiert, wenn zwei Punkte identisch sind?
Bezeichnen:

- Punkte der Liste als **Ecken**
  (obwohl u.U. gar keine ¨Ecke¨ zu sehen)
- Verbindungssegmente als **Kanten**
- identische Ecken, die adjazent zueinander sind, als **Mehrfachecken**

Zeichnen ein Polygon durch Zeichnen der Kanten.

7.2 Klassifizierung von Polygonen



7.3 Einfachheit

einfach – nicht einfach:

Polygon ist einfach, wenn zwei Kanten höchstens eine Ecke gemeinsam haben und keine zwei Ecken identisch sind; sonst nicht einfach.

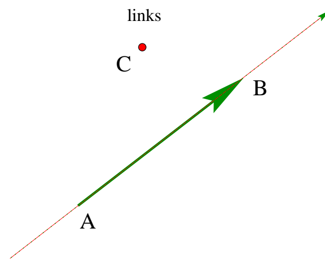
Test auf Einfachheit folgt, sobald noch einige ¨Details¨ geklärt sind.
Überlappend? Einschlüsse?

Dazu müssen wir erst wissen, was Innen und Außen ist.

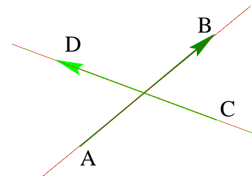
Vereinbarung: wenn nicht anders vermerkt, sind im Folgenden zwei Ecken stets verschieden.

7.4 Orientierung

von drei Punkten bzw. von einem Punkt und einem Segment



7.5 Schnittpunkt von Segmenten bzw. Strahl und Segment



1. Methode

Parametrische Form der beiden Geraden:

$$\begin{aligned} P(t) &= A + t \cdot (B - A) = A + tX \\ P(s) &= C + s \cdot (D - C) = C + sY \end{aligned}$$

Setzen $P(t) = P(s)$ und bestimmen s und t:

$$\begin{aligned} P(t) &= P(s) \\ A + tX &= C + sY \end{aligned}$$

oder als Koordinatengleichungen:

$$\begin{aligned} tx_0 &= (c_0 - a_0) + sy_0 \\ tx_1 &= (c_1 - a_1) + sy_1 \end{aligned}$$

Eliminieren von t und weiteres Umformen:

$$\begin{aligned} \frac{x_0}{x_1} \cdot [(c_1 - a_1) + sy_1] &= (c_0 - a_0) + sy_0 \\ x_0 \cdot (c_1 - a_1) + sx_0y_1 &= x_1 \cdot (c_0 - a_0) + sx_1y_0 \\ s \cdot (x_0y_1 - x_1y_0) &= x_1 \cdot (c_0 - a_0) - x_0 \cdot (c_1 - a_1) \\ &= x_1c_0 - x_0c_1 + x_0a_1 - x_1a_0 \\ s \cdot \begin{vmatrix} x_0 & y_0 \\ x_1 & y_1 \end{vmatrix} &= \begin{vmatrix} c_0 & x_0 \\ c_1 & x_1 \end{vmatrix} + \begin{vmatrix} x_0 & a_0 \\ x_1 & a_1 \end{vmatrix} \end{aligned}$$

Ist die Determinate der beiden Spaltenvektoren X und Y ungleich 0; erhalten wir:

$$s = \frac{\begin{vmatrix} c_0 & x_0 \\ c_1 & x_1 \end{vmatrix} + \begin{vmatrix} x_0 & a_0 \\ x_1 & a_1 \end{vmatrix}}{\begin{vmatrix} x_0 & y_0 \\ x_1 & y_1 \end{vmatrix}}$$

und in analogerweise

$$t = \frac{\begin{vmatrix} c_0 & y_0 \\ c_1 & y_1 \end{vmatrix} + \begin{vmatrix} y_0 & a_0 \\ y_1 & a_1 \end{vmatrix}}{\begin{vmatrix} x_0 & y_0 \\ x_1 & y_1 \end{vmatrix}}$$

Damit sich die Segmente (nicht nur die Geraden) schneiden, muss außerdem gelten:

$$0 \leq s \leq 1$$
$$0 \leq t \leq 1$$

Ist die Determinante gleich 0, so sind die beiden Geraden der Segmente parallel oder sogar identisch.

Wie unterscheidet man die beiden Fälle? Übung!

2. Methode

Implizite Form der Geraden des einen Segments

$$(x - a_0)(b_1 - a_1) - (y - a_1)(b_0 - a_0) = 0$$

parametrisierte Form der Geraden des anderen Segments

$$x(s) = c_0 + s \cdot (d_0 - c_0)$$
$$y(s) = c_1 + s \cdot (d_1 - c_1)$$

Einsetzen der parametrisierten Gleichungen in die implizite Gleichung

$$(c_0 - a_0 + s \cdot (d_0 - c_0))(b_1 - a_1) - (c_1 - a_1 + s \cdot (d_1 - c_1))(b_0 - a_0) = 0$$

und umformen:

$$s \cdot [(d_0 - c_0)(b_1 - a_1) - (d_1 - c_1)(b_0 - a_0)]$$
$$= (c_1 - a_1)(b_0 - a_0) - (c_0 - a_0)(b_1 - a_1)$$

und somit mit Determinaten

$$s \cdot \begin{vmatrix} d_0 - c_0 & b_0 - a_0 \\ d_1 - c_1 & b_1 - a_1 \end{vmatrix} = \begin{vmatrix} a_0 - b_0 & c_0 - a_0 \\ a_1 - b_1 & c_1 - a_1 \end{vmatrix}$$

Und es ergeben sich die gleichen Resultate wie bei der ersten Methode (*durch entsprechendes Umformen*):

$$s = \frac{\begin{vmatrix} b_0 - a_0 & c_0 - a_0 \\ b_1 - a_1 & c_1 - a_1 \end{vmatrix}}{\begin{vmatrix} d_0 - c_0 & b_0 - a_0 \\ d_1 - c_1 & b_1 - a_1 \end{vmatrix}}$$

und analogerweise

$$t = \frac{\begin{vmatrix} d_0 - c_0 & c_0 - a_0 \\ d_1 - c_1 & c_1 - a_1 \end{vmatrix}}{\begin{vmatrix} d_0 - c_0 & b_0 - a_0 \\ d_1 - c_1 & b_1 - a_1 \end{vmatrix}}$$

Auch hier überprüft man

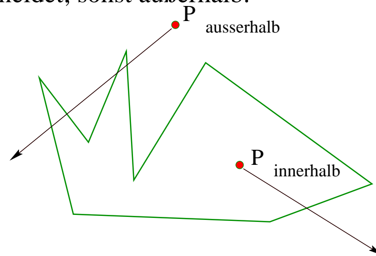
$$0 \leq s \leq 1$$
$$0 \leq t \leq 1$$

Schnittpunkt zwischen Segment und Strahl ist einfach:
–¿ man überprüft auf entsprechencd positiven Parameter!

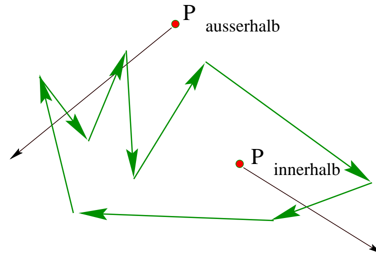
7.6 Innen und Außen

Sei P ein Punkt der Ebene. Zwei mögliche Varianten:

1. P liegt innerhalb des Polygons, wenn ein Strahl beginnend bei P eine **ungerade** Anzahl von Kanten schneidet; sonst außerhalb.



2. P liegt innerhalb des Polygons, wenn die Windungszahl von P **ungleich 0** ist; sonst außerhalb.



Die Windungszahl ist die Summe der Orientierungen von P bezüglich aller Kanten, die von einem Strahl beginnend bei P getroffen werden.

Vorsicht: Schneidet der Strahl eine Ecke, so wird natürlich wie folgt verfahren:

1. liegen beide adjazenten Ecken auf einer Seite des Strahls, wird kein Schnittpunkt (kein Summand für die Windungszahl) gezählt;

2. liegt je eine adjazente Ecke auf einer Seite des Strahls, wird nur ein Schnittpunkt (ein Summand für die Windungszahl) gezählt;

3. liegt eine Kante (Folge von Kanten) auf dem Strahl, wird wie in 1. bzw. 2. mit der ersten folgenden, nicht auf dem Strahl liegenden Kante verfahren.

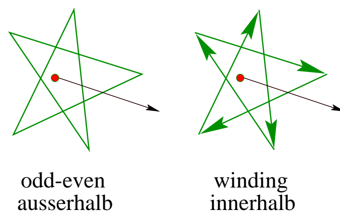
Der Beweis, dass es egal ist, welchen Strahl man für den Test heranzieht, folgt später.

Im praktischen Fall verwendet man eine Koordinatenrichtung als Strahl.

Wie behandelt man Mehrfachecken? Übung!

Insbesondere liegen Punkte auf dem Rand weder innen noch außen.Unterscheiden sich beide Definitionen?

Beispiel:



odd-even
ausserhalb

winding
innerhalb

überlappend – nicht überlappend

Ein Polygon ist nicht überlappend, wenn für jeden Punkt im Innern, dessen Windungszahl entweder gleich 1 oder -1 ist; sonst überlappend.

Einschlüsse – keine Einschlüsse

Ein Polygon hat keine Einschlüsse, wenn das Gebiet der Punkte mit Windungszahl 0 zusammenhängend ist.

7.7 Konvexität

Ist ein Polygon *einfach*, so sind die folgenden Aussagen äquivalent:

1. Bei der Berechnung der Orientierung je drei aufeinanderfolgender Ecken - entsprechend der Ordnung der Punktliste - tritt während eines Umlauf um das Polygon kein Vorzeichenwechsel auf.

2. Jedes Geradensegment zwischen zwei beliebigen Punkten aus der Fläche des Polygons liegt vollständig im Innern des Polygons.

3. Für je zwei beliebige adjazente Ecken liegen alle anderen Ecken in nur einer der beiden Halbebenen, die von der Kante zwischen den adjazenten Ecken erzeugt werden.

konvex – konkav

Ein einfaches Polygon, welches diese Aussagen erfüllt, ist konvex; sonst konkav.

Ist das Polygon nicht einfach, so sind die Aussagen nicht äquivalent!
Beweis der Äquivalenz von 1. und 3.: (*Wird noch eingefügt!*)

7.8 Scan-Line-Prinzip

(oder auch Sweep-Line-Prinzip)

Bemerkung: computational geometry ist ein wichtiges Hilfsmittel für computer graphics!

Sei **Q** eine objekt- und problemabhängige *Folge von Haltepunkten* in sortierter Koordinaten-Richtung.

Sei **L** die leere Liste.
`while` **Q** nicht leer `do`

wähle nächsten Haltepunkt aus **Q** und entferne ihn aus **Q**;
aktualisiere **L** und gibt (problemabhängige) Teilantwort aus.

`od`

Nutzen das Scan-Line-Prinzip um zu entscheiden, ob ein Polygon einfach oder nicht einfach ist.

7.9 Einfachheit-Test

Sweep-Line-Algorithmus

(*Wird noch eingefügt!*)

7.10 Konvexität-Test

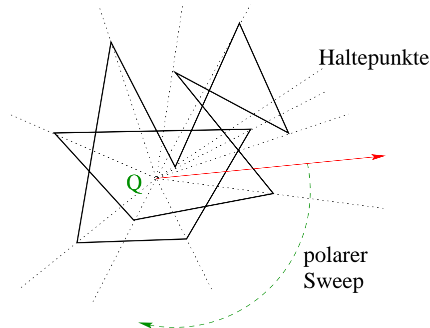
Annahme: Polygon ist einfach.

Teste Bedingung 1., d.h. ob kein Vorzeichenwechsel der Orientierung dreier aufeinanderfolgender Ecken bei einem Umlauf um das Polygon vorkommt, in linearer Zeit.

Aufsummieren der Winkeldifferenzen (mit Test auf nur positive Differenzen) bei einem Umlaufs um das Polygon und abschließenden Vergleich auf 2pi, ist möglich, aber wegen Rundungsfehlern und ungenauer Zahlendarstellung mit floating point, ist dieses Vorgehen nicht zu empfehlen.

7.11 Innen-Außen-Korrektheit

dass man einen beliebigen Strahl ausgehend vom zu untersuchenden Punkt Q wählen kann, um Innen und Außen zu unterscheiden, sieht man wie folgt mit einem *polaren* Scan-Line-Argument ein:



- Die Ecken werden bezüglich ihres Winkels (Polarkoordinaten!) aufsteigend sortiert und geben die Haltepunkte während des Scans an.
- Zwischen zwei Haltepunkten schneidet wohl jeder Strahl ausgehend vom Punkt Q die gleiche Anzahl von Kanten.
- An einem Haltepunkt bleibt die Anzahl gleich, wird um zwei erhöht oder um zwei erniedrigt.

Also ist die Summe der geschnittenen Kanten modulo 2 für jeden Strahl ausgehend von Q gleich!

Berücksichtigt man weiterhin die Orientierung der Kanten, so zeigt man leicht, dass die Windungszahl bei einem Umlauf konstant bleibt.

7.12 Konvexität-und-Einfachheit-Test

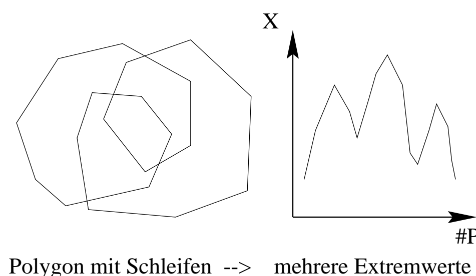
Für einfache Polygone ist es in linearer Zeit (in Anzahl der Ecken), d.h. in O(n), möglich festzustellen, ob sie konvex sind oder nicht.

Laufe in irgendeiner Richtung durch die geordnete Punkteliste und überprüfe, ob ein Vorzeichenwechsel bzgl. der Orientierung vorliegt; wenn ja, dann ist das Polygon nicht konvex.

Allerdings hat der zuvor nötige Einfachheit-Test eine Laufzeit von O(n log(n)).

Beides kann zusammen auch in linearer Zeit erfolgen:

1. Führe Vorzeichenwechseltest der Orientierung wie beschrieben durch (d.h. Überprüfen der *lokalen Konvexität*).
2. Überprüfe, ob mindestens einmal die Orientierung ungleich Null ist und keine Mehrfachecken auftreten.
3. Überprüfe, ob die x-Koordinaten (oder y-Koordinaten) der Ecken nur ein lokales Maximum und ein lokales Minimum annehmen (d.h. es gibt nur eine *Schleife*).



8 Zusammenfassung

Selbst vermeindlich einfache ¨Polygone¨ haben ihre Tücken.
Man muss auf Spezialfälle achten!
Wir wissen was:

- einfach – nicht einfach
- innen – außen
- überlappend – nicht überlappend
- und konvex – konkav

ist, und kennen Algorithmen bzw. haben die Kenntnisse solche zu entwickeln, um diese Eigenschaften voneinander zu unterscheiden, die alle Spezialfälle berücksichtigen.
Zwei wesentliche Prinzipien sind:

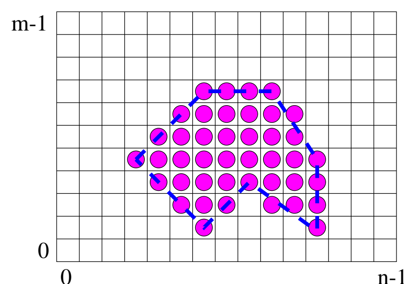
- Scan-Line-Prinzip (als Translation oder Rotation)
- Beobachtung der Veränderung von lokaler Information um so globale Information zu erhalten.

9 Füllen

9.1 Füllen von Polygonen

und anderen 2-dimensionalen Objekten
Man unterscheidet zwei Klassen

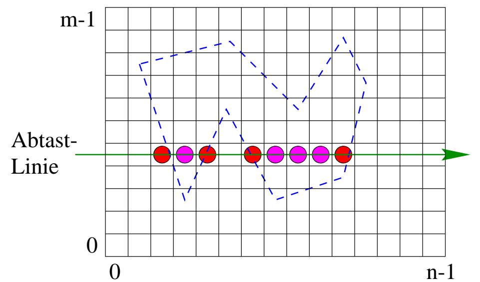
- Rand des Objektes liegt als geometrische Beschreibung vor
- ein Pixel des Inneren des Objektes ist bekannt
  - man füllt bezüglich einer bekannten Innenfarbe
  - man füllt bezüglich einer bekannten Randfarbe



9.2 Abtastlinien-Methode

Prinzip:

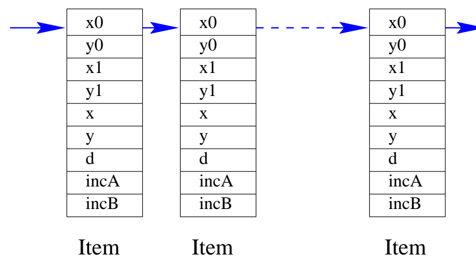
- arbeiten mit Abtastlinien (scanlines) von unten nach oben
- berechnen für jede Abtastlinie die Schnittpunkte mit dem Rand des Objekts
  (die Anzahl der Schnittpunkte ist immer gerade bei geschlossenen Kurven sofern Berührungspunkte korrekt behandelt werden)
- füllen jeden zweiten Abschnitt zwischen den Schnittpunkten beginnend mit dem ersten Abschnitt



Für ein einfaches Polygon verfährt man also wie folgt:

1. sortieren der Ecken lexikografisch nach y-Koordinate und x-Koordinate
2. beginnend mit der kleinsten y-Koordinate
3. aktualisieren der Liste der Kanten, die von der Abtastlinie geschnitten werden, es treten vier mögliche Fälle auf:
   - zwei Kanten werden eingefügt
   - zwei Kanten werden gelöscht
   - eine Kante wird eingefügt und eine Kante wird gelöscht
   - horizontale Kanten werden ignoriert
4. setzen aller Pixel in dieser Abtast-Linie für die schneidenden Kanten (z.B. mit einem leicht modifizierten Bresenham-Algorithmus)
5. füllen jedes zweiten Zwischenraumes
   (falls keine Kanten mehr geschnitten werden, ist man fertig)
6. fortfahren mit der nächsten Abtast-Linie bei 4. falls nächste Abtast-Linie keine Ecke enthält
7. fortfahren mit der nächsten Abtast-Linie bei 3. falls nächste Abtast-Linie Ecken enthält

Die Abtast-Linien-Datenstruktur hält also eine geordnete Liste von `Item`, die alle notwendigen Parameter und lokale Variablen einer Bresenham-Linien-Zeichen-Funktion enthält:

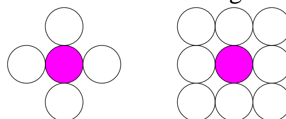


Für andere Objekte verfährt man analog!

9.3 Saatkorn-Methode

Gegeben sei ein inneres Pixel.

Man unterscheidet zwei Nachbarschaftsbeziehungen



Je nachdem werden entweder vier oder acht rekursive Aufrufe gestartet. Wurde der Rand des Objektes mit einem Bresenham-Algorithmus gezeichnet, sollte man die Vierer-Methode verwenden.

`GetPixel(x,y)` liefert Farbe für entsprechendes Pixel zurück.

`SeedFill()` füllt solange wie ¨unter¨ dem Pixel nicht die Füllfarbe und nicht die Randfarbe gefunden wird.

```
SeedFill_4(
  int x,
  int y,
  int fill_col,
  int boundary_col
) {
  int current_col=GetPixel(x,y);

  if( current_col!=fill_col &&
      current_col!=boundary_col ) {
    SetPixel(x,y,fill_col);
    SeedFill_4(x+1,y,fill_col,boundary_col);
    SeedFill_4(x,y+1,fill_col,boundary_col);
    SeedFill_4(x-1,y,fill_col,boundary_col);
    SeedFill_4(x,y-1,fill_col,boundary_col);
  }
}
```

`FloodFill()` füllt solange wie ¨unter¨ dem Pixel die Startfarbe gefunden wird.

```
FloodFill_4(
  int x,
  int y,
  int fill_col,
  int start_col
) {
  int current_col=GetPixel(x,y);

  if( current_col==start_col ) {
    SetPixel(x,y,fill_col);
    FloodFill_4(x+1,y,fill_col,start_col);
    FloodFill_4(x,y+1,fill_col,start_col);
    FloodFill_4(x-1,y,fill_col,start_col);
    FloodFill_4(x,y-1,fill_col,start_col);
  }
}
```

Natürlich sollte `FloodFill()` nur aufgerufen werden, wenn `start_col!=fill_col`, da die Rekursion sonst nicht terminiert.

Die Anzahl der rekursiven Aufrufe ist recht hoch, deshalb folgende Verbesserungen:

- selbstverwaltete Kellerstruktur anstelle der rekursiven Aufrufe
- bestimmen von Zeilenabschnitten, die gesetzt werden sollen (*Übung*)

10 Clipping

Unter ‘clipping’ versteht man das Ausschließen von Objekten bzw. Teilen von Objekten, die nicht in einem bestimmten Bereich (Fenster) liegen.

Wir betrachten hier nur zu den Koordinatenachsen parallel verlaufende rechteckige Fenster.

10.1 Clipping von Punkten

Clipping von Punkten ist einfach durch entsprechende Vergleiche der Koordinaten des Punktes mit den Fenstergrenzen möglich.

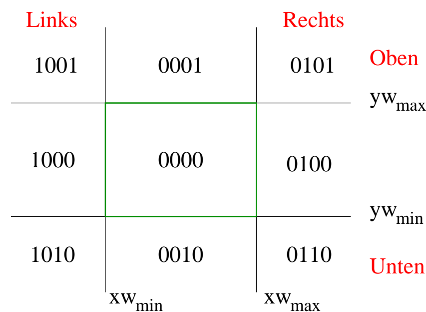
10.2 Clipping von Geradensegmenten

Betrachten zwei Clipping-Algorithmen:

- Algorithmus nach Cohen-Sutherland
- Algorithmus nach Liang-Barsky

10.3 Algorithmus nach Cohen-Sutherland

Idee:



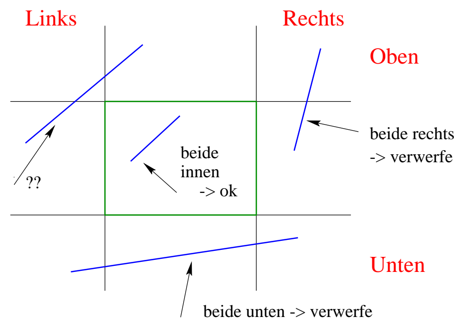
- Klassifizierung der Endpunkte bzgl. des Fensters
  - `O` oberhalb
  - `U` unterhalb
  - `L` links
  - `R` rechts

  mit jeweils einem Bit.
  Es entsteht ein Vier-Bit-Codewort

- Liegen beide Endpunkte auf der gleichen Seite des Fenster,
  –> so kann man das ganze Segment verwerfen.

- Ist kein Bit gesetzt,
  –> so liegt es vollständig im Innern des Fensters.

- Ansonsten schneidet man bzgl. eines Randes des Fensters ab und verfährt analog mit dem abgeschnittenen Segement.



Formulieren wir einige benötigte einfache Funktionen:

`int Encode(point p)` Liefert Codewort für einen Punkt zurück (*Übung!*).

`int Inside(int code_p)` Ein Punkt liegt innerhalb, wenn kein Bit im Codewort gesetzt ist, also
`return(!code_p)`.

`int Accept(int code_p, int code_q)` Ein Segment wird akzeptiert, wenn in keinem der beiden Codeworte ein Bit gesetzt ist, also
`return(code_p|code_q)`.

`int Reject(int code_p, int code_q)` Ein Segment wird verworfen, wenn in beiden Codeworten ein Bit an der gleichen Stelle gesetzt ist, also
`return(code_p&code_q)`.

`point Intersect(point p, point q, int border)` Liefert Schnittpunkt der Segmentes mit der entsprechenden Geraden des Fensters (*Übung!*).

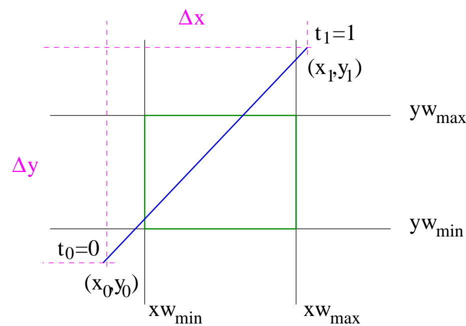
`void Swap(..., ...)` Vertauscht die beiden Argumente (*Übung!*).

Algorithmus:
(als C-Programmgerüst, welches leicht als *Übung* vervollständigt werden kann.)

```
ClipCohenSutherland(
  point p, point q
) {
  done = 0;
  while (done != ACCEPT &&
         done != REJECT     ) {
    code_p = Encode(p);
    code_q = Encode(q);
    if ( Accept(code_p,code_q) ) done = ACCEPT;
    else {
      if ( Reject(code_p,code_q) ) done = REJECT;
      else {
        if ( Inside(code_p) ) {
          Swap(&p,&q);
          Swap(&code_p,&code_q);
        }
        if (code_p&LEFT) {
          p = Intersect(p,q,left_border);
        else if (code_p&RIGHT) {
          p = Intersect(p,q,right_border);
        }
        else ...
```

```
        }
      }
    }
    if (done == ACCEPT) DrawLine(p,q,color);
}
```

10.4 Algorithmus nach Liang-Barsky



Betrachten parametrisierte Form des Segmentes

$$\begin{aligned}\Delta x &= x_1 - x_0\\ \Delta y &= y_1 - y_0\end{aligned}$$

$$\begin{aligned}x &= x_0 + t\Delta x\\ y &= y_0 + t\Delta y \qquad 0 \le t \le 1\end{aligned}$$

Dieses muss innerhalb des Fensters liegen. Wir erhalten also die Bedingungen:

$$xw_{min} \le x_0 + t\Delta x \le xw_{max}$$

$$yw_{min} \le y_0 + t\Delta y \le yw_{max}$$

Dies sind vier Ungleichungen der Form:

$$t \cdot p_k \le q_k \qquad k = 1, 2, 3, 4$$

N”amlich

$$\begin{aligned}t \cdot (-\Delta x) &\le x_0 - xw_{min}\\ t \cdot \Delta x &\le xw_{max} - x_0\\ t \cdot (-\Delta y) &\le y_0 - yw_{min}\\ t \cdot \Delta y &\le yw_{max} - y_0\end{aligned}$$

Idee des Verfahrens:

- Betrachten orientierte Gerade durch das Segment mit Ursprung in einem der Endpunkte.
- Da die Segmentgerade orientiert ist, kann man entscheiden, ob man von Innen nach Außen oder umgekehrt von Außen nach Innen eine Fenstergerade schneidet.
- Berechnen die Parameterwerte der vier Schnittpunkte. (Sofern sie nicht parallel zu einer diesen Gerade ist; siehe letzten Punkt).

- Berechnen das Minimum der Parameterwerte für ¨von Innen¨ und 0.
- Berechnen das Maximum der Parameterwerte für ¨von Außen¨ und 1.
- Ist das Minimum kleiner als das Maximum, so liegt das dazwischenliegende Geradensegment im Fenster.
- Liegt die Gerade parallel zu einer Fenstergerade, so kann man anhand von q entscheiden, ob das Segment außerhalb liegt.

Sofern die Differenz nicht Null ist, können wir t jeweils durch eine Division berechnen:

$$t = q_k/p_k$$

Ist ein $p_k = 0$, so verl"auft das Segment parallel zu einer der Fensterseiten, ist dann $q_k < 0$, so liegt es au"serhalb.

Ist ein $p_k < 0$, so l"auft man bzgl. der parametrisierten Gleichung von Au"sen nach Innen, also verschieben wir t_0.

Ist ein $p_k > 0$, so l"auft man bzgl. der parametrisierten Gleichung von Innen nach Au"sen, also verschieben wir t_1.

Es muss stets gelten: $t_0 < t_1$.

Algorithmus:

(als C-Programmgerüst, welches leicht als *Übung* vervollständigt werden kann.)

`t0` und `t1` sind globale Variablen; Die Funktion `ClipTest()` testet jeweils für eine der Ungleichungen:

```
double t0,t1;

int ClipTest(
  double p, double q
) {
  double t;

  if ( p < 0 ) {
    t  = q/p;
    if ( t > t1 ) return 0;
    t0 = Max(t0,t);
  }
  else if ( p > 0 ) {
    t  = q/p;
    if ( t < t0 ) return 0;
    t1 = Min(t1,t);
  }
  else if ( q < 0 ) return 0;
  return 1;
}
```

Hauptfunktion:

```
ClipLiangBarsky(
  point p, point q
) {
  double Dx;

  t0 = 0;
  t1 = 1;
```

```
  Dx = q.x-p.x;
  if ( ClipTest(-Dx, p.x-xwmin) ) {
    if ( ClipTest(Dx, xwmax-p.x) ) {
      Dy = q.y-p.y;
      if ( ClipTest(-Dy, p.y-ywmin) ) {
        if ( ClipTest(Dy, ywmax-p.y) ) {
          if ( t0 > 0 ) {
            p.x = p.x + t0*Dx;
            p.y = p.y + t0*Dy;
          }
          if ( t1 < 1) {
            q.x = p.x + t1*Dx;
            q.y = p.y + t1*Dy;
          }
          DrawLine(p,q,color);
        }
      }
    }
  }
}
```

10.5 Clipping von Polygonen

10.5.1 Algorithmus nach Sutherland-Hodgeman

Idee:
Verarbeite die Ecken in einer Art Pipeline.

Betrachten wir eine Fenstergerade und clippen das ganze Polygon an diesem Rand: Man merkt sich zweckmäßerweise den ersten und letzten Punkt, der durch einen Schnitt von einer Kante mit einem Rand entsteht, um das Polygon dann am Ende schließen zu können.

Statt nun das Polygon stets komplett an einem Rand zu clippen, schiebt man den Punkt gleich in der Pipeline weiter. Dies reduziert den Speicherplatz.

Wir brauchen wieder eine Reihe nützlicher Funktionen:

`int Inside(point p, border b)` Liefert 1, falls der Punkt `p` bezgl. dem Rand `b` im ¨richtigen¨ liegt, sonst 0.

`int Cross(point p, point q, border b)` Liefert 1, falls das Segment von `p` nach `q` den Rand `b` schneidet, sonst 0.

`point Intersect(point p, point q, border b)` Liefert den Schnittpunkt des Segmentes von `p` nach `q` mit dem Rand `b`.

`Out(point p)` Gibt den Punkt aus, bzw. gibt in an nochfolgende Einheiten weiter.

Wir halten die folgenden globalen Variablen, dabei halten die Komponenten der Felder jeweils einen Rand (`L=0`, `R=1`, `U=2` und `O=3`):

```
point last[4];       // letzter Punkt
point first[4];      // erster Punkt
int   newborder[4]; // Rand schon betrachtet?
```

Das Schieben der Punkte durch die Pipeline erledigt die Funktion `ClipPoint()`:

```
ClipPoint(
  point  p,
```

```
  border b
) {
  point c;

  if ( newborder[b] ) {
    // war erste Ecke fur diesen Rand
    first[p]      = p;
    newborder[b] = false;
  }
  else {
    // es ist eine weitere Ecke
    if ( Cross(P, last[b], b) ) {
      // d.h. Segment zwischen neuem Punkt und
      // letzten Punkt schneidet diesen Rand
      c = Intersect(p, last[b], b);
      if ( b < 4 )
        // es gibt noch eine Pipelinestufe
        ClipPoint(c, b+1);
      else
        // gib den Punkt aus
        Out(c);
    }
  }
  last[p] = p;   // merke als letzten Punkt
  if ( Inside(p, b) ) {
    if ( b < 4 )
      // es gibt noch eine Pipelinestufe
      ClipPoint(p, b+1);
    else
      // gib den Punkt aus
      Out(c);
  }
}
```

Am Ende muss das Polygon noch geschlossen werden, dies leistet die folgende Funktion:

```
CloseClip()
{
  border b;
  point  c;

  for ( b=0; b<4; b++ ) {
    if ( Cross(last[b], first[b], b) )
      c = Intersect(last[b], first[b], b);
      if ( b < 4 ) ClipPoint(c, b+1);
      else         Out(c);
    }
  }
}
```

Hauptfunktion:

```
ClipSutherlandHodgeman(
  int     N,
  polygon P
```

```
) {
  border b;
  int    i;

  for ( b=0; b < 4; b++) newborder[b]=true;
  for ( i=0; i < N; i++) ClipPoint(P[i],0);
  CloseClip();
}
```

10.5.2 Algorithmus nach Weiler-Atherton

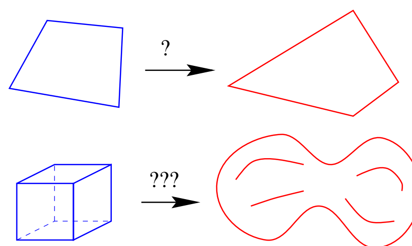
11 Geometrische Transformationen

11.1 Analytische Geometrie

3-dimensionale Punkte

Vektoren
Abstand
Ortsvektor
Vektorraum, 3-dimensionaler Punktraum
Vektoroperationen
Vektoraddition
Skalarmultiplikation
Skalarprodukt
Vektorprodukt
Hadamardsche Produkt
Nützliche Gleichungen
Lineare Abhängigkeit
Basis

11.2 Transformationen



11.3 Affine Kombination

oder auch barizentrische Kombination

Eine affine Kombination ist eine Gewichte ¨Summe¨ von Punkten; dies ist jedoch keine eigentliche Summe von Punkten (was nicht definiert wäre) sondern eine Summe von Differenzen von Punkten, also letztlich eine gewichtete Summe von Vektoren zu einem Punkt.

$$
\begin{aligned}
\bar{b} &= \bar{b}_0 + \sum_{j=1}^{n-1} \alpha_j \cdot \underbrace{(\bar{b}_j - \bar{b}_0)}_{\vec{b}_j} \\
&= \sum_{j=0}^{n-1} \alpha_j \cdot \bar{b}_j
\end{aligned}
$$

wobei f"ur die Gewichte α_j gilt:

$$\sum_{j=0}^{n-1} \alpha_j = 1$$

Wählt man alle Gewichte gleich *1/n*, so erhält man ein Art **Schwerpunkt**:

$$\bar{b} = \frac{1}{n} \sum_{j=0}^{n-1} \bar{b}_j$$

Sind alle Gewichte größer Null, dann spricht man von konvexer Kombination, der konstruierte Punkt liegt dann in der konvexen Hülle der Punkte (Beweis: *Übung*).

11.4 Affine Abbildungen

affine Vektorabbildung

$$\Phi : \mathbb{R}^3 \longrightarrow \mathbb{R}^3; \vec{v} \longrightarrow \vec{v}'$$

und zwar so dass

$$\forall\, \vec{v}_1, \vec{v}_2 \in \mathbb{R}^3, \alpha, \beta \in \mathbb{R}$$

gilt

$$\Phi(\alpha\vec{v}_1 + \beta\vec{v}_2) = \alpha\Phi(\vec{v}_1) + \beta\Phi(\vec{v}_2)$$

Multipliziert man einen Vektor von links mit einer Matrix, so stellt dies eine affine Vektorabbilung dar:

$$\Phi(\vec{v}) = A \cdot \vec{v} \quad \text{mit} \quad A \; 3 \times 3 \text{ Matrix}$$

Dies sieht man leicht ein:

$$\begin{aligned}
\Phi(\alpha\vec{v}_1 + \beta\vec{v}_2) &= A \cdot (\alpha\vec{v}_1 + \beta\vec{v}_2) \\
&= A \cdot \alpha\vec{v}_1 + A \cdot \beta\vec{v}_2 \\
&= \alpha\Phi(\vec{v}_1) + \beta\Phi(\vec{v}_2)
\end{aligned}$$

Ist die Matrix *A* singulär, dann bezeichnet man die affine Abbildung als entartet (hierzu zählen die Projektionen).

Im Folgenden betrachten wir - sofern nicht speziell vermerkt - nur nicht-entartete affine Abbildungen.

zu affinen Vektorabbildungen definiert man

affine Punktabbildungen

$$\Phi(\vec{v}) = A \cdot \vec{v} + \vec{c}$$

Der Verschiebungsvektor ist bereits eindeutig festgelegt, wenn man sich die Matrix *A* und einen Punkt mit seinem Bildpunkt vorgibt.

Insbesondere kann man eine affine Punktabbildung konstruieren, wenn man sich lediglich vier nicht in einer Ebene liegende Punkte und deren gewünschten Bildpunkte vorgibt:

$$\begin{aligned}
A &= (\vec{x}_1' - \vec{x}_0' \; \vec{x}_2' - \vec{x}_0' \; \vec{x}_3' - \vec{x}_0') \cdot \\
&\quad (\vec{x}_1 - \vec{x}_0 \; \vec{x}_2 - \vec{x}_0 \; \vec{x}_3 - \vec{x}_0)^{-1}
\end{aligned}$$

$$c = \vec{x}_0' - A\vec{x}_0$$

Eigenschaften der affinen Abbildungen:

- *det(A)=0* entartete affine Abbildung
- |*det(A)=1*| Volumen eines Spates bleibt erhalten
- Parallelen bleiben Parallelen
- Ebenen bleiben Ebenen
- Strahlensätze bleiben erhalten
- nennt man auch: 'rigid body transformation'

11.5 Homogene Koordinaten

Die affine Punktabbildung ist eine gemischte Operation aus Matrix-Vektor-Multiplikation und Vektor-Addition.

Diese gemischte Operation kann man durch Einführen von homogenen Koordinaten in eine einfache jetzt allerdings vierdimensionale Matrizen-Vektor-Multiplikation verwandeln.

11.6 Translation

in kartesischen Koordinaten

$$\vec{P'} = \vec{P} + \vec{T}$$

in homogenen Koordinaten

$$\begin{aligned} \vec{P'} &= T \cdot \vec{P} \\ &= \begin{pmatrix} 1 & 0 & 0 & t_0 \\ 0 & 1 & 0 & t_1 \\ 0 & 0 & 1 & t_2 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} p_0 \\ p_1 \\ p_2 \\ 1 \end{pmatrix} \end{aligned}$$

Für die Umkehrung erhalten wir:
in kartesischen Koordinaten

$$\vec{P} = \vec{P'} - \vec{T}$$

in homogenen Koordinaten

$$\begin{aligned} \vec{P} &= T^{-1} \cdot \vec{P'} \\ &= \begin{pmatrix} 1 & 0 & 0 & -t_0 \\ 0 & 1 & 0 & -t_1 \\ 0 & 0 & 1 & -t_2 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} p'_0 \\ p'_1 \\ p'_2 \\ 1 \end{pmatrix} \end{aligned}$$

11.7 Skalierung

in kartesischen Koordinaten mit $s_i > 0$ (unter Anwendung der hadamardschen Multiplikation)

$$\vec{P'} = \vec{P} \star \vec{S} = \vec{P} \star \begin{pmatrix} s_0 \\ s_1 \\ s_2 \end{pmatrix}$$

in homogenen Koordinaten

$$\begin{aligned}\vec{P'} &= S \cdot \vec{P} \\ &= \begin{pmatrix} s_0 & 0 & 0 & 0 \\ 0 & s_1 & 0 & 0 \\ 0 & 0 & s_2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} p_0 \\ p_1 \\ p_2 \\ 1 \end{pmatrix}\end{aligned}$$

Man beachte: Die Punkte werden ¨scheinbar¨ zum Ursprung verschoben, da der gesamte Raum skaliert wird.

Für die Umkehrung erhalten wir:
in kartesischen Koordinateni mit $s_i > 0$

$$\vec{P} = \vec{P'} \star \begin{pmatrix} 1/s_0 \\ 1/s_1 \\ 1/s_2 \end{pmatrix}$$

in homogenen Koordinaten

$$\begin{aligned}\vec{P} &= S^{-1} \cdot \vec{P'} \\ &= \begin{pmatrix} 1/s_0 & 0 & 0 & 0 \\ 0 & 1/s_1 & 0 & 0 \\ 0 & 0 & 1/s_2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} p'_0 \\ p'_1 \\ p'_2 \\ 1 \end{pmatrix}\end{aligned}$$

11.8 Scherung

Scherungen können nicht mehr durch einfache Vektoroperationen angegeben werden; wir beschränken uns hier auf die Darstellung mit homogenen Koordinaten.

Betrachten wir eine Scherung bei gleichbleibender z-Koordinate:

$$\begin{aligned}\vec{P'} &= V_z \cdot \vec{P} \\ &= \begin{pmatrix} 1 & 0 & z_0 & 0 \\ 0 & 1 & z_1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} p_0 \\ p_1 \\ p_2 \\ 1 \end{pmatrix} \\ &= \begin{pmatrix} p_0 + z_0 p_2 \\ p_1 + z_1 p_2 \\ p_2 \\ 1 \end{pmatrix}\end{aligned}$$

Analog erhält man Scherungen in den anderen Koordinatenrichtungen. Scherungen sind nicht kommutativ (*Übung!*).

Ein allgemeine Schermatrix hat die Form:

$$\begin{aligned}\vec{P'} &= V \cdot \vec{P} \\ &= \begin{pmatrix} 1 & v_0 & v_1 & 0 \\ v_2 & 1 & v_3 & 0 \\ v_4 & v_5 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} p_0 \\ p_1 \\ p_2 \\ 1 \end{pmatrix}\end{aligned}$$

Für die Umkehrung ist man auf allgemeine Matrizeninversion angewiesen.

11.9 Spiegelung

Spiegelungen sind Spezialfäll der Skalierungen, wenn man auch negative Skalierungswerte zulässt und deren Beträge 1 sind.

Beispiel einer Spiegelung an der *xy*-Ebene:

in kartesischen Koordinaten

$$\vec{P'} = \vec{P} \star \vec{S} = \vec{P} \star \begin{pmatrix} 1 \\ 1 \\ -1 \end{pmatrix}$$

in homogenen Koordinaten

$$\begin{aligned} \vec{P'} &= S \cdot \vec{P} \\ &= \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} p_0 \\ p_1 \\ p_2 \\ 1 \end{pmatrix} \end{aligned}$$

11.10 Rotation

11.11 Trigonometrische Additionstheoreme

Wie bei vielen trigonometrschen Beweisen liegt der Trick im Einzeichnen der ¨richtigen¨ Hilfslinien und Anwenden einfacher Beziehungen.

$$\begin{aligned} \frac{x}{\sin\phi} &= \frac{\cos\delta}{1} \\ \frac{y}{\sin\delta} &= \frac{\cos\phi}{1} \\ \sin(\delta+\phi) &= x+y \\ &= \cos\delta\,\sin\phi + \sin\delta\,\cos\phi \end{aligned}$$

11.12 Rotation mit Quaternionen

Was sind Quaternionen?

Quaternionen sind Elemente einer algebraischen Struktur, die als eine Erweiterung der komplexen Zahlen angesehen werden kann.

Man kann Addition und Multiplikation als innere Operationen der Struktur definieren, so dass die Körperaxiome mit Ausnahme der allgemeinen Kommutativität der Multiplikation gelten.

Man nennt einen solchen Körper auch Schiefkörper.

Quaternionen sind vierdimensionale Objekte, man schreibt sie jedoch zweckmäßigerweise als Skalar-Komponente und dreidimensionale Vektor-Komponente:

$$q = [a, \vec{v}] \in \mathbb{R}^4$$

Quaternionenaddition:

$$[a, \vec{v}] + [b, \vec{w}] = [a + b, \vec{v} + \vec{w}]$$

Quaternionenmultiplikation:

$$\begin{aligned} [a, \vec{v}] \star [b, \vec{w}] = {} & [ab - \langle \vec{v}, \vec{w} \rangle, \\ & a\vec{w} + b\vec{v} + \vec{v} \times \vec{w}] \end{aligned}$$

Man überprüfe als *Übung*, dass die Körperaxiome erfüllt sind (z.B. Assoziativgesetze und Distributivgesetze).

Quaternionen mit Skalar-Komponente gleich Null nennt man auch reine Quaternionen ('pure quaternions').

Zusammenfassung der ¨Ähnlichkeiten¨ zwischen komplexen Zahlen und Quaternionen:

Darstellung	$z = a + ib$ $q = [a, \vec{v}]$
konjugiert komplex	$\bar{z} = a - ib$ $\bar{q} = [a, -\vec{v}]$
gemischtes Produkt	$z\bar{z} = (a + ib) \cdot (a - ib)$ $= a^2 + b^2$ $= \|z\bar{z}\|$ $q\bar{q} = [a, \vec{v}] \star [a, -\vec{v}]$ $= [a^2 - \langle \vec{v}, -\vec{v} \rangle, -a\vec{v} + a\vec{v} + \vec{v} \times \vec{v}]$ $= [a^2 - \langle \vec{v}, -\vec{v} \rangle, \vec{0}]$
Betrag	$\|z\| = \sqrt{a^2 + b^2}$ $= \sqrt{z\bar{z}}$ $= \sqrt{\|z\bar{z}\|}$ $\|q\| = \sqrt{a^2 + \langle \vec{v}, \vec{v} \rangle^2}$ $= \sqrt{\|q\bar{q}\|}$
Kehrwert	$1/z = \bar{z}/(z\bar{z})$ $= (a - ib)/(a^2 + b^2)$ $= \bar{z}/\|z\bar{z}\|$ $1/q = \bar{q}/\|q\bar{q}\|$ $= [a, -\vec{v}]/(a^2 + \langle \vec{v}, \vec{v} \rangle)$

Für Einheitsquaternionen, d.h. solche mit Betrag gleich Eins, gilt weiterhin:

$$|q| = 1 \quad \Longrightarrow \quad q^{-1} = \bar{q}$$

11.13 Drehformel nach Hamilton

Mithilfe der Quaternionen lassen sich Drehungen im Dreidimensionalen sehr einfach spezifizieren und berechnen:

Sei $\vec{u}$ mit $|\vec{u}| = 1$ der normierte Vektor der Rotationsachse und sei ϕ der gew"unschte Rotationswinkel.

Konstruieren Quaternion q wie folgt:

$$q = [\cos(\phi/2), \sin(\phi/2)\,\vec{u}]$$

Interpretieren einen zu rotierenden Punkt mit Ortsvektor $\vec{x}$ als reinen Quaternion

$$x = [0, \vec{x}]$$

Dann ergibt sich der Zielpunkt $\vec{x}'$ wieder in Form eines reinen Quaternionen als

$$x' = q \star x \star \bar{q}$$

11.14 Nachweis der Korrektheit der Drehformel

Berechnen wir zuerst einige Beziehungen zwischen Winkel und Halbwinkel, die wir später brauchen werden:

$$\begin{aligned}\sin\phi &= \sin(\phi/2+\phi/2)\\ &= \sin(\phi/2)\cos(\phi/2)+\cos(\phi/2)\sin(\phi/2)\\ &= 2\sin(\phi/2)\cos(\phi/2)\end{aligned}$$

$$\begin{aligned}\cos\phi &= \cos(\phi/2+\phi/2)\\ &= \cos(\phi/2)\cos(\phi/2)-\sin(\phi/2)\sin(\phi/2)\\ &= \cos^2(\phi/2)-\sin^2(\phi/2)\end{aligned}$$

$$\begin{aligned}1-\cos\phi &= 1-\cos^2(\phi/2)+\sin^2(\phi/2)\\ &= 1-(1-\sin^2(\phi/2))+\sin^2(\phi/2)\\ &= 2\sin^2(\phi/2)\end{aligned}$$

Führen wir folgende Bezeichnungen und Abkürzungen ein:
zu rotierender Punkt

$$\vec{x} = \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix}$$

Rotationsachse

$$\vec{u} = \begin{pmatrix} u_0 \\ u_1 \\ u_2 \end{pmatrix} \quad \text{mit} \quad |\vec{u}| = 1$$

Rotationsquaternion mit Rotationswinkel ϕ

$$q = [\cos(\phi/2), \sin(\phi/2)\vec{u}]$$

Abk"urzungen

$$\begin{aligned}t &= \sin(\phi/2)\\ s &= \cos(\phi/2)\end{aligned}$$

Damit schreiben wir also die Rotation nach der Drehformel wie folgt:

$$x' = [s, t\cdot \begin{pmatrix} u_0 \\ u_1 \\ u_2 \end{pmatrix}] \star [0,\vec{x}] \star [s, -t\cdot \begin{pmatrix} u_0 \\ u_1 \\ u_2 \end{pmatrix}]$$

Nun berechnen wir eine Matrix *M*, die die folgende ¨Gleichung¨ (bei entsprechender Interpretation der reinen Quaternionen und dreidimensionalen Vektoren) erfüllt:

$$[s, t\vec{u}] \star [0,\vec{x}] \star [s, -t\vec{u}] = q \star x \star \bar{q} \equiv M \cdot \vec{x} = M \cdot \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix}$$

und zeigen dann, dass die Matrix *M* eine Rotationsmatrix ist, die man sowohl als Produktmatrix aus Achsen-Rotationsmatrizen als auch durch direkte Matrizenkonstruktion erhalten kann.

Bemerkung: Wir führen hier keine zusätzliche Translation aus. Diese ist nicht durch Multiplikation mit Quaternionen darstellbar. Man muss also beide Operationen - Rotation und Translation - wieder getrennt betrachten.

Rechnen wir die hamiltonsche Formel aus:

$$
\begin{aligned}
&[s, t\vec{u}] \star [0, \vec{x}] \star [s, -t\vec{u}] \\
&= [-t\langle\vec{u}, \vec{x}\rangle, s\vec{x} + t\vec{u} \times \vec{x}] \star [s, -t\vec{u}] \\
&= [-st\langle\vec{u}, \vec{x}\rangle - \langle s\vec{x} + t\vec{u} \times \vec{x}, -t\vec{u}\rangle, \\
&\quad t^2\langle\vec{u}, \vec{x}\rangle\vec{u} + s \cdot (s\vec{x} + t\vec{u} \times \vec{x}) + (s\vec{x} + t\vec{u} \times \vec{x}) \times (-t\vec{u})] \\
&= [-st\langle\vec{u}, \vec{x}\rangle + st\langle\vec{u}, \vec{x}\rangle - t^2\langle\vec{u} \times \vec{x}, \vec{u}\rangle, \\
&\quad t^2\langle\vec{u}, \vec{x}\rangle\vec{u} + s^2\vec{x} + st\vec{u} \times \vec{x} - st\vec{x} \times \vec{u} - t^2(\vec{u} \times \vec{x}) \times \vec{u})] \\
&= [0, t^2\langle\vec{u}, \vec{x}\rangle\vec{u} + s^2\vec{x} + 2st\vec{u} \times \vec{x} - t^2(\vec{u} \times \vec{x}) \times \vec{u})]
\end{aligned}
$$

Wir erkennen als erstes, dass tatsächlich ein reiner Quaternion als Ergebnis entsteht.

Analysieren wir die Vektorkomponente.

Hierzu notieren wir zuerst die Vektorprodukte mit Koordinatenwerten:

$$
\begin{aligned}
\begin{pmatrix} u_0 \\ u_1 \\ u_2 \end{pmatrix} \times \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} \times \begin{pmatrix} u_0 \\ u_1 \\ u_2 \end{pmatrix} &= \begin{pmatrix} u_1x_2 - u_2x_1 \\ u_2x_0 - u_0x_2 \\ u_0x_1 - u_1x_0 \end{pmatrix} \times \begin{pmatrix} u_0 \\ u_1 \\ u_2 \end{pmatrix} \\
&= \begin{pmatrix} (u_2x_0 - u_0x_2)u_2 - (u_0x_1 - u_1x_0)u_1 \\ (u_0x_1 - u_1x_0)u_0 - (u_1x_2 - u_2x_1)u_2 \\ (u_1x_2 - u_2x_1)u_1 - (u_2x_0 - u_0x_2)u_0 \end{pmatrix}
\end{aligned}
$$

Formulieren wir damit die *x*-Komponente des Resultatsvektors aus:

$$
\begin{aligned}
x_0' &= t^2(u_0x_0 + u_1x_1 + u_2x_2)u_0 \\
&\quad + s^2x_0 + 2st(u_1x_2 - u_2x_1) \\
&\quad - t^2(u_2x_0 - u_0x_2)u_2 - (u_0x_1 - u_1x_0)u_1 \\
&= x_0 \cdot (t^2u_0^2 + s^2 - t^2u_2^2 - t^2u_1^2) \\
&\quad + x_1 \cdot (t^2u_0u_1 - 2stu_2 + t^2u_0u_2) \\
&\quad + x_2 \cdot (t^2u_ou_2 + 2stu_1 + t^2u_0u_2) \\
&= x_0 \cdot (s^2 + 2t^2u_0^2 - t^2 \underbrace{(u_0^2 + u_1^2 + u_2^2)}_{=1}) \\
&\quad + x_1 \cdot (t^2u_0u_1 - 2stu_2 + t^2u_0u_2) \\
&\quad + x_2 \cdot (t^2u_ou_2 + 2stu_1 + t^2u_0u_2)
\end{aligned}
$$

Nun kann man unter Zuhilfenahme von

$$
\begin{aligned}
\vec{u}\vec{u}^\top &= \begin{pmatrix} u_0 \\ u_1 \\ u_2 \end{pmatrix} \cdot (u_0\ u_1\ u_2) \\
&= \begin{bmatrix} u_0^2 & u_0u_1 & u_0u_2 \\ u_0u_1 & u_1^2 & u_1u_2 \\ u_0u_2 & u_1u_2 & u_2^2 \end{bmatrix}
\end{aligned}
$$

und analogem Formulieren der anderen Komponenten (*Übung*) das Ergebnis in folgender Art notieren:

$$\vec{x}' = (s^2 - t^2) \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} + 2t^2 \cdot \vec{u}\vec{u}^\top - 2st \cdot \begin{bmatrix} 0 & u_2 & -u_1 \\ -u_2 & 0 & u_0 \\ u_1 & -u_0 & 0 \end{bmatrix}$$

Und wir erhalten durch Ersetzen der Abkürzungen und Anwenden der Halbwinkelbeziehungen von oben folgende Darstellung:

$$\vec{x}' = \cos\phi \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} + (1 - \cos\phi) \cdot \vec{u}\vec{u}^\top - \sin\phi \cdot \begin{bmatrix} 0 & u_2 & -u_1 \\ -u_2 & 0 & u_0 \\ u_1 & -u_0 & 0 \end{bmatrix}$$

Was hat dies mit der Rotationsmatrix zu tun?

Betrachten wir nochmals die *z*-Achsen-Rotationsmatrix und schreiben diese auch als eine ¨geschickte¨ Summe von Matrizen, wobei wir erst wieder ein paar Kurznotationen einführen:

$$\begin{aligned} E &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \\ E_{22} &= \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \\ E_{01} &= \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \\ E_{10} &= \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \end{aligned}$$

Damit erhalten wir:

$$\begin{aligned} R_z(\phi) &= \begin{bmatrix} \cos\phi & -\sin\phi & 0 \\ \sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{bmatrix} \\ &= \cos\phi \cdot E + (1 - \cos\phi) \cdot E_{22} + \sin\phi \cdot (E_{01} - E_{10}) \end{aligned}$$

Multiplizieren wir nun diese *z*-Rotationsmatrix von links und rechts mit einer *orthonormalen* Matrix *A* (was wir ja genau auch mit den *x*- und *y*-Rotationsmatrizen getan haben):

$$\begin{aligned} R &= AR_z(\phi)A^\top \\ &= A(\cos\phi E + (1 - \cos\phi)E_{22} + \sin\phi(E_{01} - E_{10}))A^\top \\ &= \cos\phi AEA^\top + (1 - \cos\phi)AE_{22}A^\top + \sin\phi A(E_{01} - E_{10})A^\top \\ &= \cos\phi E + (1 - \cos\phi)\vec{u}\vec{u}^\top + \sin\phi(\vec{w}\vec{v}^\top - \vec{v}\vec{w}^\top) \end{aligned}$$

Letzte Gleichung gilt, wenn wir A als

$$A = \begin{bmatrix} w_0 & v_0 & u_0 \\ w_1 & v_1 & u_1 \\ w_2 & v_2 & u_2 \end{bmatrix}$$

darstellen und nutzen wir weiterhin, dass f"ur die orthonormale Matrix gilt:

$$\vec{w} \times \vec{v} = \vec{u}$$

so erhalten wir letztlich

$$R = \cos\phi E + (1 - \cos\phi)\vec{u}\vec{u}^\top + \sin\phi \begin{bmatrix} 0 & u_2 & -u_1 \\ -u_2 & 0 & u_0 \\ u_1 & -u_0 & 0 \end{bmatrix}$$

12 Objektmodellierung

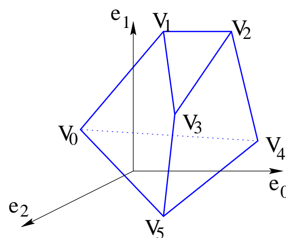
12.1 Einfache geometrische Objekte

z.B. polygonale Objekte, Kugeln, Ellipsoide, Zylinder, Kegel, Tori.

12.2 Polygonale Objekte

Als polygonale Objekte oder Polyeder bezeichnen wir Objekte, deren Oberfläche durch Polygone beschrieben wird (*polygon meshes*).

Beispiel:



Repräsentation eines solchen Objektes:

- *Punktliste* (oder Eckenliste)

$$V_0, V_1, V_2, V_3, V_4, V_5$$

- und *Kantenliste*

$$E_0 = (V_0, V_1); E_1 = (V_1, V_3); E_2 = (V_3, V_5); \ldots$$

- und *Polygonliste*
  - als Punktlisten

$$S_0 = (V_0, V_1, V_3, V_5); S_1 = (V_1, V_2, V_3); \ldots$$

  - oder Kantenlisten

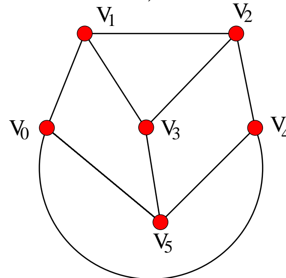
$$S_0 = (E_0, E_1, E_2, E_4); \ldots$$

Ob die Polygonliste gebraucht wird, hängt von der Anwendung ab (für Drahtgittermodelle z.B. wird sie nicht benötigt). Außer der Punktliste enthalten die Datenstrukturen nur Zeiger.

12.3 Oberfläche als Graph

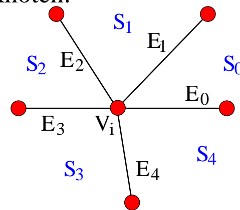
In manchen Anwendungen ist es häufig sinnvoll, einen gesamten Graphen (als Datenstruktur) der Oberfläche abzuspeichern.

Oberfläche ist Graph, Ecken sind Knoten, Kanten sind Kanten.

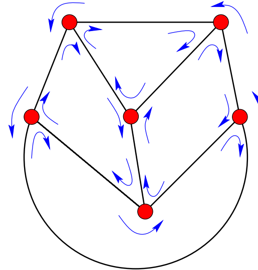


Objekt nicht ¨entartet¨ –> Graph ist planar.

Ordne die Kanten an einem Knoten:



Damit lassen sich durch Angabe jeweils einer Kante leicht die Polygone der Fassetten des Polyeders angeben:



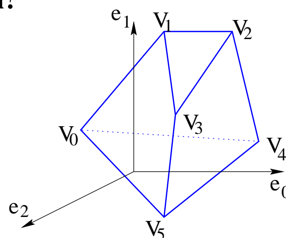
Das heißt für ein Polygon einer Fassette genügt ein Zeiger auf eine Kante im Graphen.

Hierbei sollten weitere Anforderungen an einen planaren Graphen gestellt werden, um keine ¨entarteten¨ Polyeder zu erhalten. Hierauf soll jedoch nicht weiter eingegangen werden.

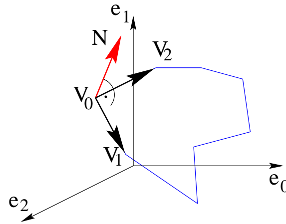
12.4 Innen und Außen

Oberfläche ist Menge von Polygonen (deren Ecken und Kanten durch Suche in einem planaren Graphen gefunden werden kann).

Was ist Innen bzw. Außen?



Normalerweise gehen wir davon aus, dass wir als Betrachter *außen* stehen. Ebenengleichung einer Fassette:



Nehmen 2 nicht linear abh"angige Kantenvektoren und bilden einen Normalenvektor der Ebene der Fassette:

$$\begin{aligned}\vec{E}_0 &= \vec{V}_1 - \vec{V}_0 \\ \vec{E}_1 &= \vec{V}_2 - \vec{V}_0\end{aligned}$$

F"ur jeden Punkt $\vec{P}$ der Ebene gilt dann:

$$\langle \vec{P} - \vec{V}_0, \vec{N} \rangle = 0$$

Dies k"onnen wir anderes schreiben und $\vec{P} = (xyz)^\top$ sowie $\vec{N} = (ABC)^\top$ einsetzen

$$\begin{aligned}\langle \vec{P}, \vec{N} \rangle \underbrace{- \langle \vec{V}_0, \vec{N} \rangle}_{=D} &= 0 \\ Ax + By + Cz + D &= 0\end{aligned}$$

und erhalten die "ubliche Schreibweise einer *impliziten Ebenengleichung*.

Mit dieser impliziten Ebenengleichung definieren wir oberhalb und unterhalb bezüglich dieser Fassette analog wie wir es bereits beim Polygon gesehen haben:

$$\langle \vec{P}, \vec{N} \rangle + D = \begin{cases} < 0 & \text{innerhalb} \\ = 0 & \text{auf der Oberfl"ache} \\ > 0 & \text{innerhalb} \end{cases}$$

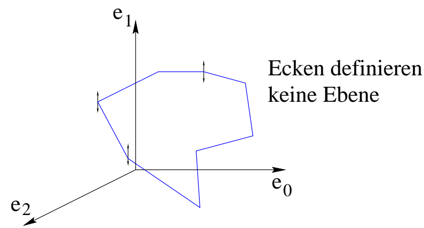
Ebenso analog wie beim Polygon definieren wir:

- Ein Punkt liegt außerhalb eines konvexen Polyeders, wenn er bezüglich aller seiner Fassetten oberhalb liegt.
- Ein Punkt liegt außerhalb des Polyeders, wenn die Anzahl der von einem Strahl geschnittenen Fassetten gerade ist, sonst innerhalb (‘odd-even Methode’).

Oder anders ausgedrückt, der Normalenvektor einer Fassette zeigt nach oben bzw. nach außen.

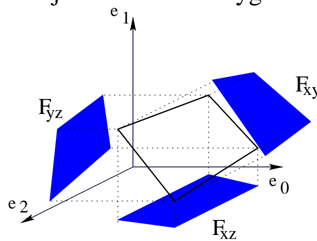
12.5 Normalenvektorberechnung von fast planaren Polygonen

- Die Ecken eines im 3-Dimensionalen gegebenen Polygons liegen u.U. nicht genau in einer Ebene
  - Rechenungenauigkeiten
  - ungenaue Eingabewerte
- Obige Rechnung mit Kreuzprodukt zweier Kantenvektoren setzt voraus, dass die Kantenvektoren nicht linear abhängig sind; diese muss man erst einmal bestimmen.



Deshalb folgende - numerisch stabile - Methode um den Normalenvektor zu bestimmen:

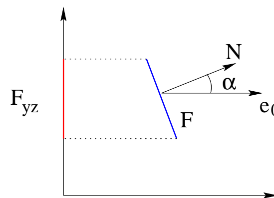
Berechnen die Flächen der Projektionen des Polygons in die drei Koordinatenebenen:



Ein Normalenvektor ergibt sich dann als

$$\vec{N} = \begin{pmatrix} F_{yz} \\ F_{xz} \\ F_{xy} \end{pmatrix}$$

Das sieht man wie folgt ein:



$$\begin{aligned} F_{yz} &= F \cdot \cos\alpha \\ &= F \cdot \langle \vec{N}, \vec{e_0} \rangle / |\vec{N}| \\ &= F \cdot N_x \end{aligned}$$

also

$$N_x = F_{yz}/F$$

und analog

$$\begin{aligned} N_y &= F_{xz}/F \\ N_z &= F_{xy}/F \end{aligned}$$

und der normierte Normalenvektor ist:

$$\mathbf{N} = \frac{1}{F} \cdot \begin{pmatrix} F_{yz} \\ F_{xz} \\ F_{xy} \end{pmatrix}$$

insbesondere ergibt sich auch:

$$F = \sqrt{F_{xy}^2 + F_{xz}^2 + F_{yz}^2}$$

12.5.1 Quadriken

Unter Quadriken versteht man eine Menge von Objekten, die durch folgende Gleichung beschrieben werden können:

Auf diese Art und Weise können Kugeln, Ellipsoide, Hyperboloide, Paraboloide, Zylinder und Tori modelliert werden.

Wir betrachten hier und im folgenden jedoch nur eine Auswahl dieser Quadriken: Kugel, Ellipsoid, und Torus.

12.6 Kugel

Für einen Punkt der Oberfläche gilt:

$$\begin{aligned} |\vec{x} - \vec{c}| &= r \\ \langle \vec{x} - \vec{c}, \vec{x} - \vec{c} \rangle &= r^2 \end{aligned}$$

F"ur $\vec{c} = \vec{0}$ erh"alt man

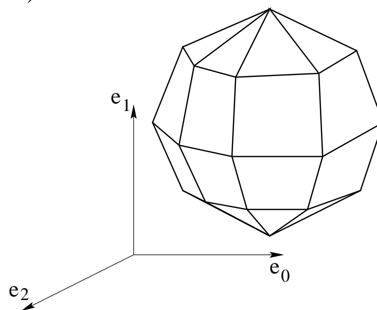
$$x^2 + y^2 + z^2 = r^2$$

oder in Polarkoordinaten f"ur $\phi \in [0, 2\pi[$ und $\delta \in [-\pi/2, \pi/2]$

$$\begin{aligned} x &= r \cdot \sin \delta \, \cos \phi \\ y &= r \cdot \sin \delta \\ z &= r \cdot \cos \delta \, \sin \phi \end{aligned}$$

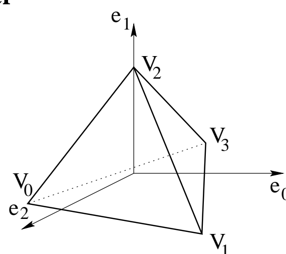
Darstellung einer Kugel z.B. durch Umwandlung der Kugel in ein Polyeder.

Klassische Methode durch Diskretisierung von Azimut und Elevation (hier nur die sichtbaren Kanten dargestellt):



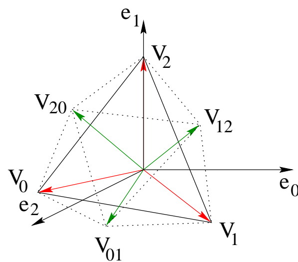
Andere Methode durch sukkzessive Unterteilung eines regulären Grundkörpers.

z.B. Grundkörper **Tetraeder**

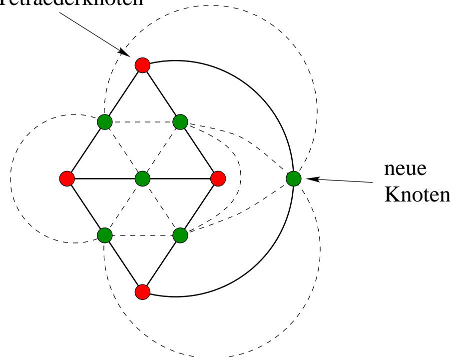


jede Kante wird in der Mitte geteilt und die entstehende Ecke auf die Kugeloberfläche projiziert:

Am Beispiel eines Dreiecks des Tetraeders:

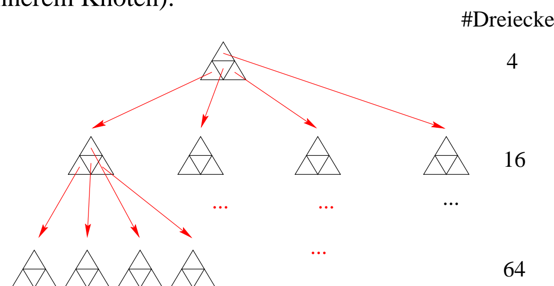


oder in der Darstellung als planarer Graph für eine Unterteilung der ersten Stufe:



Die Knoten des planaren Graphen haben alle Grad 6 mit der Ausnahme der ursprünglichen Tetraeder-Knoten, die nur Grad 3 haben.

Es ergibt sich durch Fortsetzen der Unterteilung der Dreiecke ein Baum mit Grad 5 (d.h. 4 Kinder je innerem Knoten):



Und man kann je nach gewünschter Feinheit der Unterteilung entsprechend alle Dreiecke in einer bestimmten Stufe des Baumes zur Darstellung heranziehen.

Statt des Tetraeders kann ein anderer platonischer Körper als Grundkörper herangezogen werden: **Körper#EckenFassettenform** Tetraeder4Dreiecke Würfel8Vierecke Oktaeder6Dreiecke Dodekaeder20Fünfecke Ikosaeder12Dreiecke Vorzugsweise nimmt man einen mit Dreiecken als Fassettenform, ansonsten muss man vorher entsprechend in Dreiecke zerlegen.

Beim Oktaeder steigt der Grad der ersten Knoten im planaren Graphen auf 4, beim Ikosaeder sogar auf 5, alle weiteren sukkzessive konstruierten Knoten (Ecken) haben dann Grad 6.

12.7 Ellipsoid

Die Gleichungen eines Ellipsoiden lauten:

Implizite Gleichung eines Ellipsoiden:

$$\left(\frac{x}{r_x}\right)^2 + \left(\frac{y}{r_y}\right)^2 + \left(\frac{z}{r_z}\right)^2 = 1$$

oder mit Skalierungsmatrix M_s

$$\langle M_s\vec{X}, M_s\vec{X}\rangle = r^2$$

wobei die Skalierungsmatrix folgende Form hat

$$M_s = \begin{bmatrix} r/r_x & 0 & 0 \\ 0 & r/r_y & 0 \\ 0 & 0 & r/r_z \end{bmatrix}$$

Parametrische Gleichung eines Ellipsoiden:

$$\vec{X} = \begin{pmatrix} r_x \cos\delta \cos\phi \\ r_y \sin\delta \\ r_z \cos\delta \sin\phi \end{pmatrix}$$

mit $\phi \in [0, 2\pi[$ und $\delta \in [-\pi/2, \pi/2]$.

Ein Ellipsoid ist also nichts anderes als eine skalierte Kugel.

12.8 Torus

Es gibt noch mehr Quadriken (z.B. Paraboloid, Hyperboloid usw.) deren Beschreibungen ähnlich erfolgen, auf die an dieser Stelle jedoch nicht näher eingegangen wird.

12.9 Superquadriken

Superquadriken stellen eine Verallgemeinerung der Quadriken dar. Mithilfe Einführung weiterer besonders gewählter Parameter lässt sich die ursprüngliche Form der Quadrik modifizieren.

Auch hier betrachten wir nur eine Auswahl der Quadriken.

12.10 2-dimensionale Superquadriken

Superellipse

Implizite Gleichung einer Ellipse:

$$\left(\frac{x}{r_x}\right)^2 + \left(\frac{y}{r_y}\right)^2 = 1$$

wir f"uhren einen Parameter $s \neq 0$ wie folgt ein:

$$\left(\frac{x}{r_x}\right)^{\frac{2}{s}} + \left(\frac{y}{r_y}\right)^{\frac{2}{s}} = 1$$

oder auch in der parametrischen Gleichung:

$$\begin{aligned} x &= r_x \cos^s \phi \\ y &= r_y \sin^s \phi \end{aligned}$$

mit $\phi \in [0, 2\pi[$, was man durch Einsetzen leicht verifiziert:

$$\begin{aligned} \left(\frac{r_x \cos^s \phi}{r_x}\right)^{\frac{2}{s}} + \left(\frac{r_y \sin^s \phi}{r_y}\right)^{\frac{2}{s}} &= 1 \\ \cos^2 \phi + \sin^2 \phi &= 1 \end{aligned}$$

Beispiel: Superkreis

12.11 3-dimensionale Superquadriken

Analog wie im 2-Dimensionalen führen wir - um ein Superellipsoid zu erhalten - zwei Parameter ein:

Implizite Gleichung eines Ellipsoiden:

$$\left(\frac{x}{r_x}\right)^2 + \left(\frac{z}{r_z}\right)^2 + \left(\frac{y}{r_y}\right)^2 = 1$$

wir f"uhren zwei Parameter $s_1, s_2 \neq 0$ wie folgt ein:

$$\left(\left(\frac{x}{r_x}\right)^{\frac{2}{s_1}} + \left(\frac{x}{r_x}\right)^{\frac{2}{s_1}}\right)^{\frac{s_1}{s_2}} + \left(\frac{y}{r_y}\right)^{\frac{2}{s_2}} = 1$$

oder auch in der parametrischen Gleichung:

$$\begin{aligned} x &= r_x \cos^{s_1}\delta \cos^{s_2}\phi \\ z &= r_z \cos^{s_1}\delta \sin^{s_2}\phi \\ y &= r_y \sin^{s_2}\delta \end{aligned}$$

was man auch durch Einsetzen leicht verifiziert.

13 Kurven und gekrümmte Oberflächen

Idee: Erzeuge Kurve bzw. gekrümmte Oberfläche mithilfe einiger Kontrollpunkte und eventuell zusätzlicher Parameter.

Man beschreibt die Kurve (Fläche) als Funktion über einem (über zwei) Parameter *u* (und *v*):

Man unterscheidet zwei prinzipielle Methoden:

- **Interpolation**, d.h. die Kurve verläuft durch die Kontrollpunkte, z.B. bei
  - kontinuierlicher Darstellung von Datensätzen
  - Durchlaufen von Schlüsselpositionen
- **Approximation**, d.h. die Kurve verläuft in der ¨Nähe¨ der Kontrollpunkte, z.B. bei
  - interaktiver Modellierung
  - Ausgleichen/Glätten von Datensätzen

Wie erzeugt man nun Kurven?

13.1 Kurven

Wir betrachten an dieser Stelle lediglich die Darstellung von Kurven als Polynome über einem Parameter *u*.

Es gibt natürlich auch andere Funktionen, mit denen Kurven dargestellt werden können, z.B.:

- trigonometrische Funktionen
- Exponentialfunktionen

die auch zu Interpolations- und Approximationszwecken herangezogen werden können. Diese werden aber hier nicht weiter angesprochen.

13.2 Polynominterpolation

13.3 Zusammengesetzte Kurven

Interpolation durch Polynome hohen Grades führt unter Umständen zu starken Oszillationen der Kurve, welche oft unerwüscht sind.

Man setzt eine Kurve deshalb zweckmäßigerweise aus Kurvenstücken zusammen, die durch Polynome niedrigen Grades erzeugt werden.

Bei der Zusammensetzung ist man an bestimmten Stetigkeits- und Glattheitseigenschaften interessiert.

Dies gilt in analoger Weise auch für die Approximation, obwohl man hier stets einfach auch ein Polynom niedrigen Grades verwenden kann (z.B. Regressiongerade) und nur versucht, den Fehler der Approximation, der an den Kontrollpunkten gemacht wird, ¨irgendwie¨ zu minimieren. Wir betrachten diese Art der Approximation hier jedoch nicht weiter.

13.4 Eigenschaften zusammengesetzter Kurven

Man unterscheidet:

- Eigenschaft im Parameterraum, sogenannte *C*-Eigenschaft
- Eigenschaft im Punktraum, sogenannte *G*-Eigenschaft

13.5 *C*-Eigenschaft

Das heißt, die Kurve *h* hat

- *Cº*-Eigenschaft, wenn sie bzgl. des Parameters *u* stetig zusammensetzbar ist;
- *C¹*-Eigenschaft, wenn sie bzgl. des Parameters *u* differenzierbar zusammensetzbar ist;
- *C²*-Eigenschaft, wenn sie bzgl. des Parameters *u* zweifach-differenzierbar zusammensetzbar ist.

Bemerkung: Beachte, die parametrischen Gleichungen sind Vektorgleichungen in mehreren Koordinaten; die geforderten Eigenschaften müssen für alle Koordinatengleichungen gelten.

13.6 *G*-Eigenschaft

Hier beschränkt man sich auf geometrische Eigenschaften im Punktraum und fordert lediglich, dass sich Kurvenstücke in den Endpunkten berühren, und dort bestimmte Tangenten oder Krümmungen aufweisen.

Betrachten wir die Tangentengleichungen der Kurvenstücke:

Wir fordern also nur, dass die Normalenvektoren linear abhängig sind, nicht jedoch unbedingt gleichen Betrag haben.

Bemerkungen:

- Die Normalenvektoren können durchaus entgegengesetzte Richtung haben!
- Eine Kurve kann, obwohl sie *G¹*-Eigenschaft (oder auch *C¹*-Eigenschaft) hat, ¨Ecken¨ aufweisen. Um dies zu verhindern, muss man weitere Forderungen an die Funktion im Parameterraum stellen!

13.7 Worin liegt der Unterschied?

Man macht sich den Unterschied zwischen der *G*- und der *C*-Eigenschaft am besten klar, wenn man sich den Parameter *u* als Zeit vorstellt unter welcher man sich entlang der Kurve bewegt:

- *Cº*-Eigenschaft: Stetigkeit bzgl. Ort
- *C¹*-Eigenschaft: Stetigkeit bzgl. Geschwindigkeit
- *C²*-Eigenschaft: Stetigkeit bzgl. Beschleunigung

Mit geeigneter Reparametrisierung kann man jede Kurve mit *G*-Eigenschaft in eine Kurve mit entsprechender *C*-Eigenschaft überführen, d.h.

Wir wollen jedoch auf diese Reparametrisierungen nicht näher eingehen.

Um nun keine unerwünschten, abrupten Änderungen im Verlauf einer Animation oder einer parametrischen Textur zu erhalten, sollte man in diesen Fällen Kurven und Flächen mit entsprechender *C*-Eigenschaft wählen.

Für einfache geometrische Modellierung von Objekten genügt *G*-Eigenschaft.

Da man sich in der Computer Grafik meist höchstens *C²*- bzw. *G²*-Eigenschaft fordert, genügen Interpolationen oder Approximationen mit höchstens kubischen Funktionen.

13.8 Splines

Man unterscheidet u.a. folgende Spline-Typen:

- kubische Splines
- Hermitsche Splines
- Cardinal-Splines
- Catmull-Rom-Splines
- Bézier-Splines
- B-Splines
- Beta-Splines
- Rationale Splines

Bemerkung: Verwendet man jeweils kubische Polynome als Grundlage der Splines, so kann man natürlich jeden Typ in einen anderen umrechnen, hierzu später mehr.

13.9 Spline-Repräsentation

Wir verwenden drei Strategien um Splines zu repräsentieren:

- Formulieren einer Reihe von Randbedingungen
- Matrix-Repräsentation
- Darstellung mit Gewichtsfunktionen (auch Basisfunktionen oder 'blending functions' genannt)

Man kann jeweils eine Darstellung in eine andere umrechnen.

13.10 kubische Splines

Matrixschreibweise für x-Koordinaten Polynom:

Analog kann man dies auch für das y-Koordinaten Polynom schreiben und erhält in Vektorschreibweise:

Wir können C wie folgt umschreiben:

13.11 Hermite-Splines

13.12 Cardinal-Splines

Anders als bei den Hermite-Splines fordern wir, dass die Tangenten an den Kontrollpunkten 1 bzw. 2 zu den Sekanten durch die Punkte 0 und 2 bzw. 1 und 3 parallel verlaufen.

Eine Cardinal-Spline liegt also den Kontrollpunkten 1 und 2, die er auch interpoliert.

Wir fordern nur Parallelität der Sekanten mit den Tangenten, weshalb wir einen Freiheitsgrad erhalten, den wir mit s festlegen. Als Randbedingungen der Kurve mit vier Kontrollpunkten erhalten wir:

$$\begin{aligned} p(0) &= b_1 \\ p(1) &= B_2 \\ p'(0) &= 1/2 \cdot (1-t)(b_2 - b_0) \\ p'(1) &= 1/2 \cdot (1-t)(b_3 - b_1) \end{aligned}$$

wobei man auch schreibt:

$$s = 1/2 \cdot (1-t)$$

Mit analoger Methode wie bei den Hermite-Splines können wir den Spline in Matrixschreibweise formulieren:

$$\begin{aligned} p(u) &= [u^3\ u^2\ u\ 1] \cdot M_C \cdot \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} \\ M_C &= \begin{pmatrix} -s & 2-s & s-2 & s \\ 2s & s-3 & 3-2s & -s \\ -s & 0 & s & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \end{aligned}$$

und lesen direkt die entsprechenden Basisfunktionen ('cardinal blending functions') ab:

$$\begin{aligned} B_0(u) &= -s\,u^3 + 2s\,u^2 - s\,u \\ B_1(u) &= (2-s) \cdot u^3 + (s-3) \cdot u^2 + 1 \\ B_2(u) &= (s-2) \cdot u^3 + (3-2s) \cdot u^2 + s\,u \\ B_3(u) &= s\,u^3 - s\,u^2 \end{aligned}$$

Spezialfälle der Cardinal-Splines sind die Catmull-Rom-Splines, bei den t=0 gesetzt ist.

13.13 Bézier-Splines

Wurden von Bézier bei Renault zum Design von Autokarrosserien erfunden.

Betrachten wir zuerst die allgemeine Form mit *n+1* Kontrollpunkten. Sie nutzen die Bernstein-Polynome als Basisfunktionen:

Beziér-Splines:

$$p(u) = \sum_{k=0}^{n} b_k \; B_{k,n}(u)$$

wobei

$$B_{k,n}(u) = \binom{n}{k} u^k \; (1-u)^{n-k}$$

die Bernstein-Polynome sind und b_k f"ur $k = 0, \ldots, n$ die $n+1$ Kontrollpunkte sind.

Für die Binomial-Koeffizienten gelten die folgenden Regeln:

$$\begin{aligned}
\binom{n}{k} &= \begin{cases} \frac{n!}{k!(n-k)!} & 0 \le k \le n \\ 0 & \text{sonst} \end{cases} \\
\binom{n}{k} &= \binom{n-1}{k} + \binom{n-1}{k-1} \\
\binom{n}{k} &= \frac{n-k+1}{k} \binom{n}{k-1}
\end{aligned}$$

Damit lassen sich die folgenden Eigenschaften der Bernstein-Polynome herleiten:
Und es ergeben sich die folgenden Eigenschaften der Bézier-Splines:

- Die Endpunkte werden interpoliert

$$\begin{aligned}
p(0) &= b_0 \\
p(1) &= b_n
\end{aligned}$$

- Sie liegt vollständig in der konvexen Hülle der Kontrollpunkte, da für jeden Parameterwert *u* das Polynom eine konvexe Kombination der Kontrollpunkte darstellt.
- Die Steigung in den Endpunkten ist identisch mit der Steigung der Geraden durch die jeweils beiden letzten Kontrollpunkte.

$$\begin{aligned}
p'(0) &= n \cdot (b_1 - b_0) \\
p'(1) &= n \cdot (b_n - b_{n-1})
\end{aligned}$$

- Alle Kontrollpunkte sind notwendig, um den Verlauf der Kurve zu bestimmen.
- Sie sind invariant gegenüber affinen Transformationen:

$$\begin{aligned}
\Phi\left(\sum_{k=0}^{n} b_k \; B_{k,n}(u)\right) &= A \cdot \left(\sum_{k=0}^{n} b_k \; B_{k,n}(u)\right) + \vec{c} \\
&= \sum_{k=0}^{n} A \cdot b_k \; B_{k,n}(u) + \vec{c} \cdot \sum_{k=0}^{n} B_{k,n}(u)
\end{aligned}$$

$$= \sum_{k=0}^{n} B_{k,n}(u) \; (A \cdot b_k + \vec{c})$$

$$= \sum_{k=0}^{n} B_{k,n}(u) \; \Phi(b_k)$$

D.h. man kann erst die Kontrollpunkte transformieren und dann die Kurve konstruieren.

Einige Nachteile der Bézier-Splines:

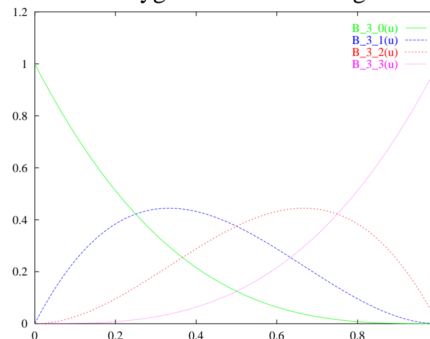
- Alle Kontrollpunkte werden beim Konstruieren verwendet
  –> keine lokale Kontrollmöglichkeiten
- Der Grad des resultierenden Polynoms hängt nur von der Anzahl der Kontrollpunkte ab.
- Bilden die Kontrollpunkte keinen einfachen Polygonzug, so können ¨Spitzen¨ im Kurvenverlauf entstehen.

  Bemerkung: Die Kurve ist als Polynom im Parameterraum sowohl bezüglich der x-Koordinate als auch der y-Koordinate differenzierbar.
- Mit steigender Anzahl von Kontrollpunkten steigt der Rechenaufwand pro zu berechnendem Punkt.

13.14 Kubische Bézier-Splines

Betrachten wir also wieder eine zusammengesetzte Spline-Kurve aus kubischen Funktionen.

Die entsprechenden Bernstein-Polygone sehen wie folgt aus:



13.15 Konstruktionstechniken:

- Identifiziert man den ersten und den letzten Kontrollpunkt zweier aufeinanderfolgender Abschnitte, so erhät man *Gº* (und somit *Cº*) Eigenschaft.
- Liegen der vorletzte Kontrollpunkt eines Abschnitts, der gemeinsame Kontrollpunkt, und der zweite Punkte des nächsten Abschnitts auf einer Geraden, so erhät man *G¹* Eigenschaft (d.h. Tangenten sind gleich).
- Sind zusätzlich auf der Geraden die Abstände gleich, so erhält man *C¹* Eigenschaft (Ableitungen im Parameterraum sind gleich).
- Entsprechend kann man auch Bedingungen an die Kontrollpunkte stellen um *C²* Eigenschaft zu erhalten; allerdings schränkt dies die zur Modellierung zur Verfügung stehenden Freiheitsgrade sehr ein.
  (Wir gehen hier allerdings nicht weiter auf Krümmungen von Kurven ein.)

13.16 Matrix-Schreibweise:

In Matrix-Schreibweise erhalten wir:

13.17 B-Splines

B-Splines sind eine Verallgemeinerung der Bézier-Splines; sie bieten erweiterte Kontrollmöglichkeiten:

- Der Grad des Polynoms kann unabhängig von der Anzahl der Kontrollpunkte gewählt werden.
- Die Kontrollpunkte bestimmen den Verlauf der Kurve nur in einem lokalen Bereich der gesamten Kurve.

Mit Basisfunktionen schreiben wir das Polynom als:

Die Rekursionsformeln nach Cox-DeBoor für die Basisfunktionen lauten:

D.h. die Basisfunktionen mit $d=1$ sind Box-Funktionen:

Eigenschaften der B-Splines:

Damit ist der Einfluss der Kontrollpunkte auf einen Bereich der Kurve beschränkt, der durch d aufeinanderfolgende Intervalle des Knotenvektors bestimmt wird.

Weiterhin definieren die Basisfunktionen in einem Intervall des Knotenvektors eine konvexe Kombination:

B-Splines haben also die gleichen Eigenschaften wie Bézier-Splines, nämlich:

- Invarianz bezg. affiner Transformationen
- Kurve liegt in der konvexen Hülle der Kontrollpunkte

13.18 Klassifizierung der B-Splines

Man klassifiziert die B-Splines je nach Typ des Knotenvektors in

- uniforme B-Splines
- offen uniforme B-Splines
- nicht-uniforme B-Splines

13.19 uniforme B-Splines

die Differenzen jeweils zweier aufeinanderfolgender Einträge des Knotenvektors sind gleich:

Uniforme B-Splines sind *periodisch*, d.h. alle Basisfunktionen sind jeweils verschobene Versionen einer Grundfunktion:

Dass dies gilt, rechnet man leicht nach:

13.20 Beispiel: quadratischer B-Spline

Die Basisfunktionen haben folgenden Verlauf:

13.21 Bemerkungen

- Man kann den gleichen Kontrollpunkt mehrfach angeben,
  dadurch wird die Kurve näher zu diesem Kontrollpunkt hingezogen.
- Lässt man für d auch 1 zu, so erhält man als ¨Kurve¨ gerade die diskreten Kontrollpunkte.

13.22 Beispiel: kubischer B-Spline

oder in Matrix-Schreibweise:

Man kann damit leicht berechnen:

Und man sieht, dass die B-Splines in gewisser Weise ähnlich zu den Cardinal-Splines sind, nur dass die Randkontrollpunkte nicht interpoliert werden.

13.23 Beta-Splines

Beta-Spline stellen eine Verallgemeinerung der B-Splines dar.

Idee: Lege Bedingung (z.B. eine G-Eigenschaft) für Knotenvektorpunkte fest.

Beispiel: *G¹*-Eigenschaft:

$$p'_{j-1}(u_j) = p'_j(u_j) \quad \text{f'ur } j \text{ geeignet}$$

dabei bezeichnet p_{j-1} das Polynom im Intervall $[u_{j-1}, u_j]$ und p_j das Polynom im Intervall $[u_j, u_{j+1}]$ (au"serhalb der Intervalle verschwinden die B-Spline-Basisfunktionen).

13.24 Rationale Splines

Rationale Splines lassen sich nicht mehr durch einfache Polynome über einem Parameter darstellen, vielmehr werden Quotienten von Polynomen betrachtet:

$$p_n(u) = \frac{\sum_{k=0}^{n} \omega_k b_k B_{k,d}(u)}{\sum_{k=0}^{n} \omega_k B_{k,d}(u)}$$

wobei f'ur $k = 0 \ldots n$

$$\begin{array}{ll} b_k & n+1 \text{ Kontrollpunkte sind,} \\ \omega_k & n+1 \text{ Gewichtsfaktoren sind,} \\ B_{k,d}(u) & \text{B-Spline Basisfunktionen vom Grad } d \text{ sind.} \end{array}$$

Für die Gewichtsfaktoren gilt wieder, dass sie sich zu 1 aufsummieren.

Für die B-Spline-Basisfunktionen wird meist eine nicht-uniforme Aufteilung des Knotenvektors vorgenommen, weshalb man die entstehenden Kurven allgemein als NURBS ('non-uniform rational B-splines') bezeichnet.

13.25 Beispiel für einen NURBS

Verwenden wir als Basisfunktionen quadratische B-Splines mit offenem, uniform unterteiltem Knotenvektor. Genauer:

$$\begin{array}{rcl} d & = & 3 \\ n & = & 2 \quad \text{d.h. 3 Kontrollpunkte} \\ u & = & (0,0,0,1,1,1) \\ \omega_0 & = & \omega_2 \; = \; 1 \\ \omega_1 & = & r/(r-1) \quad \text{f'ur} \quad r \leq 0 < 1 \end{array}$$

Wir erhalten als Gleichung für die Kurve:

$$p(u) = \frac{b_0 \; B_{0,3}(u) + r/(r-1) \cdot b_1 \; B_{1,3}(u) + b_2 \; B_{2,3}}{B_{0,3} + r/(r-1) \cdot B_{1,3} + B_{2,3}}$$

Vorteile der NURBS:

- Quadriken (z.B. Kreise und Ellipsen) können *exakt* nachgebildet werden.
- NURBS sind invariant gegenüber Projektionen (die vorherigen Splines waren nur invariant gegenüber affinen Transformationen).
  Hierauf gehen wir allerdings nicht weiter ein.

13.26 Darstellung von Splines

Wir zeichnen einen Spline als Polygonzug, wobei wir drei Varianten betrachten, um die Ecken des Polygonzuges zu berechnen:

- Polynomauswertung
  (geeignet z.B. bei nicht äquidistanten Parameterwerten)
- Vorwärtsdifferenzen
  (sehr schnelle Methode, geeignet z.B. bei äquidistanten Parameterwerten)
- Unterteilungsmethode
  (geeignet z.B. zur adaptiven Darstellung)

13.27 Polynomauswertung

13.28 Vorwärtsdifferenzen

Wir m"ochten das Polynom (hier in der x-Koordinate)

$$p_x(u) = a_x\ u^3 + b_x\ u^2 + c_x\ u\ + d_x$$

an den $u_0, u_1, \dots u_k, u_{k+1}, \dots$ auswerten, wobei stets gilt:

$$u_{k+1} = u_k + \delta$$

Wir lassen abk"urzend den Index x bei den Koeffizienten a_x, b_x, c_x, d_x sowie den Index k beim Parameterwert u_k weg und schreiben statt $p_x(u_k)$ einfach x_k.

Und erhalten für zwei aufeinanderfolgende Parameterwerte:

$$\begin{aligned} x_k &= a\,u^3 + b\,u^2 + c\,u\ + d \\ x_{k+1} &= a \cdot (u+\delta)^3 + b \cdot (u+\delta)^2 + c \cdot (u+\delta) + d \end{aligned}$$

Die Differenz erster Ordnung ergibt sich zu:

$$\begin{aligned} \Delta x_k &= x_{k+1} - x_k \\ &= 3a\delta u^2 + (3a\delta^2 + 2b\delta) \cdot u + (a\delta^3 + b\delta^2 + c\delta) \end{aligned}$$

und damit die Differenz zweiter Ordnung

$$\begin{aligned} \Delta^2 x_k &= \Delta x_{k+1} - \Delta x_k \\ &= 3a\delta \cdot (u+\delta)^2 + (3a\delta^2 + 2b\delta) \cdot (u+\delta) + (a\delta^3 + b\delta^2 + c\delta) \\ &\quad - 3a\delta u^2 - (3a\delta^2 + 2b\delta) \cdot u - (a\delta^3 + b\delta^2 + c\delta) \\ &= 6a\delta^2\ u + 6a\delta^3 + 2b\delta^2 \end{aligned}$$

und schließlich die Differenz dritter Ordnung:

$$\begin{aligned} \Delta^3 x_k &= \Delta^2 x_{k+1} - \Delta^2 x_k \\ &= 6a\delta^3 \end{aligned}$$

welche unabhängig vom Parameter *u* konstant ist.

Wir berechnen anfangs:

$$\begin{aligned}
x_0 &= a\, u_0^3 + b\, u_0^2 + c\, u_0 + d \\
\Delta x_0 &= 3a\delta u_0^2 + (3a\delta^2 + 2b\delta) \cdot u_0 + (a\delta^3 + b\delta^2 + c\delta) \\
\Delta^2 x_0 &= 6a\delta^2\, u_0 + 6a\delta^3 + 2b\delta^2 \\
\Delta^3 x_0 &= 6a\delta^3
\end{aligned}$$

Mithilfe der Vorwärtsdifferenzen kann man nun aufeinanderfolgende Werte des Polynoms in *u* mit lediglich drei Additionen berechnen.

$$\begin{aligned}
x_{k+1} &= x_k + \Delta x_k \\
\Delta x_{k+1} &= \Delta x_k + \Delta^2 x_k \\
\Delta^2 x_{k+1} &= \Delta^2 x_k + \Delta^3 x_k
\end{aligned}$$

wobei $\Delta^3 x_k$ eine Konstante ist!

13.29 Unterteilungsmethode

Die Berechnungsformeln für die neuen Kontrollpunkte ergeben sich als:

$$\begin{aligned}
b_{1,0} &= b_0 \\
b_{1,1} &= 1/2 \cdot (b_0 + b_1) \\
b_{1,2} &= 1/4 \cdot (b_0 + 2b_1 + b_2) \\
b_{1,3} &= 1/8 \cdot (b_0 + 3b_1 + 3b_2 + b_3) \\
\\
b_{2,0} &= 1/8 \cdot (b_0 + 3b_1 + 3b_2 + b_3) \\
b_{2,1} &= 1/4 \cdot (b_1 + 2b_2 + b_3) \\
b_{2,2} &= 1/2 \cdot (b_2 + b_3) \\
b_{2,3} &= b_3
\end{aligned}$$

13.30 Umrechnung von kubischen Splines

Die vorgestellten kubischen Splines sind eindeutig bestimmte Polynome dritten Grades. Man kann sie also von einer Darstellung eine andere umschreiben. Dies ist einfach, wenn man die Matrixschreibweise kennt:

Seien die beiden Matrixdarstellungen bekannt:

$$\begin{aligned}
p(u) &= [u^3\ u^2\ u\ 1] \cdot M_{spl1} \cdot M_{geo1} \\
q(u) &= [u^3\ u^2\ u\ 1] \cdot M_{spl2} \cdot M_{geo2}
\end{aligned}$$

Es muss f'ur alle u gelten:

$$p(u) = q(u)$$

Wir erhalten also:

$$M_{spl1} \cdot M_{geo1} = M_{spl2} \cdot M_{geo2}$$

und damit:

$$M_{geo2} = M_{spl2}^{-1} \cdot M_{spl1} \cdot M_{geo1}$$

D.h. die geometrische Randbedingung des zweiten Splines kann man aus der geometrischen Randbedingung des ersten Splines und den beiden Spline-Matrizen berechnen.

14 Komplexe Objekte

14.1 Translationsobjekte

Idee: Nehme eine 2-dimensionale Kurve und verschiebe sie entlang einer Geraden oder einer Kurve

Eine mögliche Methode der Darstellung ist wieder das Erzeugen eines Polyeders mit Oberflächenpunkten als Ecken und entsprechende Kanten und Fassetten. **Beachte:** es kann unter Umständen zu Selbstdurchdringungen kommen.

14.2 Rotationsobjekte

Idee: Nehme eine 2-dimensionale Kurve und rotiere sie um eine Achse.

Eine mögliche Methode der Darstellung ist wieder das Erzeugen eines Polyeders mit Oberflächenpunkten als Ecken und entsprechende Kanten und Fassetten. Weitere Variationen von Objekten erhält man, indem man die Basisfigur während der Translation oder Rotation modifiziert:

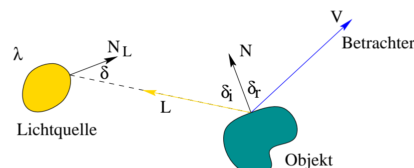
14.3 Gekrümmte Flächen

15 Licht und Farbe

15.1 Licht

¨Computergrafik¨-Modell des Lichtes:

- Licht breitet sich geradlinig aus (entweder als Welle oder als Menge von Teilchen)
- Spektrum verschiedener Frequenzen (Wellenlängen)
- Licht befindet sich gleichzeitig in seinem gesamten Ausbreitungsbereich
- gehorcht ¨einfachen¨ physikalischen Gesetzen (Snellsche Gesetz, Lambertsche Gesetz, Fresnelsche Gesetz)



Interaktion des Lichtes mit Objekten hängt ab:

- von der Wellenlänge (Frequenzspektrum) des Lichtes
- von der Geometrie zwischen Lichtquelle (Strahler) und Objekt
- vom Material des Objektes
- von der Oberflächenstruktur des Objektes

- dabei müssen Integrale über die Raumwinkel gebildet werden!
  (darauf gehen wir aber erst bei Radiosity ein, falls es soweit kommt ...)

Sehr einfaches Modell:

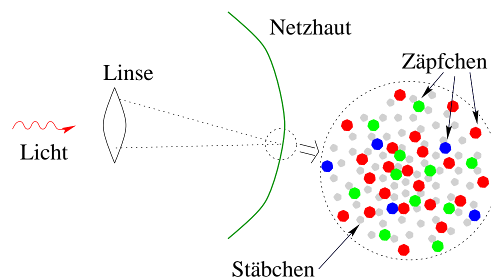
$$
\begin{aligned}
I &= F(I_L, \delta_i, \delta_r, c) \\
&= I_L \cdot \langle \vec{L}, \vec{N} \rangle \cdot \langle \vec{V}, \vec{N} \rangle \cdot c
\end{aligned}
$$

d.h. die dargestellte Intensit"at ist nur eine Funktion der Intensit"at der Lichtquelle I_L, deren Einfallswinkels δ_i, des Betrachtungswinkels δ_r und einer konstanten Materialeigenschaft c (Farbe).

Bemerkung: Je fotorealistischer die Darstellungen sein sollen, umso genauer und besser muss man die tatsächlichen physikalischen Gegebenheiten simulieren.

Wir werden später weitere Modelle kennenlernen.

15.2 Menschliches Wahrnehmungssystem



- Sinneszellen, Einteilung in
  - Zäpfchen
    * Anzahl ca. 6-7 Millionen
    * verantwortlich für Farbwahrnehmung
    * drei Klassen: rot/gelb-empfindlich (ca. 65%), grün-empfindlich (ca. 33%), blau-empfindlich (ca. 2%)
    * Verteilung weder bezüglich Ort noch Typ gleichmäßig (Sehzentrum ist stärker rot/gelb-empfindlich)
  - Stäbchen
    * Anzahl ca. 75-150 Millionen
    * verantwortlich für Hell/Dunkel-Wahrnehmung
    * Verteilung nicht gleichmäßig
    * hohe Kantenauflösung (Kontrastverstärkung)
- Wahrnehmungseigenschaften
  - sichtbares Frequenzspektrum im Bereich von 380-780nm
  - Dynamik der Intensitätswahrnehmung *10 hoch 10*
  - Anzahl unterscheidbarer Farben ca. 350000
  - ca. 10 Gigabit pro Sekunde Informationsverarbeitung des Auges
  - Intensitätsdifferenzauflösung ca. 2% (abhängig von Farbe und Hintergrundintensität)
  - logarithmisch subjektives Empfinden (d.h. den Unterschied zwischen Paaren von Intensitätswerten empfinden wir als gleich, wenn deren Verhältnis gleich ist)
  - viele verschiedene Frequenzspektren werden als die gleiche Farbe wahrgenommen

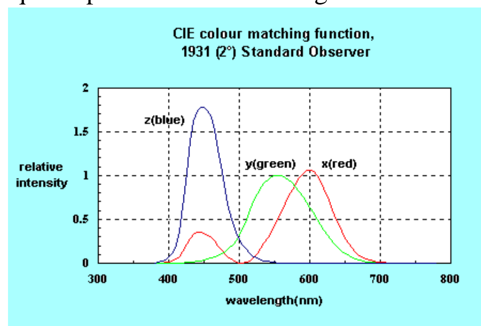
15.3 Intensität und Farbe

15.4 Klassifizierung von Farbe

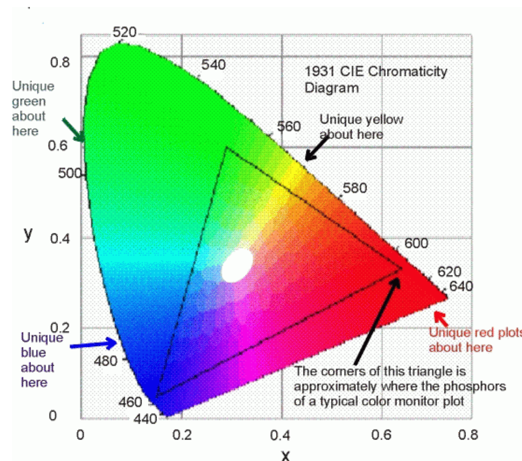
Wir beschreiben normalerweise Farbe durch folgende drei Begriffe mit entsprechender ¨Quantifizierung¨:

- Farbton ('hue'): rot, blau, braun, gelb ...
- Farbintensität ('luminance, brightness'): hell, licht, dunkel ...
- Farbsättigung ('purity, saturation'): himmel-, feuer-, ... blass, lebendig, ...

Experimente zeigen (CIE 1931), dass eine Kombination - einfache lineare gewichtete Superposition - von drei Frequenzspektren genügt, um fast alle wahrnehmbaren Farben zu erzeugen. Die Basisfrequenzspektren sehen wie folgt aus:



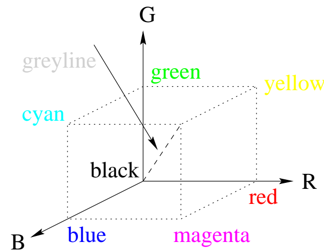
Normiert man die Intensität (Energie) des Lichtes, so kann man die Farben mit zwei Koordinaten repräsentieren und man erhält das folgende Standard-Diagramm ('color gamut'):



Die drei Frequenzspektren wurden als CIE-XYZ-Modell normiert und als internationale Referenz anerkannt. Möchte man konkret über eine Farbe reden, bezieht man sich auf diesen Standard.

Daneben gibt es verschiedene Farbmodelle, die je nach Anwendungsgebiet benutzt werden. Obwohl in diesen Modellen auch Koordinaten bezüglich dreier Grundfarben benutzt werden, ist die dargestellte Farbe auf verschiedenen Ausgabegeräten nicht notwendigerweise identisch. Der Hersteller des Ausgabegerätes sorgt jedoch üblicherweise dafür, dass möglichst viele Farben, d.h. ein großer Bereich, des Standard-Diagramms dargestellt werden können.

15.5 RGB-Modell

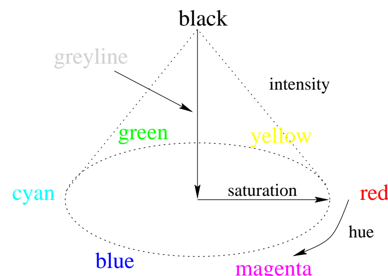


- Anordnung der Farben in einem achsenparallelen Farb-Würfel
- RGB = red-green-blue
- Ursprung ist Schwarz
- Gegenpunkt ist Weiß
- dazwischenliegende Diagonale kodiert Grauwerte (Graulinie)
- Einheiten

$$\begin{aligned} rgb &\in [0,1]^3 \subset \mathbb{R}^3 \\ &\in [0:255]^3 \subset \mathbb{N}^3 \end{aligned}$$

- Anwendung der Farbkodierung vorwiegend für
  - Bildspeicher und Farbtabellen von Monitoren
    (additive Farbmischung: alle Grundfarben zusammen ergeben Weiß)
  - Bildspeicherung in vielen Standardformaten

15.6 HSV-Modell



- Anordnung der Farben in einem Farb-Kegel (mit Scheibe oder Hexagon als Basisfläche)
- HSV = hue-saturation-(intensity)value
- Spitze ist Schwarz
- Zentrum der Basisfläche ist Weiß
- dazwischenliegende Achse kodiert Grauwerte (Graulinie)
- reine Farben liegen auf dem Rand der Basisfläche
- Kodierung lässt sich durch einfache geometrische Umrechnung aus dem RGB-Modell berechnen

- Anwendung der Farbkodierung vorwiegend für
  - interaktive Eingabesysteme, da nah an der Farbklassifizierung

Modifizierung als HLS-Modell, wobei ein Doppel-Farb-Kegel verwendet wird; dann befindet sich auch Weiß in einer Spitze.

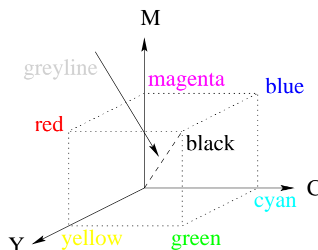
15.7 YIQ-Modell

- Kodierung der Farben mit Intensität Y und zwei Farbwerten I und Q
- Grauwerte können allein durch Y-Wert spezifiziert werden
- Berechnung aus und nach RGB-Modell durch standardisierte Matrix-Operationen

$$\begin{pmatrix} Y \\ I \\ Q \end{pmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ 0.596 & -0.274 & -0.322 \\ 0.211 & -0.522 & 0.311 \end{bmatrix} \cdot \begin{pmatrix} R \\ G \\ B \end{pmatrix}$$
$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{bmatrix} 1 & 0.956 & 0.623 \\ 1 & -0.272 & -0.648 \\ 1 & -1.105 & 1.705 \end{bmatrix} \cdot \begin{pmatrix} Y \\ I \\ Q \end{pmatrix}$$

- Diese Operationen sind nicht invers zueinander auf dem vollständigen Farbraum
- es existiert eine reversible ganzzahlige Approximation dieser Berechnung
- Anwendung der Farbkodierung vorwiegend für
  - Bildkodierung für Bildübertragung und Bildkompression
  - Fernsehtechnik

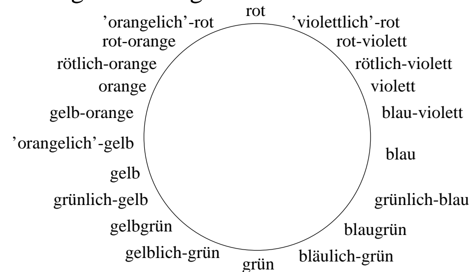
15.8 CMY-Modell



- Anordnung der Farben in einem achsenparallelen Farb-Würfel (ähnlich dem RGB-Modell)
- CMY = cyan-magenta-yellow
- Ursprung ist Weiß
- Gegenpunkt ist Schwarz
- dazwischenliegende Diagonale kodiert Grauwerte (Graulinie)
- Einheiten wie im RGB-Modell, Umrechnung durch einfache Vektoroperation CMY=Weiß-RGB bzw. RGB=Schwarz-CMY
- häufig wird zusätzlich Schwarz als vierte ¨Farbe¨ hinzugenommen
- Anwendung der Farbkodierung vorwiegend für
  - Bildkodierung für Drucker
    (subtraktive Farbmischung: alle Grundfarben zusammen ergeben Schwarz)

15.9 CNS-Modell

Farbton-Kodierung durch folgendes Diagramm:



- CNS = color naming system
- Farbintensitätskodierung durch Adjektive der Art:
  sehr hell, hell, mittel, dunkel, sehr dunkel
- Farbsättigungskodierung durch Adjektive der Art:
  blass, normal, stark, lebendig
- es lassen sich leicht mehr als 500 Farben kodieren
- die Farbintensität ist am schwierigstens zu quantifizieren (häufig auch durch Skalenvergleich mit Grauwertskala).
- Anwendung der Farbkodierung vorwiegend für
  - textuelle Beschreibung von Farben
  - sprachlicher Dialog über Farbe

Generell gilt: gute interaktive Systeme erlauben eine lineare Zuordnung der Farben gemäß der menschlichen Wahrnehmung.

15.10 Darstellung von Bildern

Wir repräsentieren also Graustufen durch einen Wert und Farben durch ein 3-Tupel. Wir gehen davon aus, dass die Werte in das Intervall *[0,1]* normiert sind. (Wie dies bei einem konkreten Rendering-Verfahren erreicht wird, sehen wir in einem späteren Abschnitt.)

Nun können die meisten Ausgabegeräte keine kontinuierlichen Werte darstellen, sondern man ist auf diskrete Werte pro Pixel beschränkt; zudem ist die Auflösung unterschiedlich:

Schwarz/Weiß-Drucker 1 bpp (bit per pixel), 300-1200 dpi (dots per inch)

Farb-Monitor 8-24 bpp, drei Farben, 80-120 dpi

Farb-Sublimations-Drucker 3 bpp, 80-120 dpi

Wir müssen somit folgendes tun:

- Diskretisierung der Farbe bzw. des Grauwertes (Quantisierung)
- ¨Erzeugen¨ zusätzlicher Farb- oder Grauwerte aus vorhandenen Werten ('halftoning' und 'dithering')
- Anpassung der Auflösung
- Fehlerkorrektur

15.11 Intensitätsdiskretisierung

Das menschliche Wahrnehmungssystem arbeitet nahezu logarithmisch, d.h. entscheidend ist nicht die Differenz sondern das Verhätnis.

Der Abstand diskreter Intensitätswerte sollte dem Rechnung tragen, und das Verhätnis aufeinanderfolgende Werte sollte konstant gehalten werden.

Sei I_0 die minimal darstellbar Intensit"at und I_n die maximal darstellbare Intensit"at. Es soll gelten:

$$\frac{I_1}{I_0} = \frac{I_2}{I_1} = \ldots = \frac{I_{k+1}}{I_k} = \ldots = \frac{I_n}{I_{n-1}} = r$$

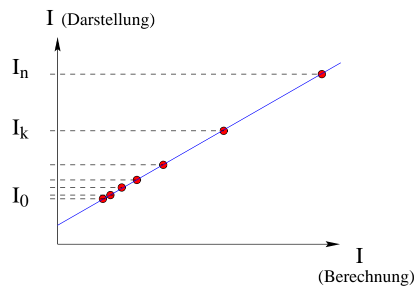
Die Zwischenwerte f"ur $k = 0 \ldots n$ berechnen sich also zu

$$I_k = r^k I_0$$

und r kann durch

$$r = (I_n/I_0)^{1/n}$$

berechnet werden.



15.12 Gamma-Korrektur

Die meisten Monitore weisen keinen linearen Zusammenhang zwischen anliegender Spannung (Eintrag in einer Farbtabelle) und Intensität auf dem Bildschirm auf.

Vielmehr kann man die Intensität besser mit einem exponentiellen Zusammenhang modellieren:

Sichtbare Intensit"at I ergibt sich zu

$$I = a \cdot V^\gamma$$

wobei a und γ Konstanten des Monitors sind und V die anliegende Spannung an der Bildr"ohre ist.

Um nun ein in einem linearen Modell berechnetes oder gespeichertes Bild adäquat darstellen zu können, muss man für einen gegebenen Monitor eine sogenannte *Gamma-Korrektur* durchführen:

Die korrekte Spannung f"ur eine Intensit"at I berechnet sich zu

$$V = (I/a)^{1/\gamma}$$

Typische Werte f"ur γ liegen zwischen 2 und 3.

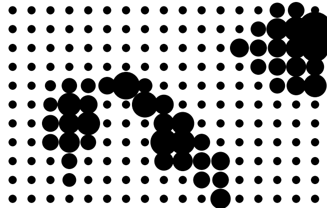
Es gibt auch Monitore, die bereits eine eingebaute Gamma-Korrektur aufweisen, d.h. im Bildspeicher liegen Werte bzw. die Einträge in den Farbtabellen werden korrekt dargestellt.

In diesem Fall muss man nur das logarithmische Empfinden beim Aufbau einer Tabelle berücksichtigen.

15.13 Anpassung an Ausgabegeräte

Stehen bei einem Ausgabegerät nur eine geringe Anzahl von Intensitätswerten/Farbwerten pro Pixel zur Verfügung, will man aber ein Bild mit vielen Werten darstellen, so versucht man durch spezielle Wahl der Intensitäten/Farben benachbarter Pixel beim Betrachter den subjektiven Eindruck von mehr Werten zu erzeugen.

Klassisches 'halftoning' nutzt schwarze Kreisscheiben mit Zentren auf einem regelmäßigen Raster und kontinuierlich einstellbarem Radius um viele Grauwerte zu erzeugen (Zeitungsdruck).

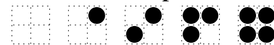


15.14 Halftoning

Für Rastergrafikgeräte kann man einen ähnlichen Effekt erzeugen:

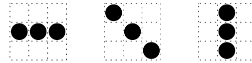
Idee: nutze Pixel-Muster zur Darstellung von Intensitätswerten oder Farbwerten.

Man fasst rechteckige Bereiche von Pixeln zusammen und setzt in diesem Bereich eine bestimmte Anzahl von Pixeln um so einen entsprechenden Grauwert zu erhalten:

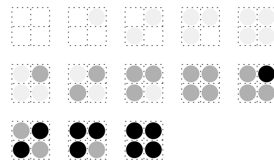


Für quadratische Bereiche mit einer Kantenlänge von n Pixeln stehen also n^2+1 viele Werte zur Verfügung.

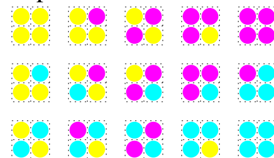
Beim Besetzen der Bereiche sollte man darauf achten, dass keine regelmäßigen Muster mit vorherrschenden Linien entstehen, d.h. folgende Muster sind z.B. möglichst zu vermeiden



Natürlich kann man das Verfahren auch anwenden, wenn mehr als eine Intensitätsstufe pro Pixel zur Verfügung steht:



oder wenn mehr als eine Farbe pro Pixel darstellbar ist (Beispiel):



Eigenschaften:

- n^2+1 viele Intensitätsstufen bei Pixelmustern mit quadratischen Bereichen der Kantenlänge n
- dabei Reduktion der Auflösung um Faktor n

Ohne Reduktion der Auflösung kommt Dithering aus.

15.15 Dithering

Idee: Kachele das Orginalbild mit einer ¨kleinen¨ Dithermatrix und setze in der Darstellung nur dann ein Pixel, wenn der Originalwert größergleich dem Matrixeintrag ist.

Wir betrachten Dithering nur für eine Darstellung als Schwarz/Weiß-Bild.

Dithermatrizen sind üblicherweise quadratisch und enthalten ganzzahlige Einträge von *0* bis *n²-1*, wobei *n* die Matrixgröße angibt. Mögliche Dithermatrizen sind:

$$D^{(2)} = \begin{pmatrix} 3 & 1 \\ 0 & 2 \end{pmatrix} \quad D^{(5)} = \begin{pmatrix} 21 & 10 & 17 & 14 & 23 \\ 15 & 2 & 6 & 4 & 9 \\ 20 & 5 & 0 & 1 & 18 \\ 12 & 8 & 3 & 7 & 13 \\ 24 & 18 & 19 & 11 & 22 \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 2 & 6 & 4 \\ 5 & 0 & 1 \\ 8 & 3 & 7 \end{pmatrix} \quad D^{(4)} = \begin{pmatrix} 0 & 8 & 2 & 10 \\ 12 & 4 & 14 & 6 \\ 3 & 11 & 1 & 9 \\ 15 & 7 & 13 & 5 \end{pmatrix}$$

Dithermatrizen höherer Ordnung kann man mithilfe folgender Rekursion berechnen:

$$D^{(2n)} = \begin{pmatrix} 4D^{(n)} + 3U^{(n)} & 4D^{(n)} + U^{(n)} \\ 4D^{(n)} & 4D^{(n)} + 2U^{(n)} \end{pmatrix}$$

mit $U^{(n)}$ sind vollbesetzte Matrizen mit 1-Eintr"agen.

Die Intensität des Originalbildes wird in das Intervall *[0,n2]* skaliert und ein Pixel wird gesetzt, wenn:

$$I(x,y) > D(x \bmod n, y \bmod n)$$

Eigenschaften:

- Auflösung bleibt erhalten
- die Größe der Dithermatrix bestimmt die Anzahl der darstellbaren Graustufen
- in der Praxis werden Dithermatrizen bis Größe 10 verwendet, d.h. nur Pixel mit Intensität oberhalb eines Schwellwertes werden gesetzt.
- man kann auch eine Dithermatrix mit nur einem Eintrag *0* verwenden
- Dithering ist parallelisierbar

15.16 Reduktion der Auflösung

Dies soll hier nur kurz erwähnt werden.

Man hat es hier mit dem generellen Abtast-Problem zu tun, und es hängt sehr von der konkreten Anwendung ab, welches Verfahren angewendet werden kann:

- Maximum (S/W Linien-Zeichnungen)
- Reguläres Abtasten (einfache Bilder mit großen Flächen)
- zufälliges Abtasten (???)
- Filtern (Bilder mit wenigen Kontrastkanten)

15.17 Fehlerkorrekturverfahren

Durch die Quantisierung der Farbwerte und durch das Dithering (insbesondere bei einer Dithermatrix mit *n=1*) entstehen Fehler, die man jedoch weitgehend ausgleichen kann, wobei allerdings Einbußen beim Kontrast hingenommen werden müssen.

Man unterscheidet:

- Fehlerverteilungsverfahren
- Fehlerdiffusionsverfahren

15.18 Fehlerverteilungsverfahren

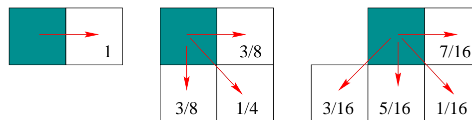
Idee: (nach Floyd-Steinberg) Der Fehler, der bei der Quantisierung bei einem Pixel entsteht, wird auf einige Nachbarpixel gewichtet verteilt, die noch nicht dargestellt worden sind.

Sei C_{org} Farb/Grau-Wert eines Pixels des Originalbild und C_{qua} der quantisierte Wert. Der zu verteilende Fehler berechnet sich zu:

$$\Delta C = \omega \cdot (C_{org} - C_{qua})$$

wobei $\omega \in [0, 1]$ ein zus”atzlicher Gewichtsfaktor ist.

Wir wollen keine Formeln schreiben, sondern die Fehlerverteilung anhand vom Grafiken veranschaulichen:



Um den Fehler auch am Ende eine Zeile verteilen zu können, läuft man abwechselnd von links nach rechts und dann von rechts nach links.

Eigenschaften:

- es können (bei feinen Strukturen) ¨Geisterbilder¨ entstehen
- inhärent sequentiell, d.h. nicht parallelisierbar

15.19 Fehlerdiffusionsverfahren

Idee: (nach Knuth) Kachele das Bild mit einer Diffusionsmatrix und verteile den Fehler bei der Quantisierung des Wertes eines Pixels auf alle Nachbar-Pixel mit größerem Eintrag in der Diffusionsmatrix.

Beispiele für Diffusionsmatrizen:

$$\begin{pmatrix} 34 & 48 & 40 & 32 & 29 & 15 & 23 & 31 \\ 42 & 58 & 56 & 53 & 21 & 5 & 7 & 10 \\ 50 & 62 & 61 & 45 & 13 & 1 & 2 & 18 \\ 38 & 46 & 54 & 37 & 25 & 17 & 9 & 26 \\ 28 & 14 & 22 & 30 & 35 & 49 & 41 & 33 \\ 20 & 4 & 6 & 11 & 43 & 59 & 57 & 52 \\ 12 & 0 & 3 & 19 & 51 & 63 & 60 & 44 \\ 24 & 16 & 8 & 27 & 39 & 47 & 55 & 36 \end{pmatrix} \quad \begin{pmatrix} 25 & 21 & 13 & 39 & 47 & 57 & 53 & 45 \\ 48 & 32 & 29 & 43 & 55 & 63 & 61 & 56 \\ 40 & 30 & 35 & 51 & 59 & 62 & 60 & 52 \\ 36 & 14 & 22 & 26 & 46 & 54 & 58 & 44 \\ 16 & 6 & 10 & 18 & 38 & 42 & 50 & 24 \\ 8 & 0 & 2 & 7 & 15 & 31 & 34 & 20 \\ 4 & 1 & 3 & 11 & 23 & 33 & 28 & 12 \\ 17 & 9 & 5 & 19 & 27 & 49 & 41 & 37 \end{pmatrix}$$

Einträge ohne größeren Nachbarn nennt man *Barone*, solche mit nur einem größeren Nachbarn *Fast-Barone*.

Bei der Diffusion des Fehlers auf die Nachbarn wird eine Gewichtung vorgenommen (siehe hierzu Literatur).

Eine weitere Verbesserung erreicht man durch eine Kantenverstärkung mithilfe folgender Operation:

$$I'(x,y) = \alpha \cdot I(x,y) + (1-\alpha) \cdot \bar{I}(x,y)$$

wobei $\bar{I}$ durch eine diskrete Konvolution des Orginalbildes entsteht:

$$\begin{aligned} \bar{I}(x,y) &= (I \star K)(x,y) \\ &= \sum_{u=-d}^{d} \sum_{v=-d}^{v=d} I(x+u, x+v) \cdot K(d-u, d-v) \end{aligned}$$

Eine einfache Konvolutionsmatrix zur Kantenverst"arkung mit $d = 1$ ist:

$$K = \begin{bmatrix} 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \end{bmatrix}$$

d.h. $\bar{I}(x,y)$ ist der Mittelwert von $I(x,y)$ mit allen seinen 8 Nachbarpixeln.

Eigenschaften:

- bei geringer Auflösung (Monitore) geringfügig schlechtere Qualität als Fehlerverteilungsverfahren
- bei hoher Auflösung (Drucker) deutlich bessere Qualität als Fehlerverteilungsverfahren
- gut parallelisierbar

16 3-dimensionale Darstellung

Unter ‘rendering’ versteht man die 2-dimensionale Darstellung 3-dimensionaler Objekte in einer Projektionsebene (‘rendering’). Bei der Betrachtung der erzeugten Darstellung ist der Beobachter gefragt, sich die wirkliche 3-dimensionale Szene vorzustellen.

Das zugrundeliegende Prinzip ist ähnlich dem des Fotografierens.

16.1 Koordinatentransformationen

Zweckmäßigerweise führt man mehrere Koordinatentransformationen durch, um letztendlich eine Darstellung auf dem Ausgabegerät (Drucker, Monitor) zu erhalten. Diese Vorgehensweise gestattet uns eine recht systematische Herangehensweise an eine Implementierung.

Bemerkungen:

- Einzelne Objekte sind häufig leicht mit eigenen, sogenannten Modell-Koordinaten (oder Objekt-Koordinaten) zu modellieren
- Alle Objekte liegen letztendlich in einem festen Welt-Koordinatensystem.
- Ist der Betrachtungsstandpunkt (Kamerapunkt) festgelegt, transformiert man die Welt entsprechend so, dass der Kamerapunkt zum Ursprung wird, und die Basis das Koordinatensystem der Kamera darstellt.
- Die immer noch 3-dimensionalen Kamera-Koordinaten werden dann - mit entsprechendem Kamera-Modell - auf die Bildebene projiziert.
- Die Bildebene wird auf dem Ausgabegerät dargestellt.
- Bei der Projektion und der Abbildung auf das Ausgabegerät wird die Sichtbarkeit von Objekten und die eigentliche Farbdarstellung (‘shading’) berechnet.

16.2 Kamera-Koordinatensystem

Kamera-Koordinatensystem:

$$(\vec{C}, \vec{U}, \vec{V}, \vec{N})$$

mit

$\vec{C}$	Kamera-Position (Kamera-Ursprung)
$\vec{N}$	Normale der Bildebene
$\vec{U}$	Horizontvektor
$\vec{V}$	Aufw"artsvektor (oder Zenitvektor)

und $(\vec{U}, \vec{V}, \vec{N})$ ist rechtsh"andige Orthonormalbasis.

16.3 Spezifikation einer Kamera

Man spezifiziert und berechnet in interaktiven Systemen das Kamera-Koordinatensystem beispielsweise wie folgt:

- Angabe der Kamera-Position
- Angabe eines Betrachtungspunktes ('point-of-interest' oder 'look-at-point')
- Angabe eines Punktes, der ¨Richtung oben¨ angibt

Damit berechnen sich die Basisvektoren wie folgt:

$$\begin{aligned} \vec{N} &= \vec{C} - \vec{L} \\ \vec{V} &= \vec{N} \times \vec{O} \times \vec{N} \\ \vec{U} &= \vec{V} \times \vec{N} \end{aligned}$$

mit anschlie"sender Normierung.

Bemerkung: Manchmal werden auch linkshändige Kamera-Koordinatensysteme verwendet!

16.4 Welt-Koordinaten in Kamera-Koordinaten

Die Umrechnung von Modell-Koordinaten in Welt-Koordinaten erfolgt durch entsprechende Transformationen, wie wir sie in vorigen Abschnitten beschrieben haben.

Die Umrechnung von Welt-Koordinaten in Kamera-Koordinaten ist wegen der orthonormalen Basen auch recht einfach:

$$\vec{U} = \begin{pmatrix} u_0 \\ u_1 \\ u_2 \\ 0 \end{pmatrix} \qquad \vec{N} = \begin{pmatrix} n_0 \\ n_1 \\ n_2 \\ 0 \end{pmatrix}$$

$$\vec{V} = \begin{pmatrix} v_0 \\ v_1 \\ v_2 \\ 0 \end{pmatrix} \qquad \vec{C} = \begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ 1 \end{pmatrix}$$

und damit Translationsmatrix

$$T_c = \begin{pmatrix} 1 & 0 & 0 & -c_0 \\ 0 & 1 & 0 & -c_1 \\ 0 & 0 & 1 & -c_2 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

und Rotationsmatrix

$$R_{uvn} = \begin{pmatrix} u_0 & u_1 & u_2 & 0 \\ v_0 & v_1 & v_2 & 0 \\ n_0 & n_1 & n_2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

und somit Transformationsmatrix f"ur Welt-Koordinaten in Kamera-Koordinaten

$$M_{cam} = R_{uvn} \cdot T_c$$

Dass diese Matrizen korrekt sind, sieht man leicht ein, denn die Kamera-Basisvektoren werden auf die kartesischen Basisvektoren abgebildet und die Kamera-Position auf den Ursprung:

$$\begin{aligned} R_{uvn} \cdot \vec{U} &= \vec{e}_0 \\ R_{uvn} \cdot \vec{V} &= \vec{e}_1 \\ R_{uvn} \cdot \vec{N} &= \vec{e}_2 \\ T_c \cdot \vec{C} &= \vec{0} \end{aligned}$$

16.5 Klassifizierung der Projektionen

Es gibt verschiedene Möglichkeiten die Projektion der 3-dimensionalen Objekte auf/in die 2-dimensionale Projektionsebene vorzunehmen:

- *parallele Projektion*
  Parallele Linien im Objektraum bleiben als parallele Linien in der Projektion erhalten.
  - *rechtwinklig*

    Die Projektionsmatrix der rechtwinkligen Projektion ergibt sich recht einfach als:

    $$M_{par\perp} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

    d.h. man kann einfach zur Berechnung der Projektion eines Punktes die z-Koordinate des Punktes weglassen.
    Man unterscheidet weiter (insbesondere für CAD Zwecke):
    * *Hauptrisse*
      Man wählt den Normalenvektor der Kamera parallel zu einer Koordinatenachse des Welt-Koordinatensystems und erhä;t so Seitenansichten.
    * *axonometrische Risse*
      Man wählt den Normalenvektor der Kamera so, dass die Projektionsebene mehrere Koordinatenachsen des Welt-Koordinatensystems schneidet:
      · *trimetrisch*
      alle Achsen werden geschnitten
      · *dimetrisch*
      nur zwei Achsen werden geschnitten
      · *isometrisch*
      alle Achsen werden geschnitten, und zwar so, dass die entstehenden Achsenabschnitte gleiche Länge haben; damit bleiben in der Projektion die original Längenverhältnisse erhalten.

– *schiefwinklig*

Damit können wir folgende Berechnungen durchführen:
Der projizierte Punkt $P' = (x', y', 0)$ berechnet sich mit den angegebenen Winkeln und L"angen wie folgt:

$$
\begin{aligned}
x' &= x + l \cos \phi \\
y' &= y + l \sin \phi
\end{aligned}
$$

wobei

$$\tan \alpha = z/l \quad \Longrightarrow \quad l = z/\tan \alpha = z \cdot l_1$$

und die Projektionsmatrix ergibt sich zu:

$$M_{para\alpha\phi} = \begin{pmatrix} 1 & 0 & l_1 \cos \phi & 0 \\ 0 & 1 & l_1 \sin \phi & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

was mit $l_1 = 0$ gerade die rechtwinklige Projektionsmatrix darstellt.
Man unterscheidet entsprechend der Winkel im CAD Bereich:

∗ *Kavalier*
Die senkrecht zur Projektionsebene verlaufenden Objektkanten werden unverkürzt dargestellt, d.h.

$$\tan \alpha = 1 \quad \Longrightarrow \quad \alpha = 45°$$

und "ubliche Wahlen von ϕ sind:

$$\phi = 30° \quad \text{oder} \quad \phi = 45°$$

∗ *Kabinett*
Die senkrecht zur Projektionsebene verlaufenden Objektkanten werden um den Faktor 2 verkürzt dargestellt, d.h.

$$\tan \alpha = 2 \quad \Longrightarrow \quad \alpha = 63.43°$$

und "ubliche Wahlen von ϕ sind ebenfalls:

$$\phi = 30° \quad \text{oder} \quad \phi = 45°$$

- *perspektivische Projektion*
  Betrachten wir folgenden etwas vereinfachten Fall, d.h. der Fokuspunkt der Projektion liegt auf der *n*-Achse des Kamera-Koordinatensystems:

16.6 Kanonische Sichtvolumen

für eine Implementierung ist es zweckmäßig den sichtbaren Bereich in ein kanonisches Sichtvolumen einzuschließen, d.h.

- der Kamerapunkt (bzw. die Kamerarichtung bei paralleler Projektion) bestimmt mit den Rändern des Ausschnitts der Bildebene, der auf dem Ausgabegerät dargestellt werden soll, die seitlichen Begrenzungsebenen des Sichtvolumens;

- man spezifiziert weiterhin eine vordere und eine hintere Begrenzungsebene, um das Sichtvolumen vollständig abzuschließen; hierzu verwendet man entweder

– die durch die Szene vorgegebenen, maximalen Koordinaten bezüglich des Abstandes von der Bildebene,
– oder die durch den Benutzer angegebenen, maximalen Koordinaten des Teils der Szene, welcher sichtbar sein soll (so will man z.B. häufig, dass die Objekte zwischen Kamerapunkt und Bildebene nicht auf die Bildebene projiziert werden).

- das endliche Sichtvolumen wird durch lineare Transformation in ein kanonisches Sichtvolumen (z.B. Einheitswürfel, oder Einheitspyramide mit Grundfläche 2 auf 2 und Höhe 1) transformiert, so dass anschließende Clipping- und Sichtbarkeitsberechnungen einfacher zu berechnen sind.

Berechnung eines kanonischen Sichtvolumens für die Parallelprojektion

$$K_{par} = S \cdot T \cdot V_{par} \cdot R_{uvn} \cdot T_c$$

wobei

$$T_c = \begin{pmatrix} 1 & 0 & 0 & -c_x \\ 0 & 1 & 0 & -c_y \\ 0 & 0 & 1 & -c_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

den Ursprung in den Kamerapunkt verschiebt,

$$R_{uvn} = \begin{pmatrix} u_0 & v_1 & v_2 & 0 \\ v_0 & u_1 & u_2 & 0 \\ n_0 & n_1 & n_2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

die Weltkoordinaten in Kamerakoordinaten transformiert,

$$V_{par} = \begin{pmatrix} 1 & 0 & -p_x/p_z & 0 \\ 0 & 1 & -p_y/p_z & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

durch eine Scherung die schiefwinklige Projektion in eine orthogonale umwandelt,

$$T = \begin{pmatrix} 1 & 0 & 0 & -xw_{min} \\ 0 & 1 & 0 & -yw_{min} \\ 0 & 0 & 1 & -F \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

den Ursprung in eine Ecke des Quaders verschiebt,

$$S = \begin{pmatrix} 1/(xw_{max} - xw_{min}) & 0 & 0 & 0 \\ 0 & 1/(yw_{max} - yw_{min}) & 0 & 0 \\ 0 & 0 & 1/(B-F) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

den Quader zum Einheitsw"urfel skaliert.

Berechnung eines kanonischen Sichtvolumens für die perspektivische Projektion

$$K_{per} = S_2 \cdot S_1 \cdot V_{pre} \cdot T_r \cdot R_{uvn} \cdot T_c$$

wobei

$$T_r = \begin{pmatrix} 1 & 0 & 0 & -r_x \\ 0 & 1 & 0 & -r_y \\ 0 & 0 & 1 & -r_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

den Ursprung in den Fokuspunkt verschiebt,

$$V_{pre} = \begin{pmatrix} 1 & 0 & a & 0 \\ 0 & 1 & b & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

mit

$$\begin{aligned} a &= -(c_x + (xw_{min} + xw_{max})/2)/c_z) \\ b &= -(c_y + (yw_{min} + yw_{max})/2)/c_z) \end{aligned}$$

durch eine Scherung die Mittellinie des Sichtvolumens in die z-Achse (n-Achse) transformiert,

$$S_1 = \begin{pmatrix} 2c_z/(xw_{max} - xw_{min}) & 0 & 0 & 0 \\ 0 & 2c_z/(yw_{max} - yw_{min}) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

durch eine xy-Skalierung die seitlichen Begrenzungsebenen zu $45°$-Ebenen neigt,

$$S_2 = \begin{pmatrix} 1/(c_z + B) & 0 & 0 & 0 \\ 0 & 1/(c_z + B) & 0 & 0 \\ 0 & 0 & 1/(c_z + B) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

die Pyramide zur Einheitspyramide skaliert.

17 Sichtbarkeitsberechnungen

17.1 Elimination der Rückseiten

'backface culling'

Rückseiten können nur von geschlossenen Polyedern entfernt werden!

$$\langle \vec{n}, \vec{n} \rangle_O \leq 0$$

wobei $\vec{n}$ Blickvektor des Kamerakoordinatensystems und $\vec{n}_O$ Normalenvektor der Fassette.

Für konvexe Polyeder gilt weiterhin: ist eine Fassette sichtbar, so ist sie vollständig sichtbar.

Bei kanonischem Sichtvolumen genügt die z-Komponente des Normalenvektors.

17.2 Elimination nicht sichtbarer Kanten

17.3 Elimination nicht sichtbarer Flächen

17.4 *Z*-Puffer-Methode

Tiefenpuffer-Methode ('z-buffer- or depth-buffer-method')

Idee: Halte für jeden Bildschirmpunkt zusätzlich einen z-Wert, der den bzgl. der Kamera (Bildebene) berechneten Abstand kodiert. Ein Pixel wird nur dann aktualisiert, wenn das zu zeichnende Objekt für dieses Pixel einen Abstand näher zur Kamera aufweist.

Ist das Sichtvolumen normiert, d.h. Bildebene hat z-Koordinate gleich 0 und die hintere Begrenzungslinie hat z-Koordinate gleich 1, dann sieht der Algorithmus (für Polygone) wie folgt aus:

1. Initialisiere den Z-Puffer mit 0.
2. Zeichne die zu setzenden Pixel der projizierten Polygone in irgendeiner Reihenfolge, berechne dabei
   - den Abstand zur Bildebene und
   - überschreibe ein Pixel nur dann, wenn der Abstand des neu zu kleiner als der aktuelle Abstand für dieses Pixel im Z-Puffer ist

Der Z-Puffer Algorithmus funktioniert für jede Art der Projektion (insbesondere bei normiertem Sichtvolumen).

Zur Beschleunigung der Berechnung der z-Werte kann ein Scanline-Algorithmus (ähnlich wie beim Füllen von Polygonen vorgestellt) unter Zuhilfenahme einer differentielle Methode angewendet werden.

Betrachten wir den einfachen Fall einer rechtwinkligen Parallelprojektion, wobei das Sichtvolumen so normiert wurde, dass die x- und y-Koordinaten gerade Bildebenenkoordinaten entsprechen.

Ebenengleichung eines Polygons:

$$Ax + By + Cz + D = 0$$

Damit erhalten wir die z-Koordinate f"ur den Bildpunkt (x, y) als:

$$z = (-Ax - Bx - D)/C$$

und f"ur $(x + 1, y)$

$$\begin{aligned} z' &= (-A(x+1) - Bx - D)/C \\ & z - A/C \end{aligned}$$

A/C ist eine Konstante f"ur das ganze Polygon. Die Kanten des Polygon zeichnet man z.B. mit einem Bresenham-Verfahren. Der Algorithmus kann mit folgenden ¨Tricks¨ weiter verbessert werden:

- Man arbeitet generell mit einem Scanline-Verfahren.
- Man sortiert die Ecken der Polygone bzgl. ihrer y-Koordinaten der Projektion.
- Man bearbeitet nur die Polygone, die gerade von der aktuellen Scanline geschnitten werden, d.h. man hält sich eine Liste von aktiven und inaktiven Polygonen, die nach jeder Scanline aktualisiert wird.
- Man benötigt also nur einen Z-Puffer für eine Scanline (und nicht für das gesamte Bild).

Eigenschaften:

- leicht zu implementieren
- arbeitet für beliebige Objekte
  (lediglich kann es aufwendiger werden, die entsprechende z-Koordinate eines Pixels zu berechnen)
- hoher Speicherplatzverbrauch für den Z-Puffer
- dieser Speicher wird jedoch manchmal durch die Hardware der Grafikkarte unterstützt

Das Z-Puffer Verfahren kann erweitert werden, so dass auch transparente Objekte dargestellt werden.

17.5 *A*-Puffer

Idee: Man kodiert außer einem z-Wert eines Pixels auch eine Liste von Oberflächen, die an diesem Pixel sichtbar sind.

Da für normierte Sichtvolumen nur positive z-Werte vorkommen, kann man dies zweckmäßigerweise wie folgt kodieren:

Statt nur die Farbe eines Pixels zu halten, kann man auch wesentlich mehr Information für ein Pixel während der Berechnung halten, um dann später eine wesentlich bessere Darstellung berechnen zu können:

- Farbkomponenten (z.B. RGB-Wert)
- Transparenzwert
- tatsächlich überdeckte Fläche des Pixels
- Zeiger auf Objekt

17.6 Tiefensortier-Verfahren

Maler-Algorithmus ('painters-algorithm')

Idee: Sortiere die zu zeichnenden Polygone so, dass sie vollständig von ¨hinten nach vorne¨ gezeichnet werden können.

Problem: Wie bestimmt man die Zeichenreihenfolge?

Beschränken wir uns wieder zuerst auf die Darstellung von Polygonen (Fassetten von Polyedern).

Wir ben&uoml;tigen eine Vergleichsfunktion, die gegeben zwei Flächen entscheidet, welche von beiden zuerst gezeichnet werden soll.

Leider liefert uns eine solche Vergleichsfunktion keine totale Ordnung auf den Flächen, wie folgendes Beispiel zeigt:

Mit anderen Worten: die Operation ist nicht transitiv.

17.7 Unterteilungsverfahren

17.8 Zusammenfassung der Darstellungsarten

Wireframe Nur die Kanten der Polygone werden gezeichnet.

Flat-Shading Polygone werden in einer einheitlichen Farbe dargestellt. Dadurch können keine Kanten zwischen Flächen gleicher Farbe erkannt werden. (Wird bereits durch Hardware unterstützt.)

Gouraud-Shading (Wird bereits durch Hardware unterstützt.)

Phong-Shading (Wird bereits durch Hardware unterstützt.)

Ray Tracing ??

Radiosity

18 Ray Tracing

18.1 Prinzip

18.2 Strahlberechnung

18.3 Effekte

18.4 Beschleunigungsmethoden