Martin-Luther-Universität Halle-Wittenberg

Fachbereich Mathematik und Informatik

Skript zur Vorlesung

Computergrafik

gehalten von: Doz. Dr. P. Schenzel, WS 2002

Autor: Berit Haldemann

Inhaltsverzeichnis

Kapitel 1

Einleitung

- Modellierung, Beleuchtung, Dynamik von virtuellen Szenen und Interaktion von Nutzern
- Familien von Projektionen: 3D-Objekte → 2D-Displays
- Beleuchtung, Ray-Tracing
- Modellierung und VR (Virtual Reality) → eigenständiges Gebiet der Computergrafik
- Erfinden/Nachbilden von Objekten am Computer
- Realismus und Realzeit → widersprüchliche
  Tendenzen: Realismus erfodert Details
  Echtzeitdarstellung
  erfordert Datenreduktion

Realismus?

1. geometrischer Realismus (virtuelles Prototyping, z.B. Karosseriemodellierung)
2. Beleuchtungsrealismus
   Paradigma: extrem komplexe und rechenintensive Aufgabe
   optische Interaktionen: diffuses Licht, Schatten etc.
3. Verhaltungsrealismus
   Interaktion virtueller Objekte
4. verfremdete Darstellung
   Karikaturen, Trickfilme

Realzeit?

1. Bewegungsillusion
   Frame = Bild vom Computer erzeugt
   Frame = Anzahl verschiedener Bilder pro Sekunde (Hz) → Zusammenfassung individueller Frames zur optischen Vertauschung einer Kontinuität
2. Realzeitdurchgang
   üblich: 60 Frames pro Sekunde
   klassische Filme: 24 Frames pro Sekunde

3. Realzeit-Objektinteraktionen
   Rechenaufwand für Interaktionen
   Kollisionsbetrachtung

Kompromisse

- Wanderungen durch komplexe Szenen beschränken
- u.U. Beleuchtungsphänomene reduziert
- komplexe Szenen mit tausenden von Polygonen können nicht in Realzeit dargestellt werden
- verteilte VR-Umgebungen müssen Netzwerkverzögerungen berücksichtigen

Paradigma: Hardware-Entwicklungen verändern Kompromisslagen

Wie arbeitet VR-System?

Physiologie des Auges: Netzhaut mit photosensitiven Rezeptoren (Stäbchen und Zapfen)

menschliches Sehen:

komplexes Phänomen, erfahrungsbasiert

Kontrolle und Wiederkontrolle der Umgebung

→ Herausfinden von Bestätigung und Widerspruch zu menschlichem Handeln (Überlebensstrategien)

These: „Wirklichkeit ist virtuell."

3D-Sehen:

- beruht auf Stereoeffekt durch die unterschiedliche Lage der Augen
- Perspektive: Größe ist umgekehrt proportional zur Entfernung

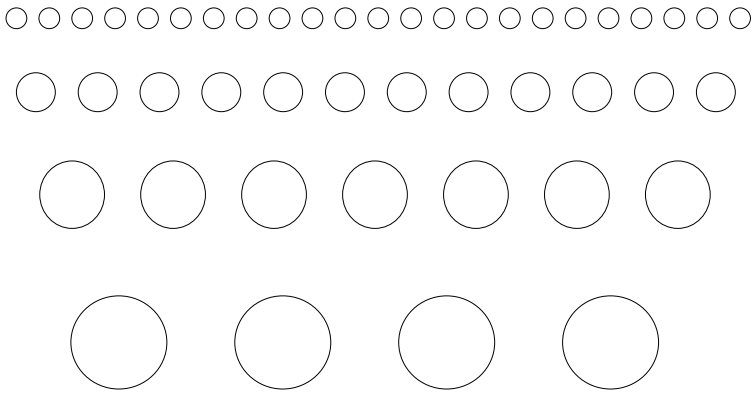
  → mathematische Hilfsmittel: Strahlensatz
  erfordert Tiefenanalyse

Skizze:

Vorwärtsskalierung simuliert Tiefeneindruck

Tiefenanalyse durch Texturgradient

Schatten und Schattierung unterstreichen räumliche Lagebeziehung



Raum- und Zeitkonsistenz:

- Augenbewegung erfolgt nicht stetig
- Objekte verändern sich in kleinen Umgebungen von Blickpunkten stetig, d.h. kleine Blickänderungen bewirken kleine Änderungen der Szene

Paradigma: Scanline, Scangeraden, Scanweg → pixelweises Zeichnen

Kapitel 2

Einführung in die Computergrafik

Ziele:

- Überblick
- Beschreibung von Eingabe- und Ausgabe-Devices
- Grafik-API

2.1 Was ist Computergrafik?

Problem: vielfältige Antworten, z.B.

- Generieren von Bildern und Steuern im Compiler
- wissenschaftliche Darstellung von Daten
- Spiele und Unterhaltungsszenarien

  ...

Tools: Hardwaretools, Softwaretools

Hardwaretools: Computer, Mouse, Inputdevices

Softwaretools: Betriebssysteme, Editoren, Compiler, Grafikroutinen

Ziel des Kurses:

Benutzen von Grafikbibliotheken

Device independent

OpenGL

Interaktivität!

Rechnerleistung und Speicherumfang:

Entwicklung der Computergrafik $\Leftarrow$ Entwicklung der Rechentechnik

Ausgabe:

Zeichnen von geometrischen Gebilden

Vertauschen von Datenmengen und wissenschaftlichen Zusammenhängen

Animationen

Eingabe:

Characters, Zahlen(Code)

Mousebewegungen

Pen

2.2 Wo benutzt man computergenerierte Bilder?

Zeichnen realer und irrealer Objekte

Schwerpunkte:

1. Kunst, Unterhaltung, Publishing, Filme, Werbung, Computerspiele, WWW, Buch-Design
2. Bildbearbeitung

   Computergrafik $\leftrightarrow$ Bildverarbeitung
   Mischen von Bildern
   Komprimieren
3. Prozessabläufe

   Modellierung
4. Darstellung von Simulationen

   z.B. reale Prozesse: Wettervorhersage
   Bewegungsabläufe: Flugsimulator
   Modellierung von realen Objekten
   Wolken, Wasser, Pflanzen
5. Computer-aided design

   virtuell Objekte konstruieren, die anschließend realisiert werden sollen
   Design:
   Realisierung eines Modells und Überprüfung der Funktionalität am Computer
   $\Rightarrow$ Kostenreduktion
6. wissenschaftliche Analyse und Visualismus

   Komplexität wissenschaftlicher Daten enorm $\Rightarrow$ grafische Darstellung zur Hervorhebung von Informationen
   makroskopische, mikroskopische Phänomene
   Molekülbau usw.

2.3 Computergenerierte Bildelemente

Problem: Wie zeichnet Computer?

Antwort: Komposition von Ausgangsprimitiven

- Polylinie
- Text
- ausgefüllte Regionen
- Rasterbilder

(teilweise Überlappung, aber gute Ausgangsthese)

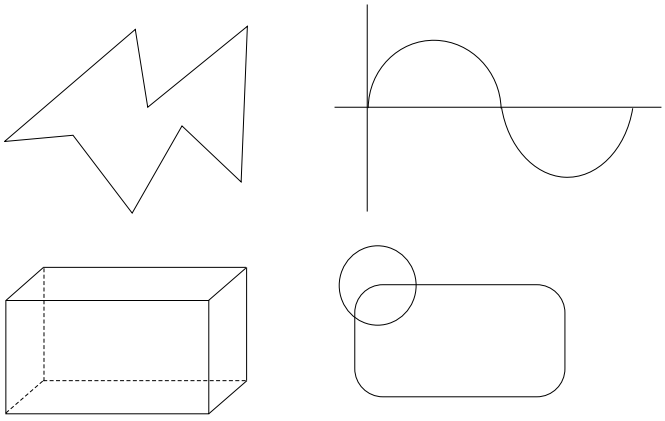
Attribute für Primitive:

Charakteristika wie Farbe, Dicke, Helligkeit u.a.

1. Polylinien

   = zusammengesetzte Folge von Strecken

   Skizze:



   „line drawings“: Bilder sind aus Polylinien gezeichnet

   Strecke: einfachste Polylinie

   z.B. Routine

   `drawLine(x1,y1,x2,y2)` zeichnet (x_1,y_1), (x_2,y_2)

   `drawDot(x1,y1)` zeichnet (x_1,y_1)

   Problem: Spezifikation des Datentyps von x_i, y_i: real, integer

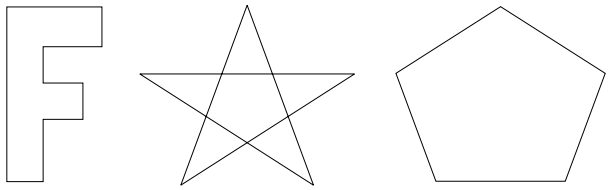
Kante: Strecke einer Polylinie

Ecke: Eckpunkt einer Polylinie

`drawPolyline(poly)` zeichnet Polylinie bestehend aus Punkten einer Liste poly, die Koordinaten (x_0, y_0)...(x_N, y_N) enthält

Polygon: = Polylinie, die geschlossen ist → (x_0, y_0)=(x_N, y_N)

z.B. Skizze:



1 2 3

einfaches Polygon: keine Überschneidungen der Strecken (bei 1, 3 in Abbildung **??**)

Attribute:

Dicke (=thickness) der Kanten

Ausführung der Kanten

2. Text

   Grafik-Devices unterscheiden text mode, graphics mode

   text mode: einfacher Eingabe-Ausgabemodus basierend auf einfachen Zeichensätzen (ASCII)

   graphics mode: eingebaute Erzeuger für grafische Symbole, z.B. grafisch wiedergegebener Text

   Befehl: `drawString(x,y,string);`

   (x,y) ist Anfangspunkt, dahin wird String gesetzt

   Text-Attribute: font, Farbe, Größe, Abstand, Orientierung

   Schriften: komplexe Arbeit zur Entwicklung ⇒ Kauf

3. Ausgefüllte Regionen

   = primitiv mit Mustern und Farbe ausfüllen

   Beispiel: Prozedur

`fillPolygon(poly,pattern)`

`poly` = Polylinie, `pattern` = Muster

Bemerkung: 3D

4. Rasterbild

   Computerbild = Menge kleiner Zellen

   pixel = picture element

   Rasterbild = gespeichtert als Array von numerischen Werten

   Array = a Zeilen x b Spalten

   numerische Werte = Werte des Pixels, das dort gespeichert wird = pixelmap, bitmap

   Wie werden Rasterbilder kreiert?

   a) handgezeichnet
   b) computergenerierte Bilder, Rendern einer Szene
   c) Sannen von Bildern

5. Grau- und Farbschattierungen

   Grauskala:

   - bilevel: 0,1: 0 = weiß, 1 = schwarz
     one-bit-per-Pixel-Bild
   - Pixel-Tiefe: Anzahl der Bits zur Repräsentation von Graustufen

     2 Bit/Pixel $\Rightarrow$ 4 Graustufen

     4 Bit/Pixel $\Rightarrow$ 16 Graustufen

     8 Bit/Pixel $\Rightarrow$ 256 Graustufen

     vernünftige Qualität

   Farbrasterbilder: Pixel eines Farbrasterbildes müssen Farbwerte zugeordnet werden

   Grundidee: Mischung von Rot, Grün, Blau

   Farbe = Tripel (23,14,51) $\Rightarrow$ Itensitäten von Rot, Grün, Blau

   Farbtiefe = Summe der Bitzahlen

   z.B. Farbtiefe

   3 Komponenten $\in \{0,1\}$

   (0,1,1) $\Rightarrow$ 0 = Rot, 1 = Blau, 1 = Grün

   Ergebnis = Cyan

Normal: Farbtiefe 8 Bit $\Rightarrow$ jeder Pixel besitzt 256 Farben

true color: Farbtiefe von 24 Bit = 1 Byte je Farbe

Bemerkung: höhere Farbtiefe lohnt nicht, da Auge keine Unterschiede mehr wahrnimmt

Bsp.: 24 Bit - Farbtiefe, 1080x1024 Auflösung erfordert $>$ 3 Mio. Bytes

2.4 Graphic Devices

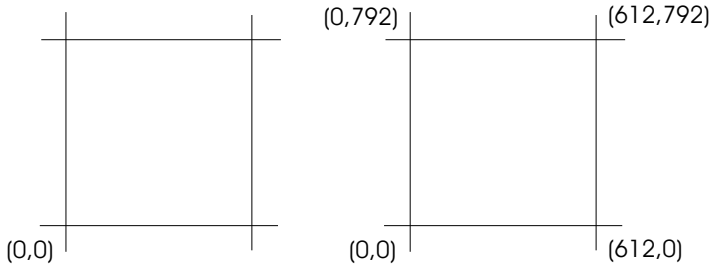
Postscript ps
device independent

Seitenbeschreibungssprache

Drucker haben einen internen Mikroprozessor, der Postscript interpretiert

Hinweise in Postscript:

Koordinatensystem



Einheit = $\frac{1}{72}$ Zoll $8\frac{1}{2}$ x 11 Zoll

Portraitmodus

Beginn eines Files: `%!`

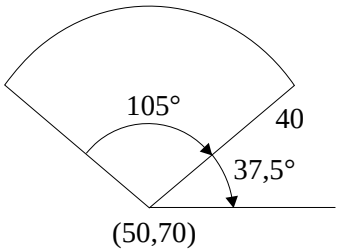
Beispiel:

1. Strecke

```
newpath 100 100 moveto  //Koordinaten des Punktes
200 200 lineto          //Linie von einem zum anderen Punkt
closepath               //Beenden des paths
stroke                  //"Tinte"
showpage                //"Druck"
```

2. Kreisbogen

```
50 70 40 37.5 105 arc
```



3. `translate, rotate, scale`

4. Def. Primitive

```
/box
{
    200 200 moveto
    300 200 lineto
    300 300 lineto
    200 300 lineto
    200 200 lineto
}
def
```

Zeichnen der Box:

```
newpath
box
stroke
80 80 translate
0.5 setgray
newpath
box
fill
showpage
```

5. weitere Bemerkung:

   - nicht ’case sensitive’

- stack-basiert
- arithmetische Operationen

  ```
  2 3 mult
  ```

- Funktionen: `cos` usw.
- Ausfüllen: `stroke, fill, setlinewidth, setgray`
- for-Schleifen

  ```
  1 -0.05 0 ... for //= Anfang Schrittweite Ende Prozedur
  ```

6. Schrift

```
/Helvetica
findfont
15 scalefont setfont
100 200 moveto
(Hallo, Welt!)
show
```

Ergänzungen zu Postscript:

- Ausfüllen

  ```
  stroke, fill, setlinewidth, setgray
  ```

- Transformationen

  ```
  translate, rotate, scale, gsave, grestore
  ```

  (kopieren, pop-up)

- Variable, Prozeduren,

  z.B.

  ```
  /drawDot
  {
      newpath
      2 copy moveto
      lineto
      stroke
  }
  def
  ```

  Anwendung: `x y drawDot`

  Boolsche Variable, Iterationen

2.5 Eingabeprimitive und Devices

Eingabe-Device:

- was ist es?
- was tut es?

Device:

- Bestandteil einer Maschine: Mouse, Keyboard, Trackball
- misst die Manipulation des Benutzers
- sendet numerische Informationen zurück zum Programm

Graphische Eingabeprimitive:

- Strings aus Charaktern
- Auswahl aus gegebener Menge
- Wert z.B. aus [0,1]
- Lokalisierung durch Koordinaten

physikalische Eingabedevices:

- Keyboard
- Buttons
- Mouse, Joystick, Trackball

2.6 Graphic displays

Verschiedene Tendenzen:

Entwicklung der Rechentechnik

1. Line Drawing

   Beispiel: Plotter, Trommelplotter

2. Video-Display

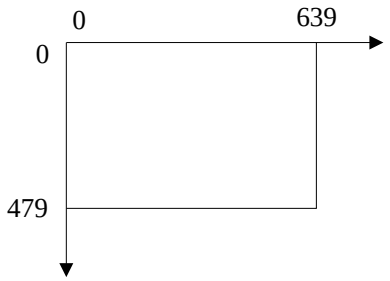
   - Vektordisplay

     Bsp.: Kathodenstrahlröhre (Oszillograph)
     sehr schnelles Zeichnen
     Nachteil: Schattierungen, Überlagerungen schwer bzw. nicht darstellbar

- Rasterdisplay

  Bildschirmfläche: Präsentationsfläche für Bilder

  z.B. 480x640 ≈ 307000 Pixel



Framebuffer: Speicher, der alle Pixelinfos des Displays enthält

Grafikkarte: verwaltet Speicher für Framebuffer

Arbeitsweise:

- Grafikprogramm im Speicher verarbeitet Befehle der CPU
- scan controller: verwaltet aktuelles Display pixelweise, sendet aktuellen Wert über Converter auf Displayfläche

Scan-Prozess:

- Pixelwert im Framebuffer wird zur korrekten Stelle des Displays geschickt
- Umwandlung der logischen Adresse in geometrische Position
- jedes Pixel ist genau einmal besucht, scan wird mehrere Male pro Sekunde wiederholt, Auffrischrate = refreshing rate

Monitor:

CRT = cathode ray tube

R,G,B - Komponenten der Farbwerte der Pixel

DAC = digital to analog converter

Konvertierung logischer Werte in aktuelle Spannungswerte

3 Einschübe in CRT:

3 Elektronenstrahlen ≈ proportional zur Spannung, stimulieren 3 dünne Phosphorschichten

4x4x4=64 verschiedenen Werte

60 mal/s = Auffrischungsrate (refreshing rate)

Bemerkung:

bessere Grafiksysteme unterstützen 24 Speicherebenen

jedes DAC hat 8 Eingabebits

$\Rightarrow$ 256 Grade für R,G,B jeweils

$\Rightarrow 2^{24} \approx$ 16 Mio. Farben

3. indizierte Farben und LUT

   LUT = look up table

   Alternative zu Pixelzuordnung mit Farben

   Beispiel: Farbtiefe 6

   64 Werte = Index in einer Tabelle

   z.B. 39 enthält 01010, 11001, 10010

   `setPalette(39,17,25,4)`

   Arbeitsweise:

   (x,y) = (479,532) soll mit Farbe 39 gezeichnet werden:

   `drawDot(479,532,39)`

   Problem: Berechnung der zu setzenden Farbwerte

   Vorteil: $2^{15} \approx$ 32k mögliche Farben, Farbpalette mit 32k Farben

4. Hard copy devices

   Drucker, Recorder

   Ausgabe bei Druckern: dot x dot

   600 dpi, 1200 dpi (dot per inch)

   Linotronic-Setzmaschinen 2540 dpi

   moderne Drucker: interner Mikropozessor interpretiert Postscript

Zusammenfassung:

- Begriffsbestimmung und Anwendungen
- grafische Wiedergabetechniken
  Rasterdisplays, Drucker
- grafische Ausgabeprimitive
  z.B. Postscript

Literaturempfehlungen: [1], [2], [3], [4], [5]

Kapitel 3

Zeichnen von Figuren

Ziele:

- Programm zum Zeichnen
- Grundsätzliches zu OpenGL
- Elementare Werkzeuge für Strecken, Polylinien und Polygone
- Interaktivitäten mit Mouse und Tastatur

3.1 Learning by doing

Benutzer:

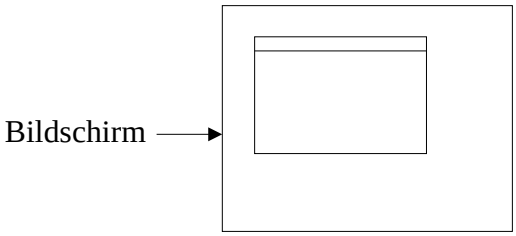
API

Hardware

Bibliotheken von Softwaretools

zwei Alternativen:

- ganzer Bildschirm zum Zeichnen
- moderner: window-basiertes Zeichnen



Koordinatensystem:

Ursprung links oben

+x-Achse nach rechts

+y-Achse nach unten

Tools zum Zeichnen: z.B.

`setPixel(x,y,Color)`

`drawPoint()`

`line(x1,y1,x2,y2)`

Forderung:

Device-Independent programmieren

OpenGL für Grafik

= API (application programming interface)

= Sammlung von Grafik-Routinen, insbesondere für 3D

windowgestütztes Programmieren ereignisgesteuert

call-back-Funktionen, falls Ereignisse vorkommen

OpenGL $\Leftarrow$ GLUT

Beispiel:

- `glutMouseFunc(myMouse)`
  registriert die Funktion `myMouse()` als Funktion, die ausgeführt wird, wenn ein Mouse-Ereignis erscheint
- Programmierer übergibt code in `myMouse()` für Aktionen

Skizze: Beispiel eines `main()`-ereignisgesteuertes Programm

Typen von Ereignissen:

1. `glutDisplayFunc(myDisplay)`

   falls Fenster neu gezeichnet werden soll, gibt die Funktion `redraw` als Ereignis zurück das geschieht, falls das Fenster geöffnet und durch ein anderes überdeckt wird
   hier ist `myDisplay()` als call-back für das Ereignis Neuzeichnen registriert

2. `glutReshapeFunc(myReshape)`

   Fenster können durch Benutzer verformt werden, durch „Ziehen“an einer Ecke
   Funktion `myReshape()` ist registriert für das Verform-Ereignis
   `myReshape()` übernimmt Argumente Breite und Länge des verformten Fensters

3. `glutMouseFunc(myMouse)`

falls eine Mouse-Taste gedrückt bzw. losgelassen wird, entsteht ein Mouse-Ereignis
`myMouse()`ist registriert als Aufruf, falls das Mouse-Ereignis eintritt
`myMouse()`übernimmt automatisch Argumente des Ortes und der Natur der Aktion

4. `glutKeyboardFunc(myKeyboard)`

   Funktion registriert `myKeyboard()`mit dem Drücken einer Taste auf der Tastatur
   `myKeyboard()`durchläuft automatisch Argumente, die lokalisieren welcher Knopf gedrückt wurde

Fenster öffnen:

OpenGL: keine systemspezifischen Kommandos für Fenster

GLUT: graphics library utility toolkit

mit Funktionen zum Fenster öffnen für das spezifische System

Skizze: erste fünf Funktionen rufen GLUT zum Fenster öffnen auf

Typen:

1. `glutInit(& argc,argv);`

   initialisiert GLUT
   Argumente signalisieren Übergabe für Argumente

2. `glutInitDisplayMode(GLUT_SINGLE/GLUT_RGB);`

   Funktion spezifiziert, wie das Fenster geöffnet werden soll

   GLUT_SINGLE: single buffer Display
   GLUT_RGB: RGB-Komponenten

3. `glutInitWindowSize(640,480)`

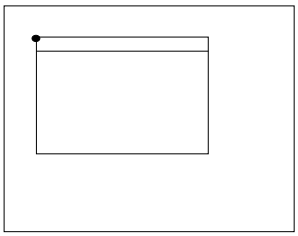
   initialisiert Fenstergröße
   640 breit, 480 hoch

4. `glutInitWindowPosition(100,150)`

   spezifiziert obere linke Ecke für Öffnen des Fensters
   bei laufendem Programm, kann Position geändert werden

5. `glutCreateWindow(„Titel“)`

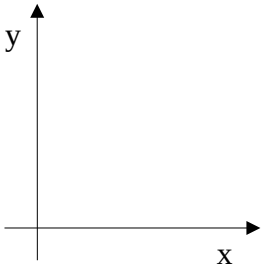
   öffnet Fenster mit vorgegebenem Titel



3.2 Zeichnen grafischer Primitive

Zeichenfunktion mit call-back-Funktion assoziiert, redraw-Ereignis wie myDisplay()

Festlegung eines Koordinatensystems



OpenGL liefert Bausteine für Grafikprimitive eingeschlossen in

```
glBegin()

glEnd()
```

z.B.:

```
glBegin(GL_POINTS);
    glVertex2i(100,50);
    glVertex2i(100,30);
    glVertex2i(150,130);
glEnd();
```

Hierzu:

`GL_POINTS, GL_LINES, GL_POLYGONS` usw.
integrierte Funktionen

`glVertex2i():`

gl = Bibliothek
Vertex = Basisbefehl
2 = Anzahl der Argumente (hier kann auch 3 oder 4 stehen)
i = Typ, hier: Integer

Unterschied: Präfix glut

Beispiel:

```
void drawDot(int x, int y)
          (GLint x, GLint y)
{ //Zeichne Punkt mit ganzen Koordinaten (x,y)
    glBegin(GL_POINTS);
        glVertex2i(x,y);
    glEnd();
}
```

Bemerkungen:

1. `glPointSize()`

   Größe der Punkte
   floating points

2. `glColor3f(red,green,blue)`

   Argumente $\in$ [0,1]
   (1.0,1.0,1.0) = weiß
   (1.0,1.0,0.0) = gelb

   Hintergrundfarbe

   `glClearColor(red,green,blue,alpha)`

   alpha = Transparenz

   `glClear(GL_COLOR_BUFFER_BIT)`

   initialisiert Fenster mit Hintergrundfarbe

3. Initialisierung von Koordinatensystemen

```
void myInit(void)
{
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();               //Laden der Einheitsmatrix
    gluOrtho2D(0,640.0,0,480.0);    //Typ der Produktion
}
```

Transformation für 640x480 Pixel-Fenster $\Rightarrow$ setzt view-Region

4. idle callback
   idle = untätig

   `void glutIdleFunc(myIdle)`

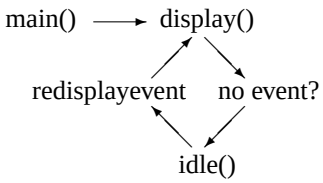
   erlaubt Hintergrundprozesse

   Aktivierung: kontinuierlicher Aufruf, falls keine Ereignisse aufgerufen werden

   aktuelles Fenster und aktuelles Menü werden vor idle callback nicht geändert

   Aufruf endet häufig mit `glutPostRedisplay()`

   Arbeitsweise:



Beispiel:

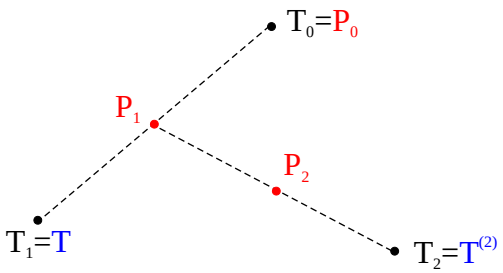
```
void myIdle()
{
    theta+=2.0;
    if(theta>=360.0)
        theta-=360.0;
    glutPostRedisplay();
}
```

Hinweis:

- direkter Aufruf von displaycallback zeichnet mehrfach
- `glutPostRedisplay()`: garantiert, dass Fenster höchstens einmal gezeichnet wird, falls die Ereignisschleife durchlaufen wird

Beispiel: **Sierpinsky-Dreieck**

Skizze:



1. Fixieren Punkte T_0, T_1, T_2
2. Wähle Punkt P_0 zufällig aus $\{T_0, T_1, T_2\}$
3. Wähle Punkt T zufällig aus $\{T_0, T_1, T_2\}$
4. Konstruiere Punkt P_k als Mittelpunkt der Strecke TP_{k-1}
5. Zeichne P_k
6. Iteriere 3. bis 5.

Hinweise:

- Klasse für Punkte definieren: `GLintPoint`

```
class GLintPoint
{
    public:
        GLint x,y;
}
```

- Speicherreduzierung (nicht nötig P_k zu speichern)
- Zufallszahlen

```
int random(int m)
{
    return rand()%m //modulo m
}
```

Problem: `glOrtho2D`

Zeichnen funktionaler Zusammenhänge

z.B. $f(x) = e^{-x}cos(2\pi x)$, zeichne $(x_i, f(x_i))$

Problem: Fenster und Punkte „passen“nicht zusammen

Lösung: Skalierung, für x-Achse, y-Achse

innere Schleife in `myDisplay`

```
glBegin(GL_LINE_STRIP);
for(GLdouble x=0,x<4.0,x+=0.005)
{
    glVertex2d(A*x+B,C*func+D);
}
glEnd();
glFlush();
```

A,B,C,D Skalierungs- bzw. Verschiebungsparameter

3.3 Linien zeichnen

OpenGL - Möglichkeiten

```
GL_LINES
```

z.B.:

```
glBegin(GL_LINES);
    glVertex2i(40,100);
    glVertex2i(202,96);
glEnd();
```

liefert Strecke

Attribute

```
glColor3f();
glLineWidth();
```

Alternative:

```
GL_LINE_STRIP
GL_LINE_LOOP //fuer geschlossene Linien
```

Hinweis zur Übung:

Punkte aus einem Datenfile einlesen

Struktur von polyline.dat:

Struktur	**polyline.dat**
21	Anzahl der Polylinien
4	Anzahl Punkte der ersten Polylinie
169 118	Koordinaten des ersten Punktes
172 120	Koordinaten des zweiten Punktes
...	...
5	Anzahl Punkte der zweiten Polylinie

usw.

Anwendungen:

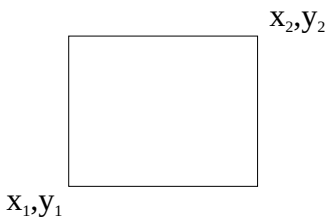
- Parametrisierung von Figuren: Spiegeln, Drehungen, Verschieben, Skalieren
- Muster generieren

weitere linienbasierte Primitive:

- Rechteck

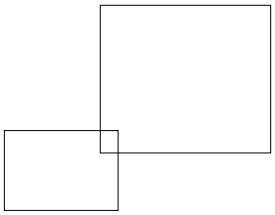
```
glRecti(GLint x1,GLint y1,Glint x2,GLint y2)
```

zeichnet achsenparallele Rechtecke



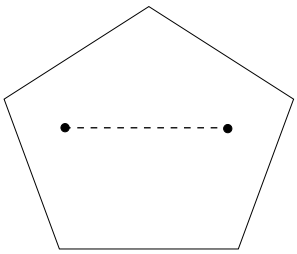
z.B.:

```
glClearColor(1.0,1.0,1.0,0.0);
glClear(GL_COLOR_BUFFER_BIT);
glColor3f(0.6,0.6,0.6);
glRecti(20,20,100,70);
glColor3f(0.2,0.2,0.2);
glRecti(70,50,150,130);
glFlush();
```

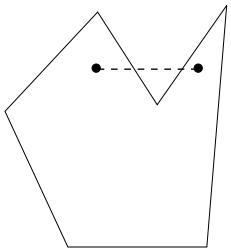


- Polygone

  konvexes Polygon (Verbindungsstrecke zwischen 2 Punkten liegt wieder innerhalb des Polygons)



konvex



nicht konvex: konkav

häufig: beschränken auf konvexe Polygone

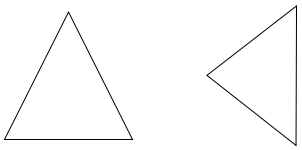
```
glBegin(GL_POLYGON);
    glVertex2f(x0,y0);
    ...
    glVertex2f(xn,yn);
glEnd();
```

zeichnet konvexes Polygon mit Eckpunkten $(x_0,y_0),...,(x_n,y_n)$

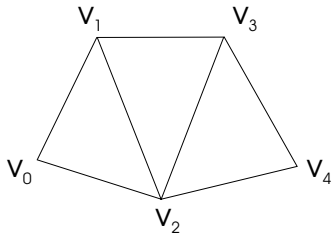
geschlossen $(x_0,y_0) = (x_n,y_n)$

- weitere Grafikprimitive:

  `GL_TRIANGLES` (zeichnet Dreiecke aus 3 Punkten)



  `GL_TRIANGLE_STRIP` (zeichnet Dreiecke mit Ecken v_0,v_1,v_2 / v_1,v_2,v_3 / v_2,v_3,v_4)



  `GL_QUADS` (zeichnet Rechtecke)

  `GL_QUAD_STRIP`

3.4 Interaktionen mit Mouse und Keyboard

Ziel: benutzerdefinierte Kontrolle von Grafik-Anwendungen

Möglichkeiten von OpenGL: über GLUT
Registrierung von call-back-Funktionen

Möglichkeiten:

- `glutMouseFunc(myMouse)`

registriert `myMouse()` mit Ereignis, das erscheint, falls Mouseknopf gedrückt bzw. losgelassen wird

- `glutMotionFunc(myMovedMouse)`

  registriert `myMovedMouse()` mit Ereignis, das erscheint, falls die Mouse bewegt wird bei gedrückter Mousetaste

- `glutKeyboardFunc(myKeyboard)`

  registriert `myKeyboard()` mit Ereignis, falls Taste gedrückt wird

Mouseinteraktion

`myMouse()`

```
void myMouse(int button, int state, int x, int y);
```

mit

- button: `GLUT_LEFT_DOWN`, usw.
- state: `GLUT_UP, GLUT_DOWN`
- x,y: Koordinaten der Pixel
  x = Pixelzahl von links
  y = Pixelzahl von oben

Beispiel:

Druck linke Mousetaste ⇒ Punkt zeichnen
Druck rechte Mousetaste ⇒ Beenden des Programms

```
void myMouse(int button,int state,int x,int y)
{
    if(button==GLUT_LEFT_BUTTON && state==GLUT_DOWN)
    {
        drawDot(x,screenHeight-y);
    }
    else if(button==GLUT_RIGHT_BUTTON && state==GLUT_DOWN)
    {
        exit(-1);
    }
}
```

Bemerkung: `exit(-1)` gibt -1 zurück, was generiert wird

Beispiel:

1. Zeichnen von Rechteck

übergibt Eckpunkte mit Mouse

zeichnet achsenparalleles Rechteck

2. Zeichnen von Polylinien mit der Mouse

Ersetzen der Punkte wird vermieden

speichert Punkte im Array

Füllen des Arrays = Ende des Prozesses

Mouseklick: Fenster gesäubert und neu gezeichnet

3. Mousebewegung

`glutMotionFunc(myMovedMouse)`

`void myMouse(int x,int y)`

Freihandzeichnen

```
void myMovedMouse(int mousex,int mousey)
{
    GLint x=mousex;
    GLint y=screenHeight-mousey;
    GLint brushsize=20;
    glRecti(x,y,x+brushsize,y+brushsize);
    glFlush();
}
```

4. Keyboard-Interaktionen

call-back-Fkt. `myKeyboard();`

registriert über `glutKeyboardFunc(myKeyboard);`

`void myKeyboard(unsigned int key,int x,int y);`

Wert von key: ASCII-Wert

x,y Mouse-Position (y von unten gezählt!!)

Beispiel:

Drücke p: zeichne Punkt von Mouse-Position

Drücke „←“: füge Punkt zu Liste hinzu

Drücke E: Programmende

Hinweis:

Falls „p“gedrückt gehalten und Mouse bewegt wird, entsteht Folge von Punkten

$\Rightarrow$ Freihandzeichnen

5. Fenstermanagement

GLUT unterstützt mehrfache Fenster und Unterfenster

z.B. zweites Hauptfenster

```
id=glutCreateWindow(„zweites Fenster“);
glutSetWindow(id);
```

`glutInitDisplayMode` vor `glutCreateWindow`
(eigene call-back-Funktion)

6. Menüs

GLUT erlaubt „pop up“-Menüs

Definition von Einträgen

Verknüpfung der Menü-Einträge mit Mouse-Tasten bzw. Keyboard-Tasten

z.B. callback `demo_menu`

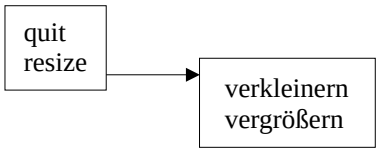
```
glutCreateMenu(demo_menu);
glutAddMenuEntry(„quit“,1);
glutAddmenuEntry(„vergrößern“,2);
glutAddMenuEntry(„verkleinern“,3);
glutAttachMenu(GLUT_RIGHT_BUTTON);
```

Aufruf

```
void demo_menu(int id)
{
    if(id==1) exit();
    else if(id==2) size=2*size;
    else if(id==3) size=size/2;
    glutPostRedisplay();
}
```

Hinweis:

GLUT unterstützt auch hierarchische Menüs



```
sub_menu=glutCreateMenu
glutAddMenu(„resize“,sub_menu);
```

Zusammenfassung:

- erste Schritte in OpenGL als API
  OpenGL nützliches Hilfsmittel
  unabhängig von Hardwarevoraussetzungen
- windowbasierte Arbeitsumgebung mit Interaktionen mit Interaktion von Mouse und Keyboard
- primitive Grafikroutinen zum Zeichnen von Punkten, Linien, Streckenzügen usw.
- einfache Beispiele

  Übung $\Rightarrow$ Zusammenstellen von Routinen für Grafik-Toolbox

Literaturempfehlungen: [6], [7], [8], [9]

Kapitel 4

Werkzeuge zum Zeichnen

Ziele:

- Einführung des viewports
- Transformation von Fenster zum viewport
- Zeichenumgebung für Weltkoordinaten
- komplexe Figuren zeichnen
- Weltkoordinaten und Weltfenster
  Vereinfachung von Grafikanwendungen
  aspect ratio
- einfache 3D-Applikationen

4.1 Einleitung

Bildschirmgröße: [0,screenWidth-1]x[0,screenHeight-1]

Darstellung von Funktionen: z.B. Definitionsbereich (DB) [-1,1], Wertebereich (WB) [100,20]

Ziel: Zusammenhang herstellen

Hilfsmittel: Skalierungen, Drehungen, Clippen usw.

z.B. $sinc(x) = \frac{sin\pi x}{\pi x}$

DB = $[-\infty, \infty]$, WB = [-1,1]

```
void myDisplay(void)
{
    glBegin(GL_LINE_STRIP);
    for(GLfloat x=-4.0;x<4.0;x+=0.1)
    {
        GLfloat y=sin(3.14*x)/3.14*x;
        glVertex2f(x,y);
```

```
        }
        glEnd();
        glFlush();
    }
```

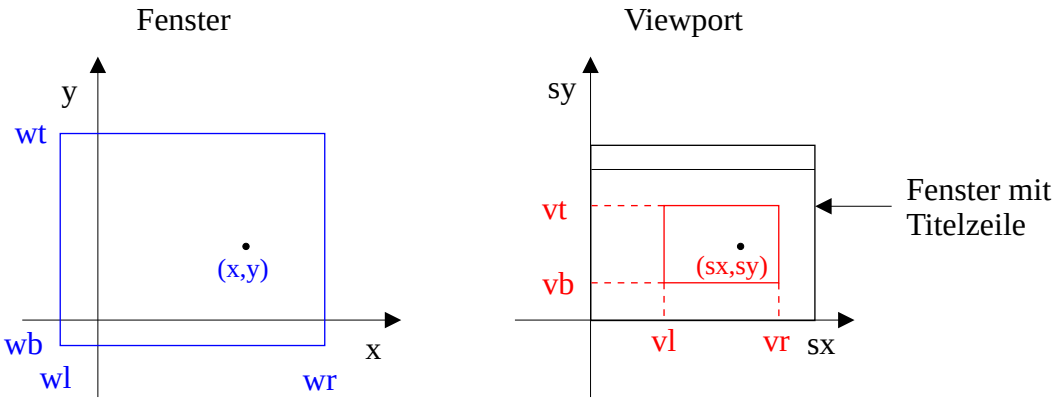
Bemerkung: Operationen erfolgen in natürlichen Koordinaten

Problem: Wie müssen (x,y) skaliert und verschoben werden, damit Bild im Bildschirmfenster erscheint?

Lösung:

- Festlegung eines Weltfensters
- Festlegung des viewports
- Berechnung der Abbildung

Abbildung vom Weltfenster zum viewport:



Bemerkungen:

- Weltfenster kann von beliebiger Größe und Position sein
- viewport sollte innerhalb des Bildschirmfensters liegen
- beide Fenster haben im Allgemeinen nicht gleiche Verhältnisse

Window-viewport-Abbildung:

Ansatz:

$$sx = Ax + C$$
$$sy = By + D$$

Hierbei: A,B - Skalierungen, C,D - Verschiebungen

Wie bestimmt man A,B,C und D?

Lösung: Ähnlichkeitsabbildung

$$\frac{sx-vl}{vr-vl} = \frac{x-wl}{wr-wl}$$

Auflösen:

$$sx = \frac{vr-vl}{wr-wl}x + vl - \frac{vr-vl}{wr-wl}wl$$

entsprechend für sy

Ergebnis:

$$sx = Ax + C$$
$$sy = By + D$$

mit

$$A = \frac{vr-vl}{wr-wl} \qquad C = vl - A * wl$$
$$B = \frac{vt-vb}{wt-wb} \qquad D = vb - B * wb$$

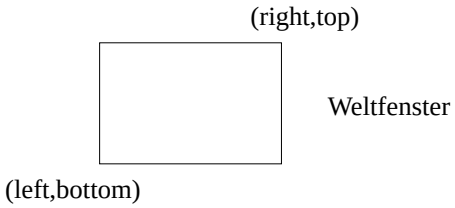
4.2 Eigenschaften von Ähnlichkeitsabbildungen

OpenGL
Finde Werte A,B,C,D

→ einfache Realisierung:

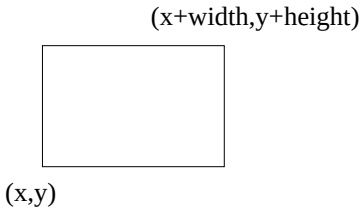
- `gluOrtho2D() //zeichnet Weltfenster`

```
void glutOrtho2D(GLdouble left,GLdouble right,GLdouble bottom,GLdouble top);
//setzt Fenster mit (left,bottom)(right,top)
```



- `glViewport() //zeichnet Viewport`

```
void glViewport(GLint x,GLint y,GLint width,GLint height);
//setzt viewport auf (x,y)(x+width,y+height)
```



→ Programmablauf:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity;
    //Laden der Einheitsmatrix um Matrix zu initialisieren
glOrtho2D(0.0,2.0,0.0,0.1);
glViewport(40,60,360,240);
```

Ergebnis:

jeder Punkt (x,y) wird in OpenGL bei Benutzung z.B. von `glVertex2*(x,y)` der Transformation $sx = Ax + C$, $sy = By + D$ unterzogen

→ Vergleich:

1. bisherige Verfahren:

   in der `main()`:

   ```
   glutInitWindowSize(640,480) //Bildschirmgroesse gesetzt
   ```

2. neu:

   in Methode `myInit()`:

   ```
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   gluOrtho2D(0.0,640.0,0.0,480.0)
   ```

Anwendungen:

1. Beispiel (siehe Kopie)

   → Transformation, sodass Sinusfunktion gesehen wird

2. Clippen: OpenGL clippt automatisch (siehe Kopie)

   Zoomen dadurch, dass neues Fenster festgelegt
   → clippen = das, was außerhalb des neuen Fensters, wird rausgeschnitten

   Annahme: `6eck()` zeichnet Sechsecke

   → verschiedene Ausschnitte zeichnen

```
setWindow(...);     //Fenster fuer Ansicht
setViewport(...);   //setzt viewport fuer jede Ansicht
6eck();             //neu Aufrufen derselben Funktion
```

3. Zoomen: (siehe Kopie)

   Bsp.:

```
float cx=0.3, cy=0.2; //Zentrum des Fensters
float H,W=1.2;        //Hoehe und Breite
float aspect=0.7;     //Verhaeltnis von Hoehe und Breite

//setze viewport
for(int frame=0;frame<NumFrames;frame++)
{
    clear screen                    //Loeschen
    W*=0.7                           //reduziert Fensterbreite
    H=W*aspect;                     //Verhaeltnis der Seiten erhalten
    setWindow(cx-W,cx+W,cy-H,cy+H); //setzt neues Fenster
    6eck();                         //neu Aufrufen derselben Funktion
}
```

4. Hinweise:

   Probleme: Zyklus folgender Operationen

   a) Löschen gezeichneter Figur
   b) (langsames) Zeichnen der neuen Figur
      (kann zum Flackern führen)

   besser:

   a) stehendes Bild der gezeichneten Figur
   b) unmitelbares Ersetzen der fertig neu gezeichneten Figur

   → OpenGL-Lösung: double buffering:

   `glutSwapsBuffers();`
   Reservierung eines separaten Buffers mittels:
   `GLUT_DOUBLE` anstelle von `GLUT_SINGLE` in `main`
   im Programm:

`glutInitDisplayMode(GLUT_DOUBLE/GLUT_RGB)`
`glutSwapBuffers()`in Codefragment nach `6eck()`

Automatisches Setzen von Window und viewport:

Problem: Programmierer weiß im Allgemeinen nicht, wo das interessante Objekt liegt (→ Dino in Übung)

Lösungsansätze: suche Bounding-Box = achsenparalleles Rechteck, das das Rechteck umhüllt

→ Berechnung einer Bounding-Box

1. falls Endpunkte der Objektlinien in ein Array p[i] gespeichert sind, dann bestimme

   max p[i].x, min p[i].x

   min p[i].y, min p[i].y

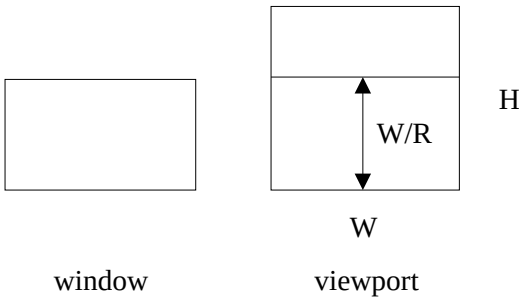
2. prozedural definierte Objekte (Funktionen):
   zwei Durchläufe zu zeichnen
   → für ersten Durchlauf wende 1. an
   → 2. Durchlauf zeichne/ausgeben mit der Bounding-Box aus erstem Durchlauf

„aspect ratio“

→ Beibehalten der Seitenverhältnisse vom Window und viewport (um Verzerrungen zu vermeiden)

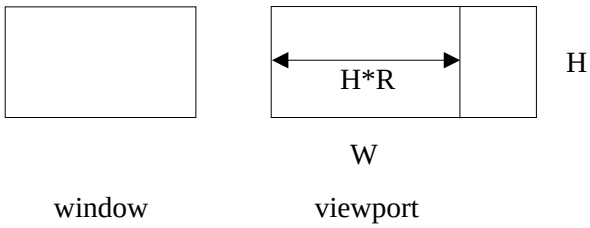
R=aspect ratio : Verhältnis von Breite und Höhe des Window
W,H : Breite und Höhe des viewport

1. Fall: $R > \frac{W}{H}$



→ `setViewport(0,W,0,W/R);`

2. Fall: $R \leq \frac{W}{H}$



→ `setViewport(0,H*R,0,H);`

→ OpenGL unterstützt Neuzeichnen des Fensters in Realzeit

```
glutReshapeFunc();
glutReshapeFunc(myReshape);          //Aufruf der Routine
void myReshape(GLsizei W,GLsizei H)
```

4.3 Clippen

Clippen = fundamentale Operation

→ Weglassen von Objekten, die außerhalb eines Bereiches liegen, der gezeichnet werden soll

OpenGL: Objekt wird automatisch im Weltfenster geklippt

Clipp-Algorithmus von OpenGL

Hier: Algorithmus, der Strecken außerhalb eines achsenparallelen Rechtecks clippt

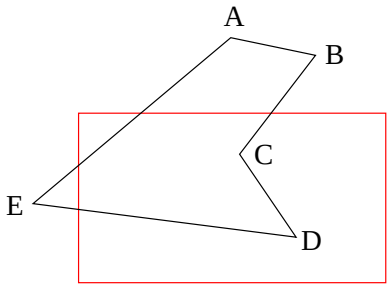
Ziel: Entwicklung einer Routine

`clipSegment(p1,p2,window)`

Hierbei: p1,p2: Endpunkte einer Strecke, Window: achsenparalleles Rechteck

Rückgabe 1: falls mindestens 1 Punkt der Strecke sichtbar

Rückgabe 0: falls kein Punkt der Strecke im Fenster sichtbar



Strecke	Rückgabe
CD	1
AB	0

Problem: viele Lagemöglichkeiten der Strecke relativ zum Rechteck

Anspruch: effektive Organisation, um unnötige Schnittpunktberechnungen zu vermeiden

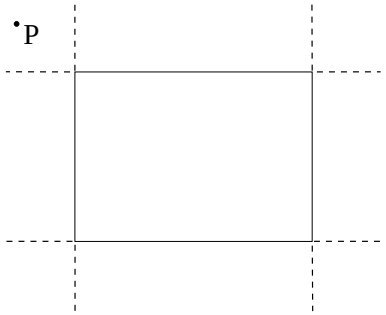
Algorithmus von Cohen-Sutherland

Idee: formalisieren triviales Akzeptieren und triviales Verwerfen einer Strecke, übrige Strecke sinnvoll unterteilen

(x) die sowieso in Fenster liegen

(y) die insgesamt außerhalb liegen

- Test für triviales Akzeptieren und Verwerfen: Codierung eines Punktes bzgl. des Rechteckes



Codierung von P: TTFF

P ist links vom Window: T(=True)

P ist oberhalb vom Window: T

P ist rechts vom Window: F(=False)

P ist unterhalb vom Window: F

→ schauen durch Begrenzungspunkte des Rechtecks, aber Vergleich der Koordinaten

Ergebnis: Strecke mit Endpunkten P,Q

→ triviales Akzeptieren: P,Q haben Code FFFF

→ triviales Verwerfen: P,Q haben T an selber Stelle

→ Divide-and-Conquer-Strategie: Abschneiden der Strecke und wiederholen

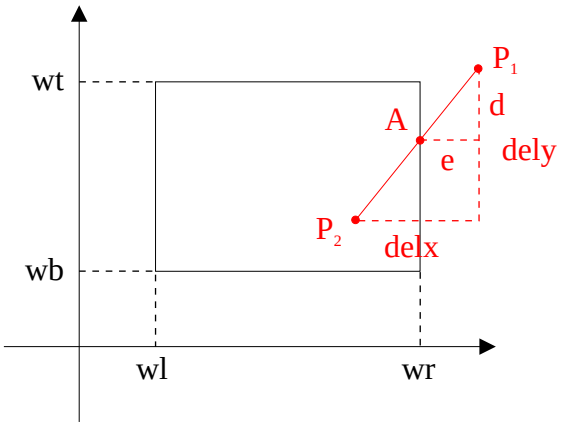
```
- do
  {
      bilde das Codewort von p1 und p2

      if(triviales Akzeptieren)
          return 1;

      if(triviales Verwerfen)
          return 0;

      fuer alle uebrige zerlege Strecke an der naechsten Fensterseite
      und verwerfe den aeusseren Teil
  }
  while(1);
```

- zerlegen und verwerfen



$$delx = P2.x - P1.x$$
$$dely = P2.y - P1.y$$

nach Strahlensatz:

$$\frac{d}{dely} = \frac{e}{delx}$$

Koordinate von A:

$$d = e * \frac{dely}{delx}$$
$$e = P1.x - wr$$

Ergebnis:

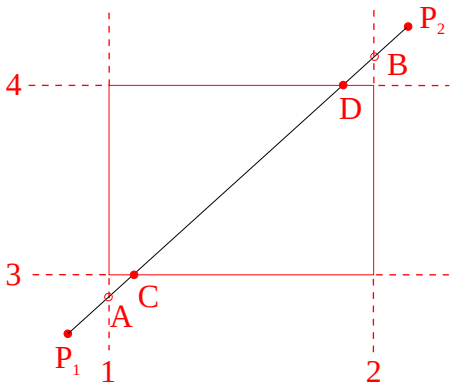
$$A = \left(wr, P1.y + (wr - P1.x) * \frac{dely}{delx} \right)$$

Bemerkungen:

- andere Berechnungen (für Schnitte) sind ähnlich
  → vertausche dely mit delx
- was passiert, falls dely=0 bzw. delx=0?
  → selbst überlegen

Bemerkung:

- in jeder do-Schleife wird der Code für Endpunkt der Strecke berechnet
- falls triviales Akzeptieren bzw. triviales Verwerfen falsch ist, testet der Algorihmus, ob P1 außerhalb ist
- falls P1 außerhalb, wird Schnittpunkt berechnet, äußeres Stück der Strecke weggelassen
- falls P1 innerhalb liegt, muß P2 außerhalb liegen, dann wird P2 gegen den Rand geclippt



Alternative Clipptechniken und Ergänzungen:

Clipp-Algoritmus von Liang-Barsky $P_i=(x_i,y_i)$, i=1,2 zwei verschiedene Punkte der Ebene

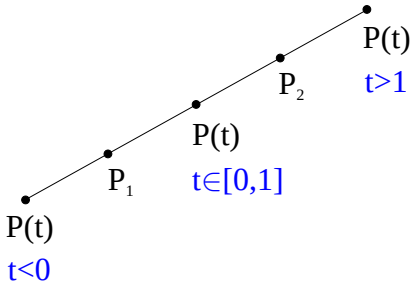
Parameterdarstellung der Strecke durch P_1,P_2

$$g : P(t) = P_1 + t(P_2 - P_1), t \in \mathbb{R}$$

Darstellung der Komponenten:

$$x(t) = x_1 + t(x_2 - x_1)$$
$$y(t) = y_1 + t(y_2 - y_1)$$

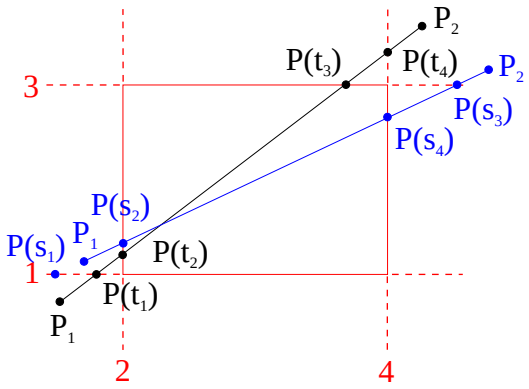
Skizze:



Hinweis:

- t∈[0,1]: P(t) liegt auf der Strecke $\overline{P_1P_2}$
- t<0: P(t) liegt vor P_1
- t>1: P(t) liegt nach P_2

Idee von Liang-Barsky:



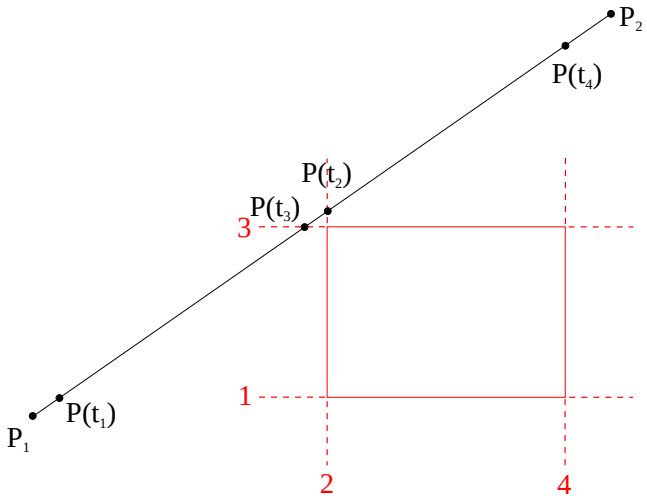
Akzeptieren: $0 < t_1 < t_2 < t_3 < t_4 < 1$

Clipp-Teil: $\overline{P(t_2)P(t_3)}$

$s_1 < 0 < s_2 < s_4 < s_3 < 1$

Clipp-Teil: $\overline{P(s_2)P(s_4)}$

Verwerfe:



$0 < t_1 < t_3 < t_2 < t_4 < t$ wird entsprechend $0 < s_2 < s_4 < s_1 < s_3 < 1$

Hinweise:

- triviales Akzeptieren und Verwerfen von Strecken, Clipp-Teil wird zurückgegeben
- Implementieren: Vermeide floating-point-Operationen

  z.B. Test mit oberer Fensterkante

  Parameterwert

  $$t = \frac{y_{max} - y_1}{y_2 - y_1}$$

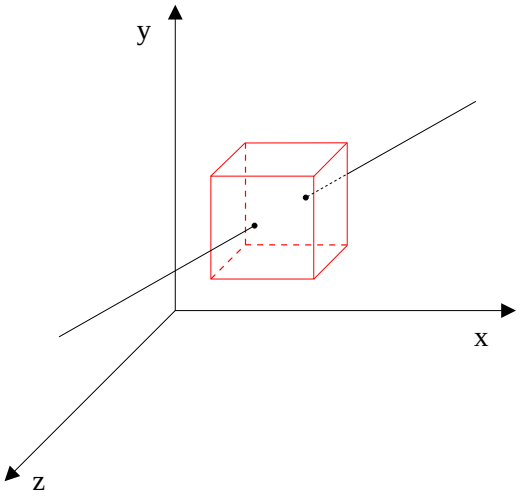
  teste mit

  $$t(y_2 - y_1) - (y_{max} - y_1)$$

  Tests können ohne floating-point-Operationen ausgeführt werden
- nur falls der Schnitt berechnet werden muss, sind floating-point-Operationen notwendig
- Vorteil gegenüber Cohen-Sutherland: keine mehrfache Berechnung mit ein und derselben Strecke

Erweiterung von Cohen-Sutherland in 3D

Clippen gegen achsenparallele Quader



betrachte Begrenzungen

$$x_{min} \leq x \leq x_{max}$$
$$y_{min} \leq y \leq y_{max}$$
$$z_{min} \leq z \leq z_{max}$$

Erweiterung:

4-Bit-Code der Punkte in der Ebene

6-Bit-Code der Punkte im Raum

4.4 Bemerkungen zu fortgeschrittenen Zeichentechniken

Ziel: Freiheit im Umgang mit Weltkoordinaten, Primitiven usw. soll organisiert werden

Vorgehensweise: Benutzen von Klassen ⇒ canvas (Leinwand), Zeichnen von Routinen in canvas

Einschränkung: Implementierung in OpenGL (prinzipiell DOS, Turbo C++ ... möglich)

nützliche Klassen: `class Point2, class IntRect, class RealRect`

Deklaration für canvas.h, Daten enthalten:

current position cp, window, viewport, windwow-viewport-Transformation

Arbeitsweise:

- canvas übernimmt Breite, Höhe des Bildschirmfensters als Argument

- enthält Funktionen um Größe des Fensters zu setzen und zurückzugeben
- enthält keine Daten zu window-viewport-Transformation
- `lineTo(), moveTo()` aktuelles Zeichnen in Weltkoordinaten

Anwendung: Schildkrötengrafik

Bezeichnung: Schildkröte

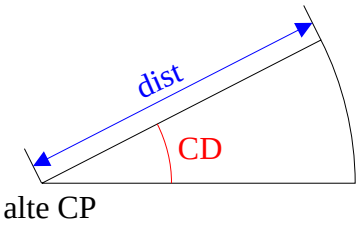
Bestimmung von Ort CP=current position

Vorgabe einer Richtung CD=current direction

Ziel: Klasse zur Kontrolle der Schildkröte

1. `turnTo(float angle)`
   übergibt Winkel CD=angle
2. `turn(float angle)`
   vollführt Drehung im mathematisch positiven Sinn um Winkel angle, CD+=angle
3. `forward(float dist,int isVisible)`
   vollführt geradlinige Bewegung von CP mit der Länge dist in CD

   `isVisible`$\neq$`0` wird sichtbare Linie gezeichnet, andernfals nicht



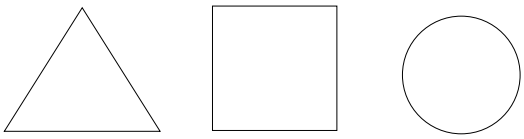
Beispiel: Polyspirale

```
for(some members of iterations)
{
    forward(length,1);  //zeichne Gerade in CP
    turn(angle);        //drehe um Winkel
    length+=increment;  //veraendert Laenge
}
```

4.5 Weitere Grafikprimitive

Definition:

Ein Polygon heißt regulär, falls alle Seiten gleich lang und alle Innenwinkel kongruent sind.



n=3 n=4 n=1000

Mathematisches Modell:

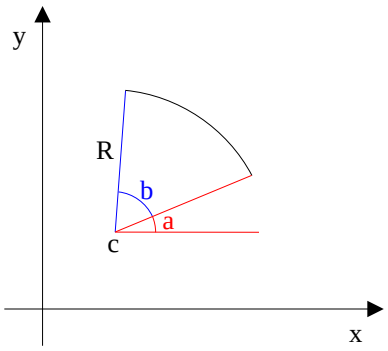
$$P_i = (Rcos(2\pi i(n), Rsin(2\pi(n)), i = 0, ..., n-1$$

R = Umkreisradius

Kreise und Kreisbögen:

Idee: Kreis = reguläres n-Eck mit „großer“Anzahl von Ecken

Kreisbögen:



Codefragment:

```
void drawArc(Point2 center,float radius,float startAngle,float sweep) {
    //startAngle, sweep in Grad
    const int n=30;                          //Anzahl der Segmente
    float angle=startAngle*3.14159/180;      //Umrechnung in Bogenmass
    float increment=sweep*3.141/(180*n);
    float cx=center getx(), cy=center gety(); //Uebergabe Mittelpunkt
    cvx.moveTo(cx+radius*cos(angle),cy+radius*sin(angle));
    for(int k=1;k<n,k++;angle+=agnleInc)
```

```
        cvs.lintTo(cx+radius*cos(angle),cy+radius*sin(angle));
}
```

Allgemein zu Parameterdarstellungen:

zwei Typen von Kurven:

- implizit:

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

z.B. Kreis in (0,0) mit Radius r:

$$x^2 + y^2 - r^2 = 0$$

Ellipse:

$$\left(\frac{x}{a}\right)^2 + \left(\frac{y}{b}\right) - 1 = 0$$

Spezialfall:

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

explizite Darstellung:

$$y = g(x)$$

- Parameterdarstellung:

$$x = x(t), y = y(t)$$

t Parameter, z.B.

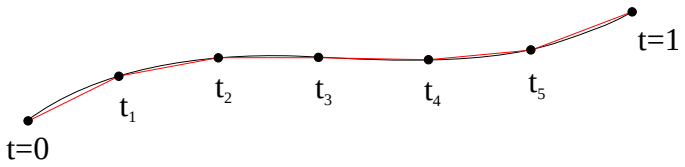
$$x = t^2, y = t^3$$

Kreis:

$$x = r\cos t, y = r\sin t, t \in [0, 2\pi]$$

Bemerkung: Zeichnen von parametrisiert gegebenen Kurven

$$P(t) = (x(t), y(t)), t \in [0,1]$$

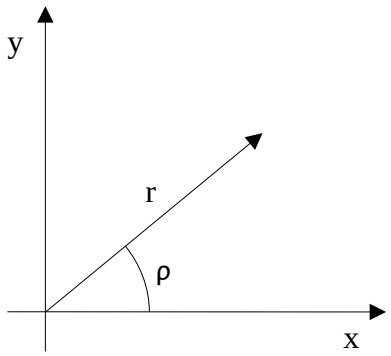


Unterteile [0,1] in hinreichend viele Teilintervalle und zeichne den Streckenzug

Hierzu:

```
glBegin(GL_LINES);
for(int i=0;i<m;i++)
    glVertex2f(x(t[i]),y(t[i]));
glEnd();
```

Polarkoordinaten:



$$x = r\cos\varphi$$
$$y = r\sin\varphi$$
$$\varphi = arctan(y/x)$$
$$r = \sqrt{x^2+y^2}$$

Benutze `Atan2(x,y)` (C++-Funktion)

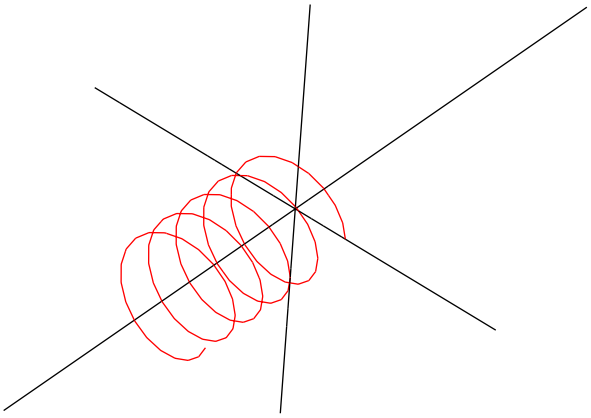
Räumliche Kurven in Parameterdarstellung

$$P(t) = (x(t), y(t), z(t))$$

t Parameter

1. z.B. Helix

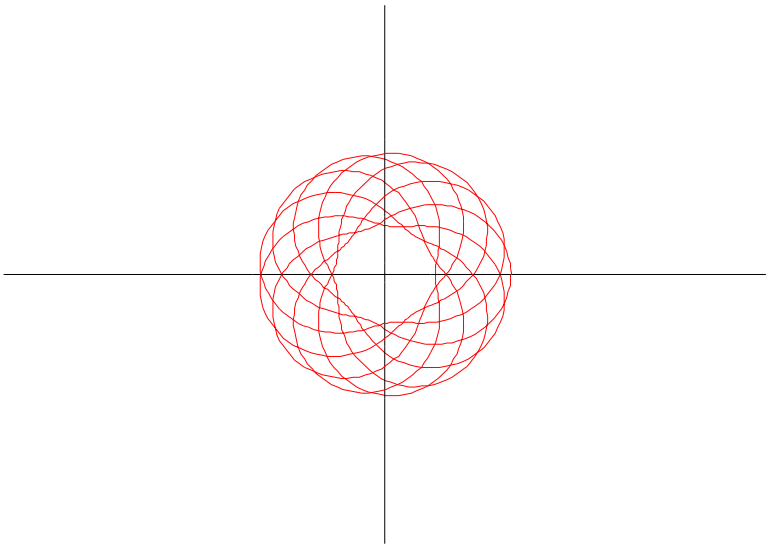
$$x(t) = \cos t, y(t) = \sin t, z(t) = bt$$

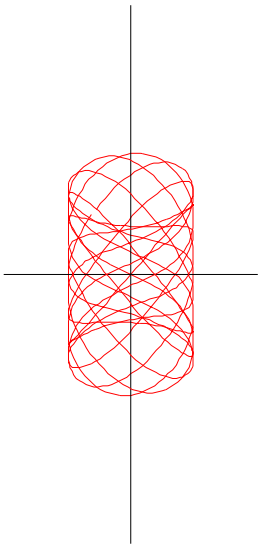


2. Torusknoten

$$x(t) = (a + b\cos(qt))\cos(pt)$$
$$y(t) = (a + b\cos(qt))\sin(pt)$$
$$z(t) = c\sin(qt)$$

z.B.: $q = 13$, $p = 8$





Zusammenfassung:

- Entwicklung von Programmierwerkzeugen
- Objekte in Weltkoordinaten definiert und auf viewport übertragen

- einfache Clipp-Algorithmen erarbeitet und implementiert
- Darstellungsparameter definierter Kurven

Kapitel 5

Mathematische Hilfsmittel

Ziele:

- Wiederholung der Vektorrechnung
- Zusammenhang von geometrischen Gebilden und algebraischen Darstellungen
- Beschreibung wichtiger Koordinatensysteme
- Geometrische Anwendungen algebraischer Konstruktionen

  Schnittpunktberechnung
  Abstandsbestimmung
  orthogonale Projektion u.a.

5.1 Grundlagen über Vektoren

Wiederholung:

Vektor = Element eines Vektorraumes

Vektorraum hier $\mathbb{R}^n$, n=2,3,4: $\mathbb{R}^n = \{(a_1, ...a_n), a_i \in \mathbb{R}, i = 1, ...n\}$

Operationen:

1. Addition

$$(a_1, ...a_n) + (b_1, ..., b_n) = (a_1 + b_1, ..., a_n + b_n)$$

   Rechenregeln:

   - abgeschlossen
   - assoziativ
   - neutrales Element $\underline{0}$=(0,...,0)
   - entgegengesetztes Element
   - kommutativ

2. Multiplikation mit Skalar

$$\lambda(a_1,...,a_n) = (\lambda a_1,...,\lambda a_n)$$

Rechenregeln:

- Distributivgesetz

$$\begin{aligned}\lambda(\underline{a}+\underline{b}) &= \lambda\underline{a}+\lambda\underline{b}\\ (\lambda\mu)\underline{a} &= \lambda\underline{a}+\mu\underline{a}\\ (\lambda\mu)\underline{a} &= \lambda(\mu\underline{a})\end{aligned}$$

- Unität

$$1\underline{a} = \underline{a}$$

Linearkombinationen:

$$a_1,...a_n \in \mathbb{R} : \sum \lambda_i a_k$$

Wiederholung:

affiner Raum: Punktmenge P mit Vektorraum V (für uns: $P = V = \mathbb{R}$)

Rechenregeln:

1. Zu jedem Punkt $p \in P$ und jeden Vektor $\underline{v} \in V$ existiert ein Punkt $p+\underline{v} \in P$ mit

$$\begin{aligned}p+\underline{0} &= p\\ p+(\underline{v}+\underline{w}) &= (p+\underline{v})+\underline{w}\end{aligned}$$

für $p \in P$ und $\underline{v},\underline{w} \in V$

2. Zu je zwei Punkten $p,q \in P$ existiert ein eindeutig bestimmter Vektor $\underline{v} \in V$ mit $p+\underline{v} = q$

Wichtig: Unterschiede zwischen Punkten und Vektoren im $\mathbb{R}^n$

man setzt oft den Vektor $\underline{v}$ mit dem Ortsvektor $0+\underline{v} = p$ gleich

Skalarprodukt im $\mathbb{R}^n$:

$\underline{a} = (a_1,...,a_n), \underline{b} = (b_1,...b_n)$

Def.:

$$\begin{aligned}\underline{a}\cdot\underline{b} &:= a_1b_1+...+a_nb_n \in \mathbb{R}\\ \cdot &: \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}\end{aligned}$$

Eigenschaften:

1. $\underline{a} \cdot \underline{b} = \underline{b} \cdot \underline{a}$
2. $(\underline{a} + \underline{b}) \cdot \underline{c} = \underline{a} \cdot \underline{c} + \underline{b} \cdot \underline{c}$
3. $(s\underline{a}) \cdot \underline{b} = s(\underline{a} \cdot \underline{b})$
4. $\underline{a} \cdot \underline{a} = \underline{a}^2 \geq 0$

   $$|\underline{a}| := \sqrt{\underline{a} \cdot \underline{a}} = \sqrt{a_1^2 + \ldots + a_n^2}$$

   heißt Länge oder Norm von $\underline{a}$
5. Es gilt (Schwarzsche Ungleichung):

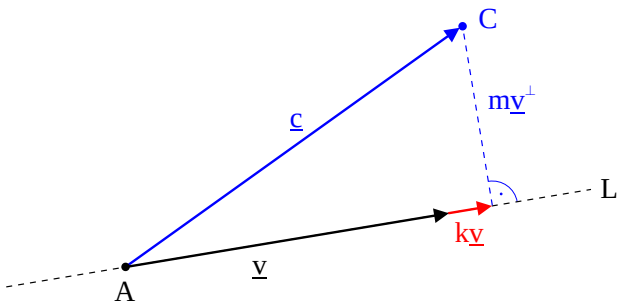
   $$|\underline{a} \cdot \underline{b}| \leq |\underline{a}||\underline{b}|$$

6. Für $\underline{a} \neq \underline{0}, \underline{b} \neq \underline{0}$ definiert man den Winkel $\sphericalangle(\underline{a}, \underline{b})$ durch

   $$\cos \sphericalangle(\underline{a}, \underline{b}) = \frac{\underline{a} \cdot \underline{b}}{|\underline{a}| \cdot |\underline{b}|}$$

   (Bemerkung: Es gilt: $-|\underline{a}| \cdot |\underline{b}| \leq \underline{a} \cdot \underline{b} \leq |\underline{a}||\underline{b}|$, wegen Schwarzscher Ungleichung, also macht die Definition Sinn!)
7. $\underline{a} \neq \underline{0} \neq \underline{b}$ heißen orthogonal, falls $\underline{a} \cdot \underline{b} = \underline{0}$, abgekürzt $\underline{a} \perp \underline{b}$

orthogonale Projektion:



Aufgaben:

1. Wie weit ist C von der Geraden durch A und $\underline{v}$ entfernt?
2. Wo liegt der Fußpunkt des Lotes von C auf L?
3. Wie zerlegt man $\underline{c}$=C-A in ein Vielfaches von $\underline{v}$ und einem dazu sekrechten Vektor?

Hierzu: betrachte $\mathbb{R}^2$

$\underline{v} = (v_x, v_y), \underline{v}^\perp = (v_y, -v_x)$

nicht eindeutig: $\underline{v}^\perp = (-v_y, v_x)$ und alle Vielfachen

Ansatz:

$$\underline{c} = k \cdot \underline{v} + m \cdot \underline{v}^\perp$$

Bestimme k und m:

multipliziere Skalar mit $\underline{v}$ bzw. $\underline{v}^\perp$

$$\underline{c} \cdot \underline{v} = k \cdot \underline{v} \cdot \underline{v} + m \cdot \underbrace{\underline{v}^\perp \cdot \underline{v}}_{0}$$

Lösung:

$$k = \frac{\underline{c} \cdot \underline{v}}{|\underline{v}|^2}$$

$$m = \frac{\underline{c} \cdot \underline{v}^\perp}{|\underline{v}^2}$$

Abstand:

$$|m \cdot \underline{v}^\perp| = |\frac{\underline{v}^\perp \cdot \underline{c}}{|\underline{v}|^2} \cdot \underline{v}^\perp| = \frac{\underline{v}^\perp (C - A)}{|\underline{v}|}$$

= Hessesche Abstandsformel

Fußpunkt des Lotes:

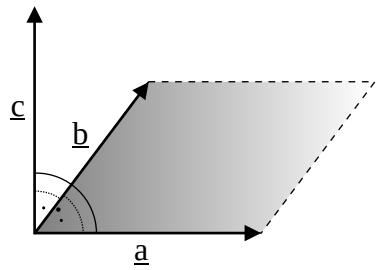
$$A + \frac{\underline{v} \cdot (C - A)}{|\underline{v}|} \cdot \underline{v}$$

Kreuzprodukt, Vektorprodukt nur im $\mathbb{R}^3$!

gegeben: $\underline{a}, \underline{b} \in \mathbb{R}^3$

gesucht: $\underline{c} \in \mathbb{R}^3$ mit folgenden Eigenschaften:

1. $\underline{c} \perp \underline{a}, \underline{c} \perp \underline{b}$
2. $|\underline{c}|$ = Fläche des durch $\underline{a}$ und $\underline{b}$ aufgespannten Parallelogramms



3. $\underline{a}, \underline{b}, \underline{c}$ bilden ein Rechtssystem

Lösung:

$$\underline{a} = (a_x, a_y, a_z)$$
$$\underline{b} = (b_x, b_y, b_z)$$

eindeutige Lösung:

$$\underline{a} \times \underline{b} = (a_y b_z - a_z b_y, a_z b_x - a_x b_z, a_x b_y - a_y b_x)$$

Merkregel: $\begin{pmatrix} i & j & k \\ a_x & a_y & a_z \\ b_x & b_y & b_z \end{pmatrix}$ → Determinante nach 1. Zeile entwickeln

Kreuzprodukt: $\times : \mathbb{R}^3 \times \mathbb{R}^3 \to \mathbb{R}^3$

Hilfsmittel:

Matrizenmultiplikation

$$(a_1, ..., a_n) \cdot \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix} = \sum a_i b_i$$

Einführung einer Matrix $M_{\underline{a}} = \begin{pmatrix} 0 & -a_z & a_y \\ a_z & 0 & -a_x \\ -a_y & a_x & 0 \end{pmatrix}$ = schiefsymmetrisch

$$M_{\underline{a}} \cdot \underline{b}^T = \begin{pmatrix} 0 & -a_z & a_y \\ a_z & 0 & -a_x \\ -a_y & a_x & 0 \end{pmatrix} \cdot \begin{pmatrix} b_x \\ b_y \\ b_z \end{pmatrix} = \underline{a} \times \underline{b}$$

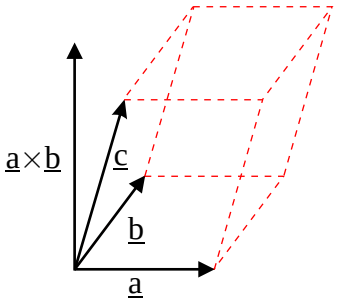
Rechenregel: $\underline{a} \times \underline{b} = -\underline{b} \times \underline{a}$

Spatprodukt:

$\underline{a}, \underline{b}, \underline{c} \in \mathbb{R}^3$

$(\underline{a} \times \underline{b}) \cdot \underline{c}$ Skalar

$$(\underline{a} \times \underline{b}) \cdot \underline{c} = det \begin{pmatrix} a_x & a_y & a_z \\ b_x & b_y & b_z \\ c_x & c_y & c_z \end{pmatrix}$$



Das Skalarprodukt $(\underline{a} \times \underline{b}) \cdot \underline{c}$ ist das orientierte Volumen des von $\underline{a}, \underline{b}, \underline{c}$ aufgespannten Parallelepipeds.

Bemerkung: $(\underline{a} \times \underline{b}) \cdot \underline{c} = 0$ genau dann, wenn $\underline{a}, \underline{b}, \underline{c}$ linear abhängig sind.

5.2 Darstellung geometrischer Objekte

affine Linearkombination

$\underline{v}_1, ..., \underline{v}_m \in \mathbb{R}^n$ gegebene Vektoren

$\underline{w} = a_1\underline{v}_1 + a_2\underline{v}_2 + ... + a_m\underline{v}_m$ heißt affine Linearkombination, falls $a_1 + a_2 + ... + a_m = 1$ gilt.

$\underline{w}$ heißt konvexe Linearkombination, falls $a_1 + ... + a_m = 1$ und $a_i \in [0,1], i = 1, ..., m$.

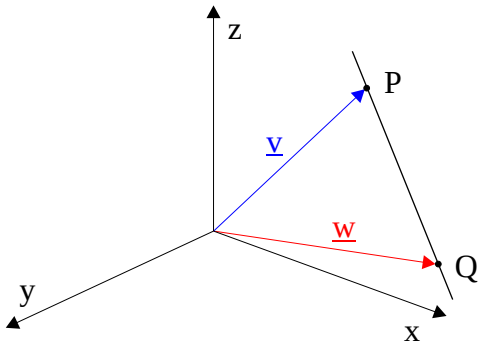
Beispiele:

1. $\underline{v}, \underline{w}$ linear unabhängig

   a) Menge aller Linearkombinationen von $\underline{v}, \underline{w}$ ist eine Ebene durch P, Q und den Ursprung

   b) Menge der affinen Linearkombinationen:
   $t\underline{v} + (1-t)\underline{w} = \underline{w} + t(\underline{v} - \underline{w})$ = Gerade durch P und Q

   c) Menge der konvexen Linearkombinationen:
   $\underline{v} + (1-t)\underline{w}$, $t \in [0,1]$ = Strecke $\overline{PQ}$

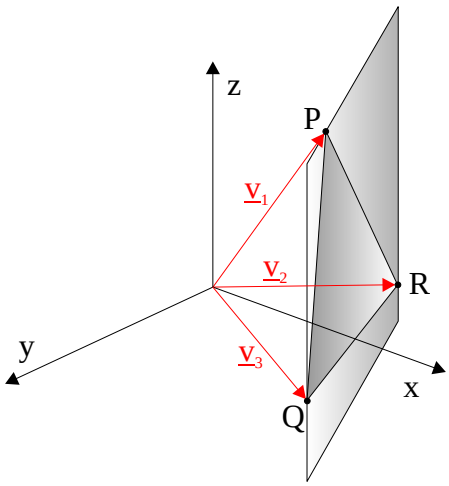


2. $\underline{v}_1, \underline{v}_2, \underline{v}_3$ linear unabhängig

   a) Menge der affinen Linearkombinationen
   $a_1\underline{v}_1 + a_2\underline{v}_2 + (1-a_1-a_2)\underline{v}_3 = \underline{v}_3 + a_1(\underline{v}_1 - \underline{v}_3) + a_2(\underline{v}_2 - \underline{v}_3)$
   = Ebene durch die Punkte P, Q, R

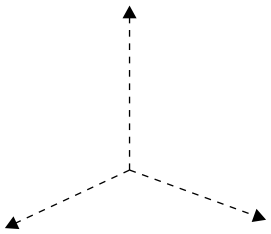
   b) Menge der konvexen Linearkombinationen = das durch P, Q, R definierte Dreieck

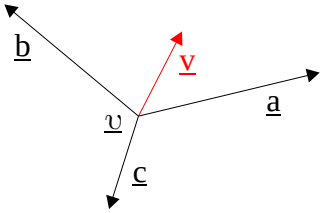
(Hierzu: konvex = mit je zwei Punkten des Gebildes gehört auch die gesamte Strecke zwischen den Punkten zum Gebilde)



Koordinatensysteme

Position eines kartesischen Koordinatensystems gegenüber einem zweiten





Darstellung eines Vektors $\underline{v}$ durch Tripel (v_1, v_2, v_3) bezüglich der Basis$\{\underline{a}, \underline{b}, \underline{c}\}$:

$$\underline{v} = v_1\underline{a} + v_2\underline{b} + v_3\underline{c}$$

Darstellung eines Punktes P

Darstellung des Vektors $p - \vartheta = p_1\underline{a} + p_2\underline{b} + p_3\underline{c}$

d.h. $p = \vartheta + p_1\underline{a} + p_2\underline{b} + p_3\underline{c}$ erfordert Tripel (p_1, p_2, p_3) bei Kenntnis von $\underline{\vartheta}$=Ursprung

Vereinheitlichte Darstellung mit homogenen Koordinaten

Vektor $\underline{v}$:

$$\underline{v} = (\underline{a}, \underline{b}, \underline{c}, \underline{\vartheta}) \begin{pmatrix} v_1 \\ v_2 \\ v_3 \\ 0 \end{pmatrix} = v_1\underline{a} + v_2\underline{b} + v_3\underline{c}$$

Punkt P:

$$P = (\underline{a}, \underline{b}, \underline{c}, \vartheta) \begin{pmatrix} p_1 \\ p_2 \\ p_3 \\ 1 \end{pmatrix} = p_1\underline{a} + p_2\underline{b} + p_3\underline{c} + \vartheta$$

Vereinheitlichte Schreibweise von Vektoren und Punkten

- Vektor = 4-Tupel: $(v_1, v_2, v_3, 0)$
- Punkt = 4-Tupel: $(p_1, p_2, p_3, 1)$

Fundamentaler Ansatz der Computergrafik: Linearkombinationen von Vektoren

- Differenz von zwei Punkten P, Q ist Vektor
- Summe eines Vektors $\underline{v}$ mit einem Punkt P ist Punkt
- Summe zweier Vektoren $\underline{u}, \underline{v}$ ist Vektor
- skalares Vielfaches $\lambda\underline{v}$ eines Vektors $\underline{v}$ ist Vektor

Linearkombination von Punkten

$P = (p_1, p_2, p_3, 1)$, $Q = (q_1, q_2, q_3, 1)$

betrachte $fP + gQ = (fp_1 + gq_1, fp_2 + gq_2, fp_3 + gq_3, f + g)$

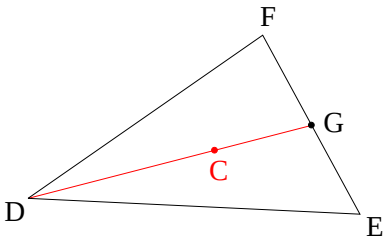
Diskussion:

1. $f + g = 0$: $fP + gQ = f(P - Q) = f\underline{v}$ erhält man Vektor
2. $f + g = 1$: $fP + gQ = (fp_1 + gq_1, fp_2 + gq_2, fp_3 + gq_3, 1)$ erhält man Punkt

   $fP + (1 - g)Q$ affine Linearkombination der Punkte P, Q
3. $f + g \neq 0, 1$: man erhält keine legitime Linearkombination von Punkten!

(Vorbemerkung: multipliziere mit $\frac{1}{f+g}$ und man erhält einen projektiven Punkt, Anwendung in der perspektivischen Projektion)

Beispiel: Dreieck D,E,F



G=$\frac{1}{2}E + \frac{1}{2}F$ Mittelpunkt der Strecke $\overline{EF}$

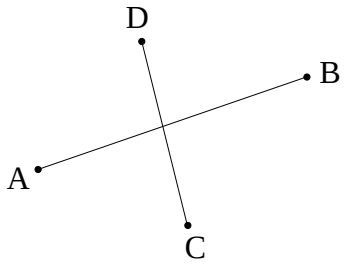
Bestimme Punkt C auf $\overline{DG}$, der Strecke $\overline{DG}$ im Verhältnis 2:1 teilt.

Lösung:

$$\begin{aligned} C &= D + \frac{2}{3}(G - D) \\ &= D + \frac{2}{3}(\frac{1}{2}E + \frac{1}{2}F - D) \\ &= \frac{1}{3}(D + E + F) \end{aligned}$$

Schwerpunkt des Dreiecks = Schnittpunkt aller Seitenhalbierenden

5.3 Durchschnitte von Strecken der Ebene



Ansatz:

$$AB(t) = A + (B-A)t = A + \underline{b}t$$
$$CD(t) = C + (D-C)t = C + \underline{d}t$$

viele Lagemöglichkeiten

Schnittpunktberechnung:

$$A + \underline{b}t = C + \underline{d}u \mid \cdot \underline{d}^{\perp}$$
$$(A-C)\cdot \underline{d}^{\perp} + \underline{bd}^{\perp}t = 0$$
$$\underline{bd}^{\perp}t = \underline{cd}^{\perp}$$

1. Fall: $\underline{bd}^{\perp} \neq 0$

   $t = \frac{\underline{cd}^{\perp}}{\underline{bd}^{\perp}}$ und $A + \underline{b} \cdot \frac{\underline{cd}^{\perp}}{\underline{bd}^{\perp}}$ ist Schnittpunkt der beiden Geraden

2. Fall: $\underline{bd}^{\perp} = 0$

   d.h. $\underline{d}$ und $\underline{b}$ sind parallel
   falls $\underline{cd}^{\perp} \neq 0$ kein Schnittpunkt
   falls $\underline{cd}^{\perp} = 0$, $\underline{c}$, $\underline{d}$ parallel und beide Geraden identisch

Hinweis: echter Schnittpunkt der Strecken liegt vor, falls $u,t \in [0,1]$

Problem: Wie kann man unnötige Schnittpunktberechnungen vermeiden?

Ziel: Verringerung des Aufwandes

Orientierter Flächeninhalt eines Dreiecks

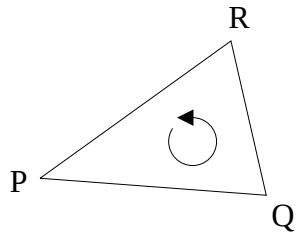
$P = (p_x, p_y)$, $Q = (q_x, q_y)$, $R = (r_x, r_y)$

$$\triangle PQR = det \begin{pmatrix} p_x & p_y & 1 \\ q_x & q_y & 1 \\ r_x & r_y & 1 \end{pmatrix}$$

heißt orientierter Flächeninhalt des $\triangle PQR$, d.h. $|\triangle PQR|$ gibt Flächeninhalt an

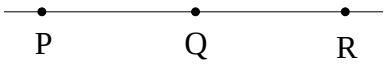
Beobachtung:

$\triangle PQR > 0$ falls



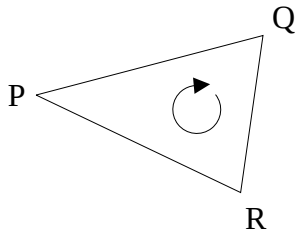
mathematisch positiv

$\triangle PQR = 0$ falls



Gerade

$\triangle PQR < 0$ falls



mathematisch negativ

Definition:

$o(PQR) = sign \triangle PQR$: Orientierung von $\triangle PQR$

Bemerkung:

Notwendig und hinreichend dafür, dass sich die Strecken $\overline{AB}$ und $\overline{CD}$ nicht schneiden ist die Bedingung

$$o(ABC) \cdot o(ABD) < 0 \wedge o(CDA) \cdot o(CDB) < 0$$

Ergebnis: Reduktion zur Berechnung der Schnittpunkte auf notwendiges Maß

5.4 Schnitte von Geraden mit Ebenen und Clippen

Clippen: prinzipielle Technik

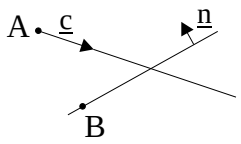
Gegeben:

Strahl $R(t) = A + \underline{c}t$

Gerade, Ebene $\underline{n}(P - B) = 0$

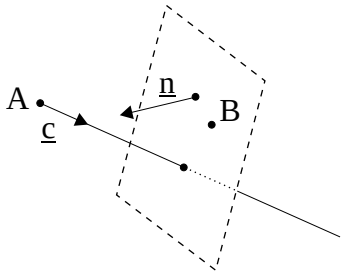
Skizze:

2D:



$$\underline{n}(P - B) = 0$$

3D:



E: $\underline{n}(P - B) = 0$

Schnittpunkt (Aufschlagpunkt): $t = t_h$

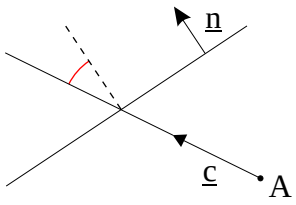
$$(A + \underline{c}t_h - B) \cdot \underline{n} = 0$$
$$t_h = \frac{\underline{n}(B - A)}{\underline{n} \cdot \underline{c}}$$

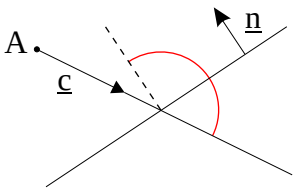
Schnittpunkt:

$$A + \underline{c}\frac{\underline{n}(B - A)}{\underline{n} \cdot \underline{c}}$$

Was passiert für $\underline{n} \cdot \underline{c} = 0$?

genauer:



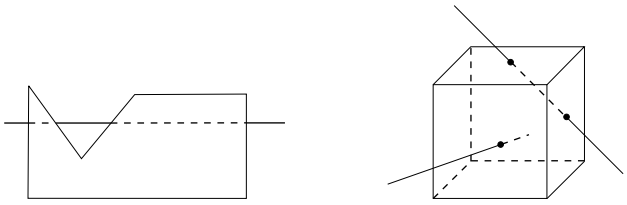


Fälle:

1. $\underline{n} \cdot \underline{c} > 0$: Strahl verläuft wie $\underline{n}$
2. $\underline{n} \cdot \underline{c} = 0$: Strahl ist parallel zur Geraden
3. $\underline{n} \cdot \underline{c} < 0$: Strahl verläuft entgegen $\underline{n}$

5.5 Schnittprobleme mit Polygonen

Darstellung von graphischen Primitiven durch Polygone

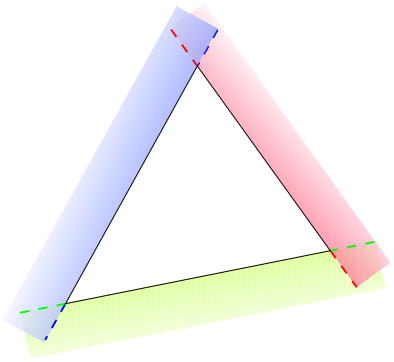


Probleme:

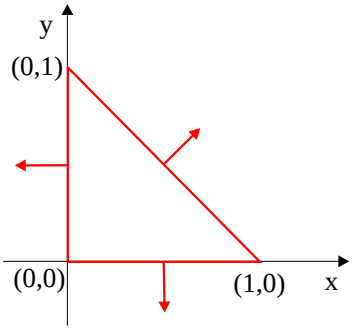
1. Test, ob ein gegebener Punkt innerhalb oder außerhalb eines Objektes liegt
2. Wo schneidet ein Strahl erstmals ein Objekt?
3. Welcher Teil einer Strecke liegt innerhalb eines Objektes?

Einschränkung für Betrachtung:

> konvexe Polygone bzw. konvexe Polyeder (konvexe Polygone bzw. konvexe Polyeder werden durch Halbebenen bzw. Halbräume begrenzt)



Beispiel eines Dreiecks

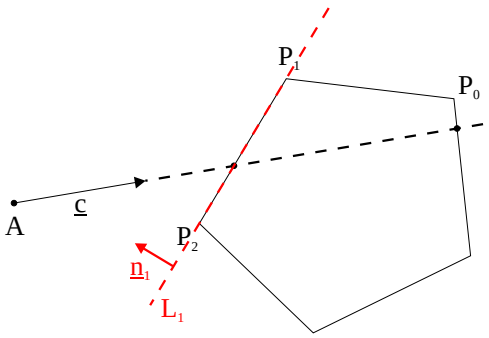


nach außen orientierte Normalenvektoren

begrenzende Geraden

$$(-1,0)\cdot(P-(0,0))=0 \quad (0,-1)\cdot(P-(0,0))=0 \quad (1,1)\cdot(P-(1,0))=0$$

Schnittberechnung und Clippen für konvexe Polygone



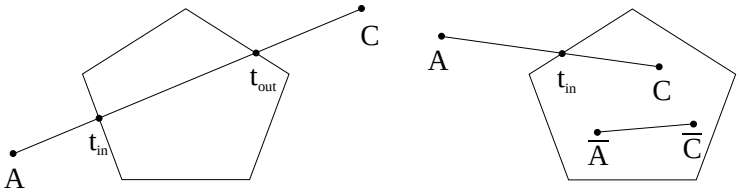
Strahl: $A+\underline{c}t$ schneidet Polygon P

Falls überhaupt Schnittpunkt vorliegt, existieren i.A. zwei Schnittpunkte

$$A+\underline{c}t_{in}$$
$$A+\underline{c}t_{out}$$

(konvexes Polygon)

Bemerkung: Auffinden von t_{in}, t_{out} löst auch das Clipp-Problem für $A+\underline{c}t$



Betrachte: $A+\underline{c}t$, $t \in [0,1]$, Strecke $\overline{AC}$ als Teil eines Strahls

Berechne: t_{in}, t_{out}

Test mit Punkten

$$A' = A+\underline{c}\cdot max(0,t_{in})$$
$$C' = A+\underline{c}\cdot min(t_{out},1)$$

Voraussetzung:

Jede Begrenzungsgerade ist gegeben durch ein Paar $\{B,\underline{n}\}$, B Punkt der Geraden, $\underline{n}$ von Polygon nach außen gerichteter Normalenvektor

Tests:

$\underline{n}\cdot\underline{c} > 0$: Strahl geht aus P heraus

$\underline{n}\cdot\underline{c} = 0$: Strahl ist parallel zur Seite

$\underline{n}\cdot\underline{c} < 0$: Strahl geht in P hinein

Vorgehensweise:

Für jede Begrenzungsgerade von P berechne:

1. Den Schnittpunkt vom Strahl $A+\underline{c}t$ mit Begrenzungsgerade

$$t_{hit} = \frac{\underline{n}(B-A)}{\underline{nc}}$$

2. Teste, ob der Strahl $A+\underline{c}t$ das Polygon verlässt oder nicht

   a) Falls der Strahl in das Polygon hineingeht, kann der Strahl nicht früher hinein gehen als t_{hit}, wie gerade gefunden. D.h. wir verwalten die kleinste Größe t_{hit} als t_{in}. Für jeden Eintritt ersetzen wir t_{in} durch $max(t_{hit}, t_{in})$.

   b) Falls der Strahl das Polygon verlässt, verwalten wir späteste Verlasszeit als t_{out}. Für jeden austretenden Strahl ersetzen wir t_{out} durch $min(t_{out}, t_{hit})$.

Also:

1. Initialisiere

   Clipp-Problem: [0,1]

   Schnitt-Problem: [a,b] a$\ll$0, b$\gg$0

2. Für jede Begrenzungsgerade $\{B, \underline{n}\}$ berechne t_{hit} und entscheide, ob der Strahl $A+\underline{c}t$ in das Polygon hinein- oder herausragt

   a) Falls t_{hit} zu einem hineinragenden Strahl gehört, setze $t_{in} = max(t_{in}, t_{hit})$

   b) Falls t_{hit} zu einem hinausragenden Strahl gehört, setze $t_{out} = min(t_{out}, t_{hit})$

3. Falls $t_{in} > t_{out}$ liegt der Strahl außerhalb von P und wir beenden. Falls $[t_{in}, t_{out}] \neq \emptyset$ berechne

$$A' = A + \underline{c} \cdot max\{0, t_{in}\}$$
$$C' = A + \underline{c} \cdot min\{t_{out}, 1\}$$

   und $\overline{A'C'}$ ist die geclippte Strecke. Für das Schnittproblem erhalte ich A' und C' als Ausgangspunkte.

Cyrus-Beck-Algorithmus

→ Clippen an einem konvexen Polygon

Quellcode: Folie (Cyrus_Beck_Clip)

```
//Prozedur clippt gegen jede Linie in L und speichert in seg
//Rueckgabe:
    //0 Intervall=leer
    //1 Teil der Strecke liegt in P
//LineSegment, LineList, Vector2 => geeignete Datentypen
//numer=n*(B-C)
//denom=n*c
```

Routine: Folie (ChopCI)

```
//benutzt "numer", "denom", um tHit zu berechnen
//=> Effekt: Abschneiden des Teils vom Kandidatenintervall, das nicht
             mehr relevant ist
```

Bemerkung:

1. $\underline{n} \cdot \underline{c} = 0$ entspricht der Parallelität des Strahls zur Begrenzungseite
2. Es gibt eine 3D-Variante des Cyrus-Beck-Alg. analog für 3-dim. konvexe Polygone

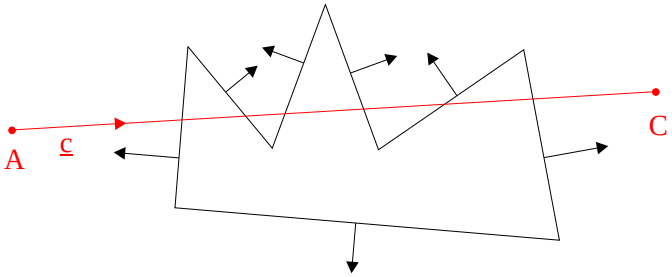
   → Begrenzungsstrecken werden zu Begrenzungsflächen

   → Strecke ⇒ Strecke im Raum

   `ChopCI()` Datentypen zu 3D-Daten erweitern

Clippen an beliebigen Polygonen

Generelles Problem:



Wann liegt der Strahl innerhalb, wann außerhalb des Polygons?

Polygon:

geg. durch Liste von Ecken $P_0, P_1, ..., P_{N-1}$

mit Voraussetzung: Polygon ist einfach, nicht überschlagen

Rückgabe: mehrere Streckenstücke

Schnittproblem: Finde Schnittpunkte mit der Folge der Strecken von P

Berechne Kantenvektor $\underline{l}_i$

$$\underline{l}_i = P_{i+1} - P_i, i = 0, ..., N-1$$

Strahl: $A + \underline{c}t$

Seitengerade: $P_i + l_i * U$

Schnittbedingung:

$$\begin{aligned} A + \underline{c}t &= P_i + l_i * U \\ \underline{c}t &= b_i + l_i * U \\ b_i &= P_i - A \end{aligned}$$

Lösung:

$$t = \frac{\underline{l}_i^{\perp} * \underline{b}_i}{\underline{l}_i^{\perp} * \underline{c}}$$

$$U = \frac{\underline{c}^{\perp} * \underline{b}_i}{\underline{l}_i^{\perp} * \underline{c}}$$

Vorgehensweise: Auffinden aller t_{hit} und Abspeichern in einer Liste

→ Initialisiere hitList als leer

```
for(int i=0;i<N;i++)
{
    bilde bi,li fuer i-te Seite;
    bestimme t,U;
    if(U in [0,1])
    {
        fuege t zu hitList hinzu;
    }
}
```

Clipp-Problem:

- Auffinden der t-Intervalle für die der Strahl innerhalb des Polygon ist
- Sortiere hitList und wähle Paare von t-Werten für Intervalle

Schnitt-Problem: Eintritt des Strahls in Polygon = kleinster Wert initialisiert

Beispiel: Folie(6a)

Strecke: A=(1,1), B=(8,2)

Polygon: (3,2), (2,0), (0,-1), (6,2), (4,1)

(b)

hitList={0,23, 0,71, 0,6, 0,38}

sort={0,23, 0,38, 0,6, 0,71}

Intervalle von t: [0,23, 0,38], [0,6, 0,7]

Problem:

Was passiert aber bei Strecken, die wie $\overline{DE}$ liegen? → umgekehrtes Clippen:

Finden: # Schnittpunkte ist ungerade (Parität) ⇒ 1 Punkt liegt innerhalb

Problem ist also: Anfangspunkt der Strecke liegt innerhalb des Polygon

$\Rightarrow$ für Anfangspunkt und Endpunkt wird # Schnittpunkte des Strahls in Richtung des andern Punktes....

- wenn #(Anfangspunkte) = #(Endpunkte) = gerade $\Rightarrow$ beide liegen außerhalb
- wenn #(Anfangspunkte) = #(Endpunkte) = ungerade $\Rightarrow$ beide liegen innerhalb des Polygon
- wenn #(Anfangspunkte) $\neq$ #(Endpunkte) $\Rightarrow$ eine liegt innen, eine außen

Zusammenfassung:

- Vektoren und Punkte (affine Geometrie) sind geeignete Hilfsmittel zur Beschreibung geometrischer Objekte
- Skalar- und Vektorprodukt als nützliche Hilfsmittel
- zwei prinzipielle Darstellungen geometrischer Objekte (implizite Darstellung und Parameterdarstellung)
- homogene Koordinaten zur vereinheitlichten Darstellung von Punkten und Vektoren, affiner Linearkombinationen
- Clipp-Algorithmen für Polygone

Literaturempfehlungen: [10], [11]

Kapitel 6

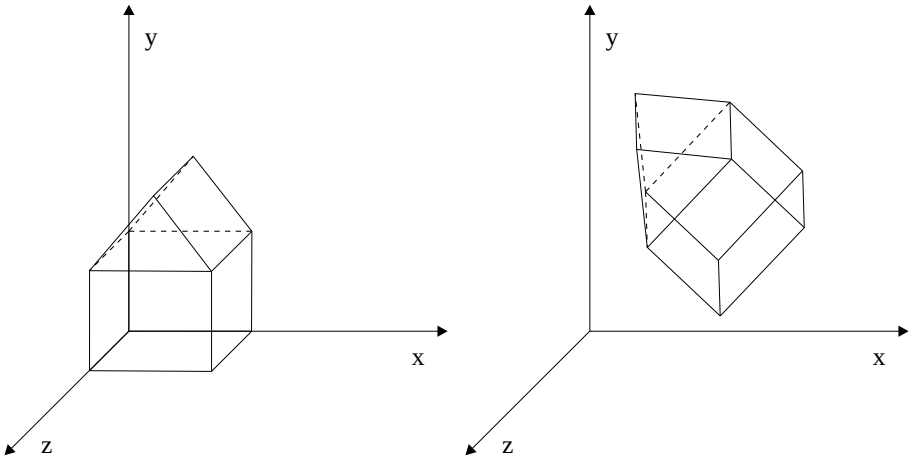
Transformation von Objekten

- Erweiterung von Window $\Rightarrow$ Viewport - Transformationen
- Hierzu:
  - Konzept der affinen Abbildung
  - Kombination von Drehung, Skalierung, Transformation
  - Umsetzen der Konzepte in graph. Anwendungen
  - Transformation von Koordinatensystemen
  - Kamerapositionierung
- Hier: Klärung des mathematischen Hintergrundes und Umsetzung in OpenGL

6.1 Begriff der Transformation

Bisher: Window - Viewport - Transformation

Fortsetzung: affine Transformation



Nutzen der Transformationen:

1. Komposition einer Szene aus Basisobjekten
2. Figuren generieren mit symmetrischen Eigenschaften (Folie 6c)
3. Betrachen einer Szene aus verschiedenen Blickwinkeln (Folie 6d)
4. Realisieren einfacher Animationen

Ziel: Graphische Routine

Graphik-Pipeline = Sequenz von Operationen, die auf alle Punkte angewendet werden

OpenGL-Pipeline (Folie 6e)

$P_1, P_2, \ldots$ - Punkte

CT: current transformation

$Q_1, Q_2, \ldots$ geänderte Punkte unter CT

Problem:

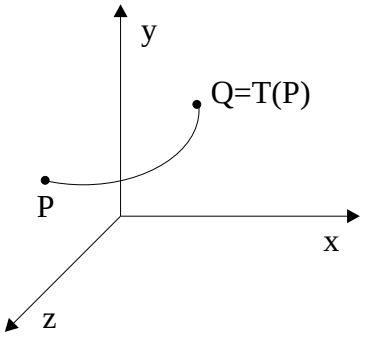
CT ist Tool zur Manipulation graph. Objekte

ist wichtig zu wissen, wie CT einzurichten ist, um gewünschte Effekte zu erzielen

Voraussetzung: Kenntnis affiner Transformationen

Transformation von Punkten und Objekten

Transformation überführt $P \in \mathbb{R}^2, \mathbb{R}^3$ in neuen Punkt Q



Anwendung auf Objekte geschieht punktweise

Forderung: T soll Geometrie „respektieren“

Bezeichnung:

$$\begin{pmatrix} Q_x \\ Q_y \\ Q_z \\ 1 \end{pmatrix} = T \begin{pmatrix} P_x \\ P_y \\ P_z \\ 1 \end{pmatrix}$$

Definition:

Eine affine Transformation ist von der Form für Punkte:

$$\begin{pmatrix} Q_x \\ Q_y \\ Q_z \\ 1 \end{pmatrix} = \underbrace{\begin{pmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ 0 & 0 & 0 & 1 \end{pmatrix}}_{=:M} \begin{pmatrix} Q_x \\ Q_y \\ Q_z \\ 1 \end{pmatrix}$$

und für Vektoren:

$$\begin{pmatrix} w_x \\ w_y \\ w_z \\ 0 \end{pmatrix} = M \begin{pmatrix} v_x \\ v_y \\ v_z \\ 0 \end{pmatrix}$$

Generelle Voraussetzung:

$$\overline{M} = \begin{pmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{pmatrix}$$
$$det\overline{M} = detM \neq 0$$

Beispiel:

affine Transformation der Ebene

1. Translation

$$\begin{pmatrix} 1 & 0 & m_x \\ 0 & 1 & m_y \\ 0 & 0 & 1 \end{pmatrix}$$

2. Skalierung

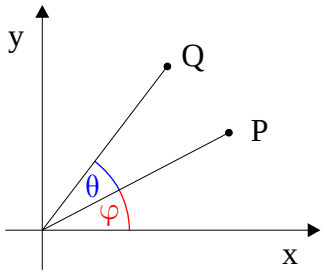
$$\begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

   $s_x \cdot s_y \neq 0$ (Determinante)
   Beispiel: $(s_x, s_y) = (-1, 2)$
   reine Spiegelung (-1,1)
   uniforme Skalierung: $s_x = s_y = 1$

3. Drehung (Rotation)

$$\begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Herleitung



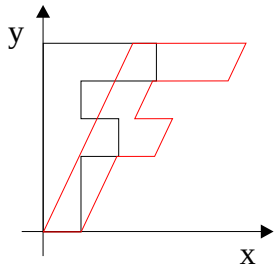
$$
\begin{aligned}
P &= (r\cos\rho, r\sin\rho) \\
P &= (r\cos(\theta+\rho), r\sin(\theta+\rho)) \\
Q_x &= r\cos\rho\cos\theta - r\sin\rho\sin\theta \\
Q_y &= r\sin\rho\cos\theta + r\cos\rho\sin\theta \\
Q_x &= P_x\cos\theta - P_y\sin\theta \\
Q_y &= P_x\sin\theta + P_y\cos\theta
\end{aligned}
$$

4. Scherung

$$
\begin{pmatrix} 1 & h & 0 \\ o & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}
$$

$$
\begin{aligned}
Q_x &= P_x + hP_y \\
Q_y &= P_y
\end{aligned}
$$

Scherung in Richtung x-Achse



Scherung in Richtung y-Achse

$detM \neq 0 \Rightarrow$ Umkehrabbildung existiert

$$Q = MP \Rightarrow P = M'Q$$

Umkehrabbildungen:

1. Translation

$$M^{-1} = \begin{pmatrix} 1 & 0 & -m_x \\ 0 & 1 & -m_y \\ 0 & 0 & 1 \end{pmatrix}$$

2. Skalierung

$$M^{-1} = \begin{pmatrix} \frac{1}{s_x} & 0 & 0 \\ 0 & \frac{1}{s_y} & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

3. Drehung

$$M^{-1} = \begin{pmatrix} \cos(-\theta) & -\sin(-\theta) & 0 \\ \sin(-\theta) & \cos(-\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} \cos\theta & \sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

4. Scherung

$$M^{-1} = \begin{pmatrix} 1 & -h & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

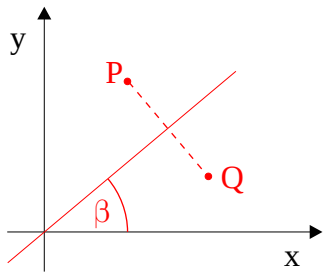
Zusammensetzung affiner Transformationen, z.B. Rotation T_R, Skalierung T_S, Translation T_T bilde $T_T T_S T_R$ (Reihenfolge beachten!!)

Beispiele:

1. Rotation um $v = (v_x, v_y)$
   - verschiebe P um -v
   - drehe um θ
   - verschiebe zurück um v

$$\begin{pmatrix} 1 & 0 & -v_x \\ 0 & 1 & -v_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & v_x \\ 0 & 1 & v_y \\ 0 & 0 & 1 \end{pmatrix}$$

2. Spiegelung an Gerade durch Ursprung



Schritte zur Realisierung:

- Drehung um $-\beta$: Spiegelachse wird x-Achse
- Spiegelung an x-Achse
- Zurückdrehung um β

$$\begin{pmatrix} \cos\beta & \sin\beta & 0 \\ \sin\beta & \cos\beta & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos\beta & -\sin\beta & 0 \\ \sin\beta & \cos\beta & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} \cos 2\beta & \sin 2\beta & 0 \\ \sin 2\beta & -\cos 2\beta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

6.2 3D-affine Transformationen

Analog zu 2D-affinen Transformationen

$$M = \begin{pmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\overline{M} = \begin{pmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{pmatrix}$$

$$detM = det\overline{M} \neq 0$$

elementare 3D-affine Transformationen

1. Translation

$$M = \begin{pmatrix} 1 & 0 & 0 & m_{14} \\ 0 & 1 & 0 & m_{24} \\ 0 & 0 & 1 & m_{34} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

2. Skalierung

$$M = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$s_x s_y s_z \neq 0$

3. Scherung

$$M = \begin{pmatrix} 1 & 0 & 0 & 0 \\ f & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

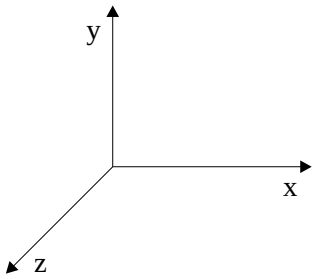
4. Drehung

   größere Variationsbreite als in 2D

   elementare Drehungen um Koordinatenachsen jeweils um Winkel β

   Abkürzung: $c = \cos\beta$ $s = \sin\beta$

   Skizze:



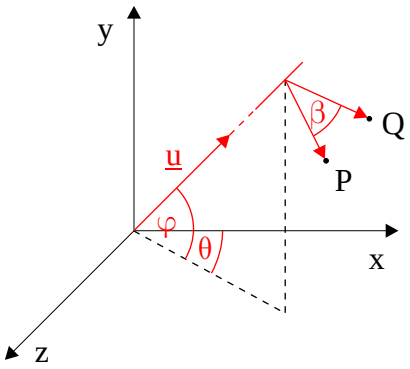
$$R_x(\beta) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & c & -s & 0 \\ 0 & s & c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$R_y(\beta) = \begin{pmatrix} c & 0 & s & 0 \\ 0 & 1 & 0 & 0 \\ -s & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$R_z(\beta) = \begin{pmatrix} c & -s & 0 & 0 \\ s & c & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Realisierung einer Drehung um Winkel β um die Achse $\underline{u} = (u_x, u_y, u_z)$ mit $|\underline{u}| = 1$

Ziel: Bestimmung der Drehmatrix



3 Schritte:

a) Führe zwei Drehungen aus, sodass u in Richtung der x-Achse zeigt. Benutze hierzu die Winkel ρ und θ.

b) Führe eine Drehung um die x-Achse um den Winkel β durch.

c) Man mache die Drehungen aus a) rückgängig.

Realisierung

Drehungen: $R_y(\theta), R_z(-\rho), R_x(\beta)$ und die Umkehrungen: $R_y(\theta)^{-1}, R_z(-\rho)^{-1}$

Ergebnis:

$$R_{\underline{u}}(\beta) = R_y(-\theta)R_z(\rho)R_x(\beta)R_z(-\rho)R_y(\theta)$$

$$\cos\rho = \frac{\sqrt{u_x^2 + u_y^2}}{\sqrt{u_x^2 + u_y^2 + u_z^2}}$$
$$\cos\theta = \frac{u_x}{\sqrt{u_x^2 + u_y^2}}$$

Vorteil: leicht implementierbar

Nachteil: unanschaulich

Alternative bietet OpenGL

Übergabe `glRotate(float β,float ux,float uy, float uz)`

Matrix:

M siehe Folie

$c = \cos\beta$, $s = \sin\beta$

$\underline{u} = (u_x, u_y, u_z)$

$|\underline{u}| = \sqrt{u_x^2, u_y^2, u_z^2} = 1$

Bemerkung: Eulerscher Satz

Jede orientierungserhaltende, metrische lineare Abbildung des $\mathbb{R}^3$, die den Ursprung 0 fest lässt, kann als $R_{\underline{u}}(\beta)$ realisiert werden. Hierbei bezeichnet $R_{\underline{u}}(\beta)$ die Rotation um eine Achse $\underline{u}, |\underline{u}| = 1$ um den Winkel β.

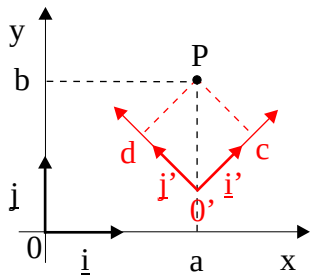
Zusammenfassung:

1. affine Transformationen erhalten affine Linearkombinationen
2. affine Transformationen erhalten Geraden und Ebenen
3. Streckenverhältnisse bleiben ungeändert

6.3 Wechsel von Koordinatensystemen

Aufgabe: Betrachte Koordinaten eines Punktes P in verschiedenen Koordinatensystemen

Beispiel:



T affine Transformation

$$T(0) = 0'$$
$$T(\underline{i}) = \underline{i}'$$
$$T(\underline{j}) = \underline{j}'$$

Einsetzen

$$\begin{pmatrix} a \\ b \\ 1 \end{pmatrix} = M \begin{pmatrix} c \\ d \\ 1 \end{pmatrix}$$
$$\begin{pmatrix} c \\ d \\ 1 \end{pmatrix} = M^{-1} \begin{pmatrix} a \\ b \\ 1 \end{pmatrix}$$

Ergebnis:

Angenommen das Koordinatensystem 2 geht aus dem Koordinatensystem 1 durch die Transformationsmatrix M und $P = \begin{pmatrix} p_x \\ p_y \\ 1 \end{pmatrix}$ sind die Koordinaten im System 2, dann hat P die Koordinaten MP im System 1.

Zusammensetzung:

$$\begin{pmatrix} a \\ b \\ 1 \end{pmatrix} = M_1 \begin{pmatrix} c \\ d \\ 1 \end{pmatrix} = M_1 \left(M_2 \begin{pmatrix} e \\ f \\ 1 \end{pmatrix} \right) = P_1 M_2 \begin{pmatrix} e \\ f \\ 1 \end{pmatrix}$$

Achtung:

affine Transformationen

Folge von affinen Transformationen T_1, T_2, T_3 wird auf P angewendet

bilde $M_3 M_2 M_1$ „Vormultiplikation“

P wird transformiert auf $Q = M_3 M_2 M_1 P$

affiner Koordinatenwechsel

bilde $M_1 M_2 M_3$ „Nachmultiplikation“

ein Punkt P mit den Koordinaten im transf. System hat im ursprünglichen Sysstem die Koordinaten $M_1 M_2 M_3 P$

Hinweis: OpenGL

OpenGL besitzt Werkzeuge für sukzessive Anwendungen von Transformationen um CT = current transformation zu bilden

OpenGL benutzt die „Nachmultiplikation“, d.h. für OpenGL ist es natürlich in tranformierten Koordinatensystemen zu rechnen

6.4 Benutzen von affinen Transformationen

Ziel: Arbeit mit OpenGL für Programmierung

Beispiel: zeichne Haus, rotiere um -30° und verschiebe um (32,25)

Realisierung:

1. schwieriger Weg

   Array v[] von Punkten, verschiedene Aufrufe von `moveTo(),lineTo()`, bilde Matrix M, bilde Routine `transform2D()`, transformiere jeden Punkt v[i] in `house()`, Befehle `cvs.moveTo(transform2D(M,v[0],...)`

   Voraussetzung: source code von `house()`

2. leichter Zugang

   Transformation soll nicht auf jeden Punkt angewendet werden

   Idee:

   Window-Viewport-Transformation wendet Transformation als Teil von `moveTo(), lineTo()` an $\Rightarrow$ Interpretation in die Graphikpipeline

   Befehle so gestalten, dass CT auf Argumente angewendet wird, Graphik-Pipeline modifizieren

   Aufruf: `glVertex2d()` mit Argument V

   - V wird mit CT in Q transformiert
   - Q durchläuft Window-Viewport-Transformation und S als Bildpunkt entsteht

   Vorteil: OpenGL transformiert automatisch model view matrix

   Spezifika für 3D:

   2D-Mechanismus: setze z=0, d.h. 2D-Graphik ist Spezialfall von 3D

   Routinen für `glRotated(),glScaled(),glTranslated()`

   Routine kreiert Matrix M und multipliziert $CT = CT \cdot M$, Reihenfolge wichtig

   Vorgehensweise:

   a) bilde die Transformationsmatrix M
   b) Ausführen $M \cdot M$
   c) Anwenden auf die Koordinatensysteme

   Hinweise:

   a) Initialisierung: `glLoadIdentity()`
   b) Änderung: `glMatrixMode(GL_MODELVIEW)`

   Abspeichern von CT für spätere Anwendung

   Folge von Drehungen, Skalierungen und Translationen bewirken Änderungen der CT

Rückgriff auf frühere CT

Stack von Transformationen

Speichern der aktuellen CT=Kopie der CT durch `pushCT()`

$\Rightarrow$ bewirkt 2 identische Kopien auf der Spitze des Stacks

top kann weiter verarbeitet werden

Rückgriff `popCT()`

Beispiel:

a) Folie

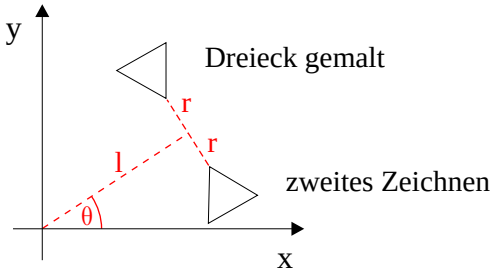
b) Tabelle:

Code	**Wirkung**
`glMatrixMode(GL_MODELVIEW);`	Änderung
`glLoadIdentity();`	$M = I_4$
`glRotate(`θ`);`	$M = MR_\theta$
`glTranslate(l,0,0);`	$M = MT_{(l,0,0)}$
`glPushMatrix();`	Speicherung M
`glTranslatef(0,r,0);`	$M = mT_{(0,r,0)}$
`drawThreePoints();`	Zeichne
`glPopMatrix();`	Lade M (oben gespeichert)
`glRotatef(`180°`);`	$M = MR_{180^\circ}$
`glTranslate(0,r,0);`	$M = MT_{(0,r,0)}$
`drawThreePoints();`	Zeichne

Ergebnis:

erstes Zeichnen: $M = (I_4) \circ T_{(l,0,0)} \circ T_{(0,r,0)}$

zweites Zeichnen: M=$R_\theta \circ T_{(l,0,0)} \circ R_{180^\circ} \circ T_{(0,r,0)}$



Übersicht über Transformationen

T Transformation mit

$$M = \begin{pmatrix} & \overline{M} & & p_x \\ & & & p_y \\ & & & p_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$detM = det\overline{M}$$

Typ	Beschreibung	Invarianten:			
		Länge	Winkel	||	Ebene
metrisch	$\overline{M}\overline{M}^T = I_3$	x	x	x	x
ähnlich	$\overline{M}\overline{M}^T = K^2 I_3$		x	x	x
affine	$det\overline{M} \neq 0$			x	x
projektiv					x

affine Abbildungen:

Transformation, uniforme Skalierung, Scherung, Drehung

6.5 3D-Szenen mit OpenGL

OpenGL unterstützt 3D-Darstellung

Kamera-Blick, Schnappschuss der Szene

Überblick: 2D-Darstellung = Spezialfall der 3D-Darstellung

1. Definition einer Viewing-Umgebung

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
gluLookAt(eye,center,up);
```

eye: Augpunkt

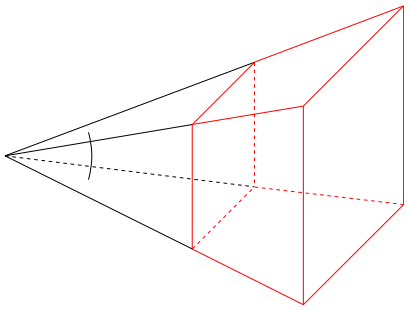
center: Blickpunkt

up: Blickrichtung

(tutors.zip!)

2. Definition einer perspektivischen Projektion

```
GL_PROJECTION
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(winkel,aspect,near,far);
```



Alternative:

`glFrustum(l,r,b,t,u,f)`

definiert die perspektivischen Projektion in Abhängigkeit vom Viewing-Volumen

l=left, r=right, b=bottom, t=top, n=near, f=far

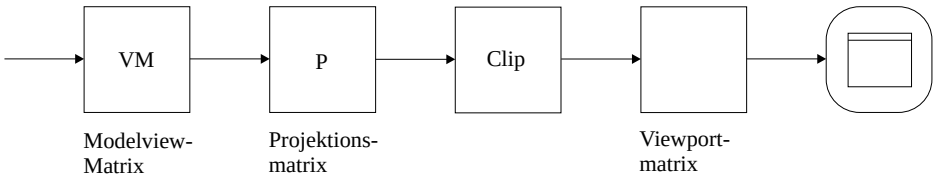
3. Definition orthografische Projektion

   `glOrtho(l,r,b,t,u,f)`

   Einbeziehen der Projektion in Graphikpipeline

   Hinweis: perspektivische Projektion natürlicher (wird genauer behandelt!)

   Graphik-Pipeline:



OpenGL-Tools:

1. Modelltransformation

```
glScaled(sx,sy,sz)
glTranslated(dx,dy,dz)
glRotated(angle,ux,uy,uz)
```

2. Kameraposition

```
glOrtho
gluLookAt
```

Setzen der Projektionsmatrix

```
glMatrixMode(GL_PROJECTION); //aktualisieren der Projektionsmatrix
glLoadIdentity();
glOrtho();
glPerspective();
```

3. Tiefenbuffer

```
glutInitDisplayMode();
glEnable(GL_DEPTH_TEST);
```

Auffrischen nach Rendern

```
glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
```

4. Verdecken von Front- oder Rückseiten

```
glCullFace(GL_FRONT,GL_BACK,GL_FRONT_AND_BACK);
glEnable(GL_CULL_FACE);
glDisable();
```

5. Zeichnen von Polygonen

default: massiv, ausgefüllt

$$\texttt{glPolygonMode(}\left\{\begin{array}{l} GL\_FRONT \\ GL\_BACK \\ GL\_FRONT\_AND\_BACK \end{array}\right\}\left\{\begin{array}{l} GL\_FILL \\ GL\_LINE \\ GL\_POINT \end{array}\right\}\texttt{)}$$

6. Updaten einer Szene

```
glutPostRedisplay();
```

Aktualisierung

a) `glutDisplayFunc()`
b) `glutIdleFunc()`

7. Ausgabe der Werte einer Matrix

```
GLfloat mat[16]; //definiert Matrix
glGetFloatv(GL_MODELVIEWMATRIX,mat);
    //kopiert Modelviewmatrix nach mat[],M[i][j] wird
    //in Element mat[4j+i], i,j=0,1,2,3 abgelegt
```

entsprechend Übergabe der Matrix

8. Zeichnen von elementaren 3D-Objekten

   GLUT liefert verschiedene vorgefertigte Objekte: Kugel, Kegel, Torus, Platonische Körper

   - `glutWireCube(GLdouble size);`
   - `glutWireSphere(GLdouble radius,GLint nSlices,GLint nStacks);`

     Slices: „Längenkreise“

     Stacks: „Breitenkreise“
   - `glutWireTorus(GLdouble inRad,GLdouble outRad,GLint nSlices,GLint nStacks);`
   - `glutWireTeapot(GLdouble size);`

   entsprechend `glutSolidSphere()`usw.

   - `glutWireDodecahedron()`

   siehe Kopie!!

Zusammenfassung:

- affine Transformation als Werkzeug der Computergraphik, erlauben Manipulationen von Szenen, Arbeit mit homog. Koordinaten
- 3D-Szenen erfordern komplexere Operationen mit Matrizen, Drehungen im Raum um beliebige Achsen
- OpenGL unterstützt affine Transformationen als Bestandteil der Graphik-Pipeline
- OpenGL erlaubt Ablage von Transformationen auf Stacks, die unabh. Verwaltung erlaubt
- Modell der Kamera zum Blick auf eine Szene wird auf OpenGL übertragen

Kapitel 7

Modellierung mit polygonalen Netzen

Ziel:

- Entwicklungswerkzeug für Objekte in 3D
- Zeichnen von Gittermodellen mit polygonalen Netzen
- Darstellung unregelmäßiger Körper mit netzartigen Objekten

Realisierung: Datenstruktur für Netze

Spezielle 3D-Objekte: Platonische Körper, Prismen, Rotationskörper usw.

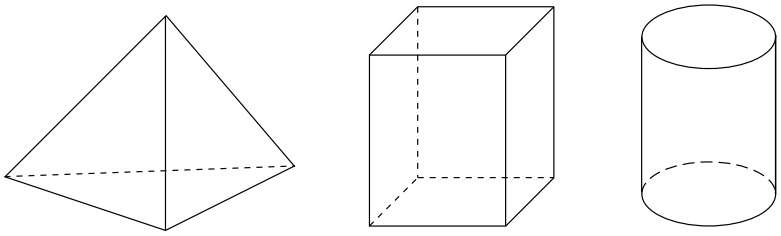
Bildung von 3D-Primitiven mit glatten Oberflächen $\Rightarrow$ Berechnen des Normalenvektors für jeden Punkt

7.1 Einführung zu polygonalen Netzen

Polygonales Netz:

Sammlung von Polygonen (=Seiten), die zusammen Oberfläche eines Objektes bilden

Beispiele:



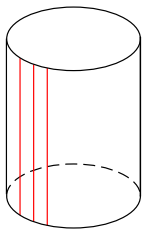
Bedeutung für Computergrafik:

- Einfachheit, basiert auf Polygonen
- einfaches Transformieren
- „einfaches“Zeichnen: Polygonroutine

- einfaches Berechnen der Normalenvektoren
- OpenGL: Rendern basiert auf dem Zeichnen von polygonalen Netzen

Beispiel:

1. Haus (siehe Folie)
   leicht zu rendern: exakte Kanten und Seitenflächen
2. Skizze:

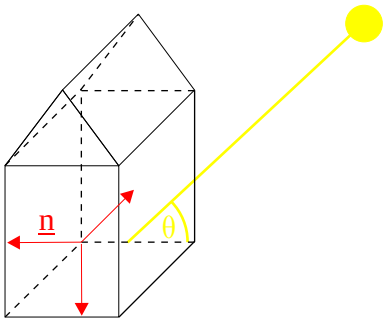


   schwer zu zeichnen, da Seitenflächen stetig ineinander übergehen
   benötigt: Techniken, die Übergänge glätten

Definition:

Polygonales Netz:

= Liste von Polygonen zusammen mit einer Richtung, von der das Polygon angesehen wird (entspricht dem Normalenvektor des Polygons)



Winkel θ

= Winkel zwischen Normale und Lichtquelle

entspricht einem Maß wieviel Licht von der Fläche gestreut wird (genauer: $\cos\theta$, wird noch behandelt)

Eckennormalen:

Zuordnung einer Eckennormale zu jedem Eckpunkt (erweist sich nützlicher als die Seitennormale beim Übergang von einer Seite zur anderen)

Realisierung polygonaler Netze:

Beispiel Folie

- verschiedene Möglichkeiten des Abspeichern im File
- z.B. Liste von 7 Polygonen, Liste für jedes Polygon mit Ecken und Normalen
- Alternative: 3 separate Listen: Ecken-, Normalen-, Seitenliste

Eckenliste: gibt Lokalisierung der Ecke im Netz

Normalenliste: enthält Normalenvektoren der vorkommenden Seiten

Seitenliste: Zeiger in beide anderen Listen

Zusammenspiel:

Eckenliste: geom. Informationen

Normalenliste: Inf. zur Orientierung

Seitenliste: Zusammenhang und topologische Information

(Normalenvektoren: normiert!!)

Bemerkungen zum Beispiel (siehe Folie):

- Ecken 0-9
- Normalen 0-6
- Normalen sind normiert
- Flächenliste mit Zeigern definiert Haus
- jede Ecke erhält die Normalenrichtung der Seite
- Eckenliste ist zyklisch geschlossen

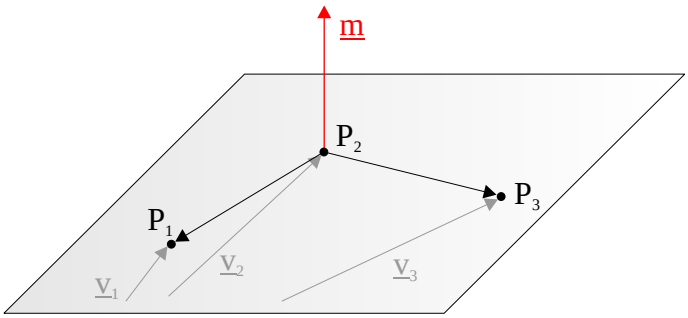
Festlegung:

Umlauf der Polygone entgegen dem Uhrzeigersinn (math. positive Richtung) entspricht der Sicht von außen

Bemerkung: Alternativ werden algorithmisch generierte Polygonnetze benutzt

Konstruktion der Normalenvektoren

Computergestützte Berechnung der Normalenebene Seitenfläche



$\underline{v}_1, \underline{v}_2, \underline{v}_3$ Ortsvektoren 3 aufeinanderfolgender Punkte

$$\underline{m} = (\underline{v}_1 - \underline{v}_2) \times (\underline{v}_3 - \underline{v}_2)$$

Ordne $\underline{m}$ jedem Punkt zu

Probleme:

1. $\underline{v}_1 - \underline{v}_2, \underline{v}_3 - \underline{v}_2$ sind annähernd parallel, d.h. $\underline{m}$ sehr klein
   $\Rightarrow$ num. Instabilitäten
2. Polygon ist nicht exakt planar, d.h. Polygon ungenau abgelegt
   $\Rightarrow$ Durchschnittsbildung anstreben

Lösung: Methode von Newell $\underline{m} = (m_x, m_y, m_z)$ wird wie folgt berechnet

N = Anzahl der Ecken des Polygons

$$m_x = \sum_{i=0}^{N-1} (y_i - y_{next(i)})(z_i + z_{next(i)})$$
$$m_y = \sum_{i=0}^{N-1} (z_i - z_{next(i)})(x_i + x_{next(i)})$$
$$m_z = \sum_{i=0}^{N-1} (x_i - x_{next(i)})(y_i + y_{next(i)})$$

(x,y,z) Koordinaten der i-ten Ecke

$$next(i) = (i+1) mod N$$

$\underline{m}$ ist nach außen gerichtet!

Hinweis: N=3

$$\underline{m} = (x_0 - x_1, y_0 - y_1, z_0 - z_1) \times (x_2 - x_1, y_2 - y_1, z_2 - z_1)$$
$$m_x = (y_0 - y_1)(z_2 - z_1) - (y_2 - y_1)(z_0 - z_1) = (y_2 - y_1)z_0 + (y_0 - y_2)z_1 + (y_1 - y_0)z_2$$

Newell:

$$m_x = (y_0 - y_1)(z_0 + z_1) + (y_1 - y_2)(z_1 + z_2) + (y_2 - y_0)(z_2 + z_0) = (y_2 - y_1)z_0 + (y_0 - y_2)z_1 + (y_1 - y_0)z_2$$

d.h. für N=2 stimmen beide überein, für N>2 addiert man noch ein Vielfaches des Normalenvektors

Vorteil der Methode von Newell:

Berechnung erfordert nur eine Multiplikation je Ecke für jede der Komponenten von $\underline{m}$, kein Test auf Kollinearität

Eigenschaften polygonaler Netze:

- Festkörpereigenschaft

  Polygonnetz umschließt kompaktes, beschränktes Raumgebiet

- Zusammenhang

  je zwei Punkte im Netz können durch einen Weg verbunden werden

- Einfachheit

  Polygonnetz heißt einfach, falls es keine Löcher enthält, d.h. wenn es stetig in eine Kugel deformiert werden kann

- Planarität

  Polygonnetz heißt planar, falls jedes Polygon eben ist (Polygonnetze aus Dreiecken sind planar)

- Konvexität

  die Verbindungsstrecke von je zwei Punkten gehört zum Körper

Programmieren mit Polygonnetzen

Ziele:

- effiziente Verfahren zur Integration von Netzen in Programmen
- Zeichnen eines Netzes
- Abspeichern eines Netzes in File

Realisierung:

Klasse Mesh

Hilfsklassen Vertex, Face, Point3, Vector3

Bemerkungen zur Folie:

Objekt Mesh

Vertexliste, Normalenliste, Flächenliste

repräsentiert durch Arrays pl, norm, face

dyn. Alloc.

Längen abgespeichert in numVerts, numNormals, numFaces

zusätzliche Datenfelder hinzufügen

Materialkonst. usw.

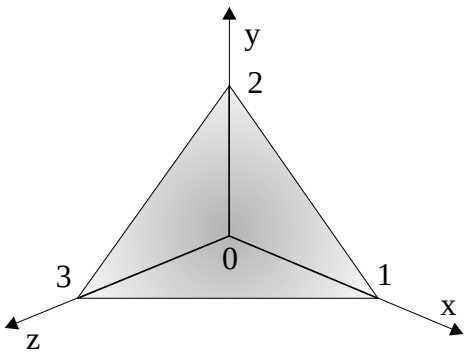
Face=Fläche

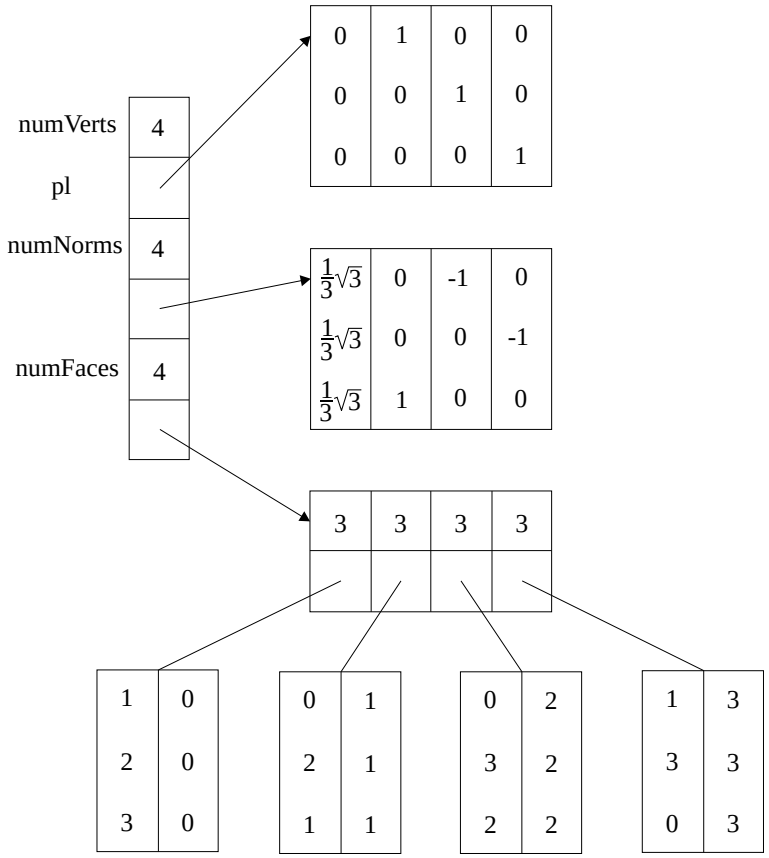
Liste von Ecken mit Pointern
organisiert als Liste von Indexpaaren v-te Ecke, f-te Ecke
pt[face[f],ver[v],vertIndex]

Hinweis:

geordneter Zugriff, leicht handhabbar, schneller Zugriff auf pl-Array

Beispiel: Tetraeder mit Ecken (0,0,0),$\sqrt{3}$ (1,0,0), (0,1,0), (0,0,1)





Zeichnen des Objektes

`Mesh::Draw()` OpenGL-basiert

mit

- Einlesen des Arrays von Flächen
- für jede Fläche Liste der Ecken und Normalen in Graphik-Pipeline einbinden

OpenGL:

Spezifikation der Ecken mit Normalenvektor

```
glNormal3f(m.x,m.y,m.z);
```

Normalvektor ist normiert!

Alternativ:

`glEnable(GL_NORMALIZE)` in `init()` normalisiert alle Normalenvektoren

Programmcode:

```
for(jede Seite f des Netzes)
{
    glBegin(GL_POLYGON);
    for(jede Ecke v in der Flaeche f)
    {
        glNormal3f(Normale in Ecke v);
        glVertex3f(Position der Ecke v);
    }
    glEnd();
}
```

Folie: Zeichnen von polygonalen Netzen

Erzeugen spezieller Netze: Einlesen von vordefinierten Netzen aus Files

Idee zum Speichern von Netzen:

→ Mesh-Klasse zum Einlesen aus Files und Abspeichern in Files

- Format des Files

- Struktur des Files:

Ecken, # Normalen, # Flächen

Ecken als Tripel von floatingpoint-Zahlen (x_i, y_i, z_i) aufgelistet und mehrere Ecken in einer Zeile möglich

Normalenvektor (normiert) als Tripel von floatingpoint-Zahlen

jede Fläche wird nach folgendem Typ aufgelistet:

Ecken der gegebenen Seite

Liste der Indizes der Eckenliste für die gegebene Seite

Liste der Indizes der Normalenliste für die Ecken der betrachteten Seite

Beispiel: siehe Folie

Code-Fragment zum Einlesen

Folie: mit `Mesh::readFile(char fileName)`

Bemerkungen:

Routine öffnet File

Routine liest File ein mit Namen in Netzobjekt

Rückgabe 0, falls ok

Rückgabe $\neq 0$, falls Fehler, z.B. kein File

Abspeichern: analog

7.2 Polyeder

Ziel: Polygonnetze zur Darstellung von Polyedern

Definition: Polyeder = ein beschränktes, einfaches, zusammenhängendes, planares Polygonnetz

Bemerkungen:

1. Es gibt verschiedene Definitionen von Polyedern in der Mathematik

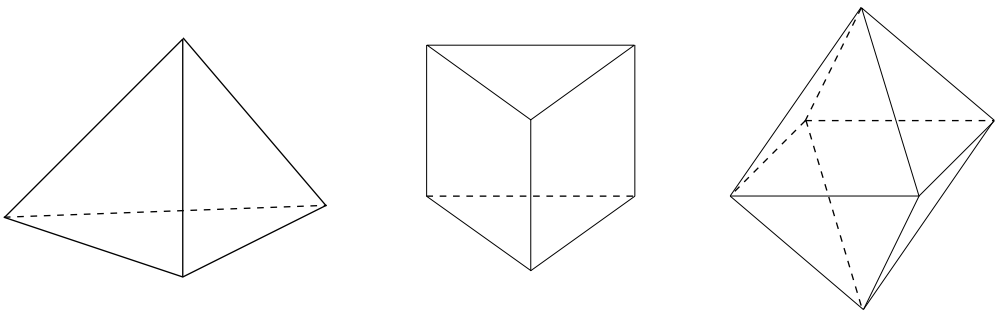
   a) konvexe Hülle endlich vieler Punkte

   b) Durchschnitt von endlich vielen Halbräumen

2. Polyeder hat folgende Eigenschaften

   - jede Kante wird von zwei Flächen begrenzt
   - wenigstens drei Kanten treffen sich in einer Ecke
   - zwei Seitenflächen sind entweder disjunkt bzw. haben keine Kante gemeinsam

Beispiele:



Eulersche Formel:

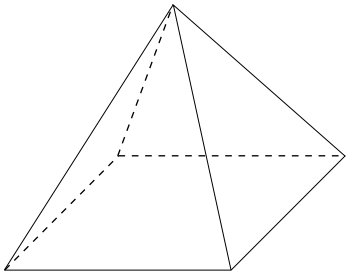
fundamentale Beziehung zwischen

v=# Ecken

e=# Kanten

f=# Flächen

$$v - e + f = 2$$



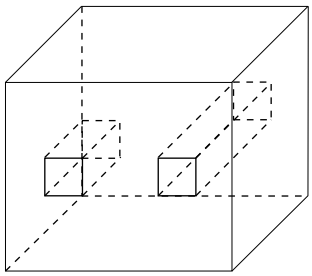
v=5
e=8
f=5

Verallgemeinerung:

$$v - e + f = 2 + h - 2g$$

h=# Löcher von Seiten

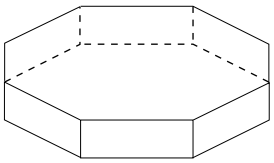
g=# Löcher des Objektes



v=24
e=36
f=15
h=3
g=1

Beipiel:

1. Prismen



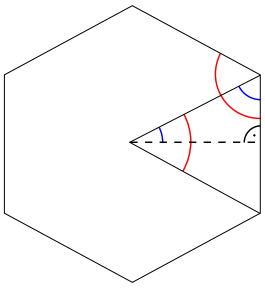
2. Platonische Körper

Definition:

Ein Polyeder heißt regulär, falls die Seitenflächen kongruente reguläre Polygone sind und in jedem Eckpunkt die selbe Anzahl dieser Polygone zusammenstoßen

z.B. Würfel: Seiten = kongruente Quadrate, in jeder Ecke = 3 Seiten

Überlegung: reguläre n-Ecke, k davon bilden eine Ecke, n≥3



Berechnen des Innenwinkels:

$$2 \cdot \left(\frac{\pi}{2} - \frac{\pi}{n}\right) = \pi\left(1 - \frac{2}{n}\right)$$

Es gilt: k≥3 und $\pi\left(1-\frac{2}{n}\right) < \frac{2\pi}{2}$, d.h.

$$1 < 2\left(\frac{1}{k} - \frac{1}{n}\right) \Leftrightarrow (k-2)(n-2) < 4$$

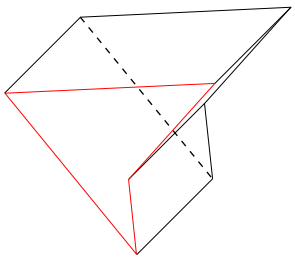
Lösungen: (Bilder auf Folie)

n	**k**	**Körper**
3	3	Tetraeder
4	3	Würfel
3	4	Octaeder
5	3	Dodekaeder
3	5	Ikosaeder

7.3 Extrahieren

Idee: 2D-Strukturen werden in 3D ausgedehnt (extrahiert)

Beispiel: Grundfläche: Viereck als Grundfläche für Prisma



Beispiel: Bild auf Folie: Pfeil (Arrow)

- Pfeil P in (x,y)-Ebene soll in z-Richtung extrahiert werden
- Punkte (x_i, y_i) des Pfeils werden in den Raum eingebettet durch Hinzufügen von z $\Rightarrow$ $(x_i, y_i, 0)$ bilden Grundfläche
- Deckfläche entsteht aus $(x_i, y_i, H) \forall$ i
- Modell des extrahierten Pfeils:

  P Grundfläche

  Q Deckfläche

  7 Rechtecke als Seitenfläche

- Normalenvektoren:
  - Seitenflächen: jede Ecke erhält Normalenvektor, der ebenen Begrenzung ausgedehnt in 3D
    $(x_i, y_i) \perp (u_i, v_i) \Rightarrow N : (u_i, v_i, 0)$
  - Grundfläche: (0,0,-1)
  - Deckfläche: (0,0,1)

Aufgabe:

Generiere ein Prisma mit polygonaler Grundfläche mit N Punkten (x_i, y_i), i=0,...,N-1

Punkte im Raum: $(x_i, y_i, 0)$, i=0,...,N-1, (x_j, y_j, H), j=N+1,...,2N

Flächen: Seitenflächen j=0,...,N-1

$j, j+N, next(j)+N, next(j)$

$next(j) = (j+1) mod N$

P, Q Grund- und Deckfläche

Normalenvektoren:

P:(0,0,-1)

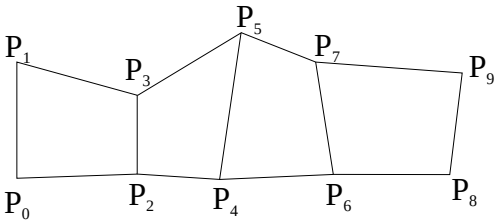
Q:(0,0,1)

Seitenfläche: Normalenvektor der ebenen Gerade

Problem: konkave bzw. überschlagene Polygone liefern u.U. Fehler

Zerlegung in konvexe Teilpolygone

Extrahieren von Quadstrips



Paare von Punkten

- geradzahlig: eine Kantenfolge
- ungeradzahlig: andere Kantenfolge

2M - Ecken

2M-2 - Seitenflächen

entsprechende Datenstrukturen

```
void Mesh::makeExtrudedQuadStrip(Point2 p[], int numpts, Vector3d)
```

→ Extrahieren verbunden mit einer Drehung oder einer Skalierung

$P = \{P_0, P_1, ..., P_{N-1}\}$

$P' = \{MP_0, MP_1, ..., MP_{N-1}\}$, M=Matrix

mit Matrizen

$$M = \begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = R_z(\theta)$$

oder

$$M = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

→ mögliche Scherung anfügen

Segmentiertes Extrahieren

Idee: Zusammensetzen einer Sequenz von Extraktionen

Grundlage: 3D-Kurven

anstelle individueller Matrizen benutze eine Raumkurve um die die Prismen gelegt werden

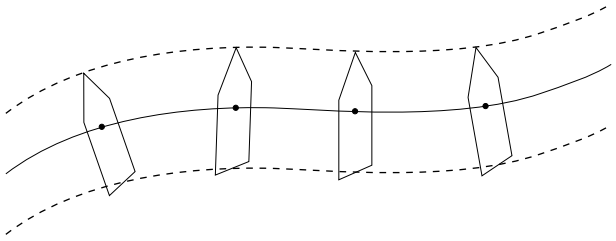
3D-Kurve = bildet Rückgrat der Röhre

Bsp.: Helix $c(t) = (\cos t, \sin t, bt)$

Vorgehensweise:

$c(t)$ wird gesampelt an den Werten $t_0, t_1, ...$

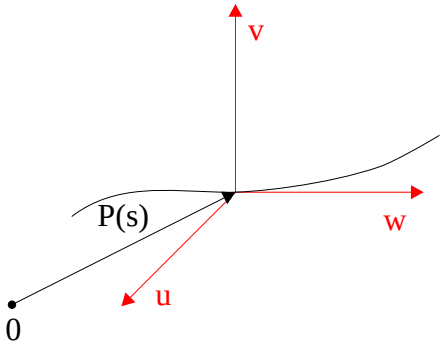
transformiertes Polygon wird in $c(t_i)$ eingehängt, und zwar so, dass das Polygon senkrecht zur Kurve hängt



7.4 Das Frenetsche Koordinatensystem

Ziel: Bewegung entlang einer Kurve, sodass Bewegungsrichtung tangential zur Kurve verläuft

Idee: Mitführen eines kartesischen Koordinatensystems, sodass eine Koordinatenrichtung in Richtung der Tangente zeigt



Verlauf der Kurve:

$$P(s) = \begin{pmatrix} x(s) \\ y(s) \\ z(s) \end{pmatrix}$$

parametrisiert durch s, Ortsvektor der Kurve

Geschwindigkeitsvektor:

$$\dot{P}(s) = \begin{pmatrix} \dot{x}(s) \\ \dot{y}(s) \\ \dot{z}(s) \end{pmatrix}$$

gibt Tangentenrichtung

Tangente der Kurve $P(s)$ in $s = s_0$: $P(s_0) + \dot{P}(s_0)$

Beschleunigung

$$\ddot{P}(s) = \begin{pmatrix} \ddot{x}(s) \\ \ddot{y}(s) \\ \ddot{z}(s) \end{pmatrix}$$

Konstruktion des Frenetschen Koordinatensystems:

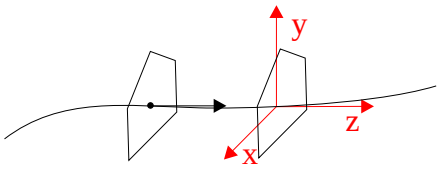
$$\underline{w} = \frac{1}{|\dot{P}(s)|} P(s)$$
$$\underline{u} = \frac{1}{|\dot{P}(s) \times \ddot{P}(s)|} \dot{P}(s) \times \ddot{P}(s)$$
$$\underline{v} = \underline{w} \times \underline{u}$$

$\{\underline{u}, \underline{v}, \underline{w}\}$ Frenetsches Koordinatensystem der Kurve P(s)

Problem: Argument klappt nicht, falls $\dot{P}(s)$, $P(s)$ parallel sind (Ausnahmefall!)

Anwendung:

c(t) Kurve wird an Werten $t_0, t_1, t_2, \ldots$ gesamplet, transformiertes Polygon wird in $c(t_i)$„eingehängt“



Zuordnung eines lokalen Koordinatensystems

Polygon liegt in x,y-Ebene

Normale in Richtung z-Achse

Frenetsches Koordinatensystem

t_i Wert

$$\left.\begin{array}{l} T(t_i) \\ N(t_i) \\ B(t_i) \end{array}\right\} \text{Frenetsches System in } t_i$$

($T(t_i)$ Tangente in t_i)

Ziel: Transformationsmatrix, die Basispolygon des Schlauches an Stelle c(t_i) transformiert

M_i transformiert $\underline{i}, \underline{j}, \underline{k}$ in $N(t_i), B(t_i), T(t_i)$

Ergebnis:

$$M_i = (N(t_i)|B(t_i)|T(t_i)|c(t_i))$$

mit

$$\begin{aligned} c(t) &= (c_x(t), c_y(t), c_z(t)) \\ T(t) &= \frac{1}{|\dot{c}(t)|}\dot{c}(t) \times \ddot{c}(t) \\ B(t) &= \frac{1}{|\dot{c}(t) \times \ddot{c}(t)|}\dot{c}(t) \times \ddot{c}(t) \\ N(t) &= B(t) \times T(t) \end{aligned}$$

Beispiel: Helix

$$\begin{aligned} c(t) &= (\cos t, \sin t, bt) \\ \dot{c}(t) &= (-\sin t, \cos t, b) \\ T(t) &= \frac{1}{\sqrt{b^2+1}} \\ \ddot{c}(t) &= (-\cos t, -\sin t, 0) \\ B(t) &= \frac{1}{\sqrt{b^2+1}}(b\sin t, -b\cos t, 1) \\ N(t) &= (-\cos t, -\sin t, 0) \end{aligned}$$

Matrix

$$M_i = \begin{pmatrix} -\cos t_i & \frac{b}{\sqrt{b^2+1}}\sin t_i & -\frac{1}{\sqrt{b^2+1}}\sin t & \cos t_i \\ -\sin t_i & -\frac{b}{\sqrt{b^2+1}}\cos t_i & \frac{1}{\sqrt{b^2+1}}\cos t & \sin t_i \\ 0 & \frac{1}{\sqrt{b^2+1}} & \frac{b}{\sqrt{b^2+1}} & bt_i \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Beispiel: Zehneck mit 30 Punkten auf Helix

Variation

Skalierungsmatrix

$$M' = M \begin{pmatrix} g(t) & 0 & 0 & 0 \\ 0 & g(t) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

hier $g(t) = t$

Beispiel: Knoten und Schlingen, Ausgangspunkt toroidale Spirale

$$c(t) = ((a + b\cos(qt))cos(pt), (a + b\cos(qt))\sin(pt), c\sin(qt))$$

a, b, c, p, q Parameter

p=2 q=5 a)
p=1 q=7 b) } siehe Folie

Rotationsflächen

Basispolygon = Profil, wird um eine Achse rotiert

Spezialfall der vorhergehenden Betrachtung: $c(t)$ = Kreislinie

Beispiel: Weinglas

Profil ist Polygon $P_j = (x_j, y_j, 0)$

Transformationsmatrix

$$M_i = \begin{pmatrix} \cos\theta_i & 0 & \sin\theta_i & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta_i & 0 & \cos\theta_i & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\theta_i = \frac{2\pi i}{k}, i = 0, ..., k-1$$

Rotation setzt $(x_j, y_j, 0)$ auf $(x_j \cos\theta_i, y_j, x_j \sin\theta_i)$, i=0,...,k-1

7.5 Gitterapproximation von glatten Flächen

bisher: Polyederdarstellung durch Netze, Nachteil: datenintensive Strukturen

Ziel: Netze per Modellierung glatter Oberflächen (Kugel, Torus usw.)

glatte Oberflächen: definiert durch Formeln

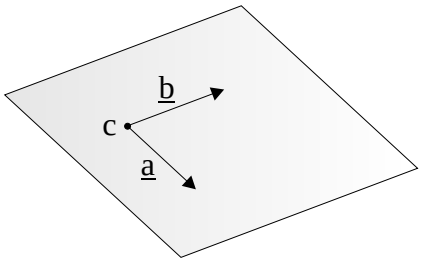
zusätzliche Info: Berechnung der Normalenvektoren (Gourand-Schattierung)

Idee: Polygonalisieren, Zerlegung in Polygone

Darstellung von Flächen:

1. Ebene

$$P(u,v) = c + \underline{a}u + \underline{b}v, a, b \in \mathbb{R}$$



Parameterform

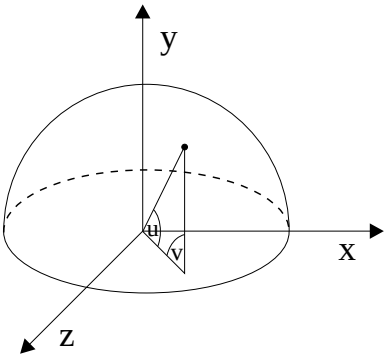
$$P(u,v) = (x(u,v), y(u,v), z(u,v))$$

hier: x, y, z lineare Funktionen in u, v

2. Allg. Parameterform einer Fläche

Kugel $M = (0,0,0)$, Radius r

$$x = r\cos u\cos v$$
$$y = r\sin u\cos v$$
$$z = r\sin v$$



3. Allg. implizite Form

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

z.B. Ebene:

$$ax + by + cz + d = 0$$

z.B. Kugel:

$$x^2 + y^2 + z^2 - r^2 = 0$$

Bemerkung:

geschlossene Fläche:

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

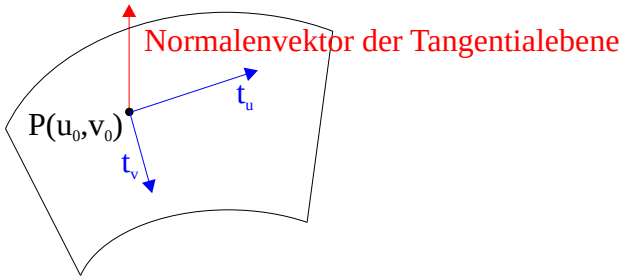
Punkt:

$$P_0 = (x_0, y_0, z_0) \in \mathbb{R}$$

$$F(x_0,y_0,z_0) = \begin{cases} < 0 & : \text{ innerhalb} \\ = 0 & : \text{ auf} \\ > 0 & : \text{ außerhalb der Fläche} \end{cases}$$

Normalenvektoren von Flächen

1. $P(u,v)$ in Parameterform



$$\underline{n}(u_0,v_0) = \left(\frac{\partial P}{\partial u} \times \frac{\partial P}{\partial v} \right) \Big|_{u=u_0, v=v_0}$$

Beispiel: Ebene

$$P(u,v) = c + \underline{a}u + \underline{b}v$$
$$\frac{\partial P}{\partial u} = \underline{a}, \frac{\partial P}{\partial v} = \underline{b}$$
$$\underline{n}(u,v) = \underline{a} \times \underline{b}$$

2. $F(x,y,z)$ implizite Fläche

$$\underline{n}(x_0,y_0,z_0) = \nabla F|_{x=x_0,y=y_0,z=z_0}$$

Beispiel:

$$F(x,y,z) = ax+by+cz+d = 0$$

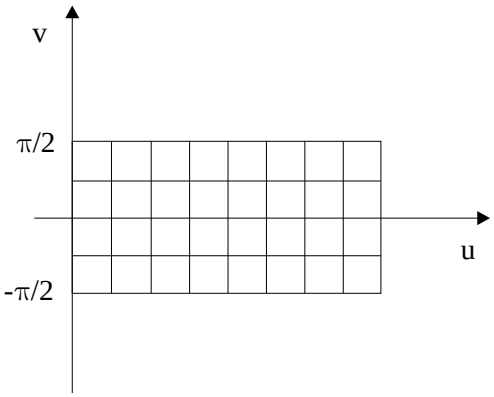
Normalenvektor:

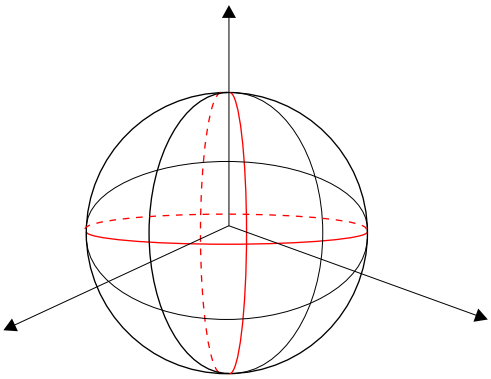
$$\nabla F = (a,b,c)$$

Zeichnen von Kugeln, Zylindern und Kegeln

1. Kugel

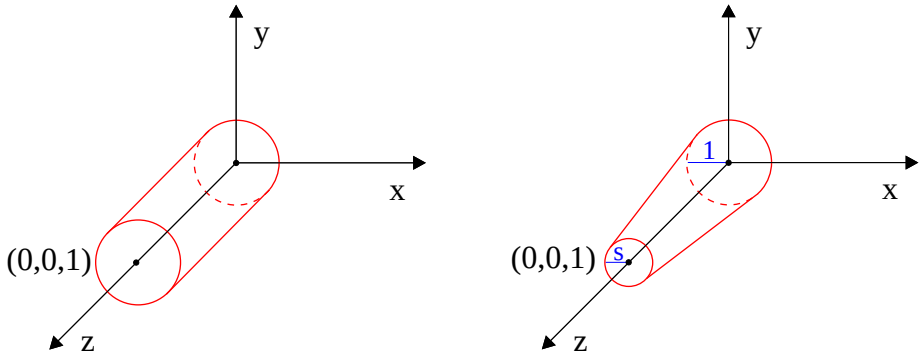
$$F(x,y,z) = x^2+y^2+z^2-1$$
$$P(u,v) = (\cos v\cos u, \cos v\sin u, \sin v)$$
$$u \in [0,2\pi], v \in [-\frac{\pi}{2},\frac{\pi}{2}]$$





Normalenvektor der Kugel in (x_0,y_0,z_0): $\underline{n} = (x_0,y_0,z_0)$

2. Zylinder, Kegelstumpf



implizite Form des Kegelstumpfes:

$$F(x,y,z) = x^2 + y^2 - (1+(s-1)z)^2, 0 \leq z \leq 1$$

s=1 Zylinder

$0 \leq s < 1$ Kegelstumpf

s=0 Kegel

Parameterdarstellung:

$$P(u,v) = ((1+(s-1)v)\cos u, (1+(s-1)v)\sin u, v)$$

Normalenvektor:

$$\nabla F = (2x, 2y, -(s-1)(1+(s-1)z))$$

Normalenvektoren:

Fläche	**P(u,v)**	**F(x,y,z)**
Kugel	P(u,v)	(x,y,z)
Kegelstumpf	(cosu,sinu,1-s)	(x,y,-(s-1)(1+(s-1)z))
Zylinder s=1	(cosu,sinu,0)	(x,y,0)
Kegel s=0	(cosu,sinu,1)	(x,y,1-z)

Erzeugen polygonaler Netze

Zerlegen in Scheiben und Stapel

n Slices um Äquator

n Stacks von Nord- zum Südpol

Slices:

$$u_i = \frac{2\pi i}{\text{n Slices}}, i = 0, \ldots, \text{n Slices} - 1$$

Stacks:

$$v_j = \frac{\frac{\pi}{2} - \pi j}{\text{n Stacks}}, j = 0, 1, ..., \text{n Stacks}$$

liefert n Streifen und (n+1) Stapel

Skizze: (Folie)

12 Streifen, 8 Stacks

Eckenliste

insgesamt 98 Punkte

Nordpol pt[0]

Normalenliste

norm[k]

Kugel norm[k]=pt[k]

Flächenliste

96 Flächen

24 Dreiecke, 72 Vierecke

# Ecken	3	3
Eckenindex	012	023
Normalenindex	012	023

Ziel: `makeSurfaceMesh()`

Allgemein:

Parameterform der Fläche:

$$p(u,v) = (x(u,v), y(u,v), z(u,v))$$

numValuesU, numValuesV

zwischen uMin, uMax bzw. vMin, vMax

Bemerkungen:

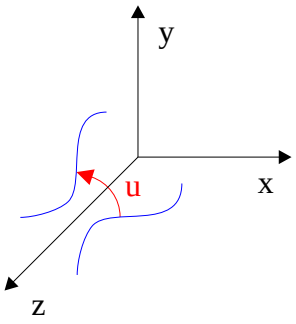
1. manche Grafikpakete sind für Dreiecke optimiert, d.h. Triangulierung der Fläche vornehmen
2. Kugel ist Spezialfall einer Rotationsfläche, Rotationsflächen lassen sich alternativ beschreiben

Rotationsflächen

Idee: Kurve $c(v) = (x(v), z(v))$ in der (x,z)-Ebene, rotiere Kurve um z-Achse

Rechnung: $c(v)$ in $\mathbb{R}^3$ eingebettet

$$c(v) = (x(v), 0, z(v))$$



$$R_z(u) = \begin{pmatrix} \cos u & -\sin u & 0 \\ \sin u & \cos u & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Zusammensetzung:

$$\begin{pmatrix} \cos u & -\sin u & 0 \\ \sin u & \cos u & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x(v) \\ 0 \\ z(v) \end{pmatrix}$$

Parameterdarstellung:

$$P(u,v) = (x(v)\cos u, x(v)\sin u, z(v))$$

Ermittlung des Normalenvektors

1. Methode:

   Bestimme den Normalenvektor der Kurve $c(v)$ in der (x,z)-Ebene

$$\dot{c}(v) = (\dot{x}(v), 0, \dot{z}(v))$$
$$\underline{n}(v) = (\dot{z}(v), 0, -\dot{x}(v))$$

$$\begin{pmatrix} \cos u & -\sin u & 0 \\ \sin u & \cos u & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \dot{z}(v) \\ 0 \\ -\dot{x}(v) \end{pmatrix}$$

$$\underline{n} = (\dot{z}(v)\cos u, \dot{z}(v)\sin u, -\dot{x}(v))$$

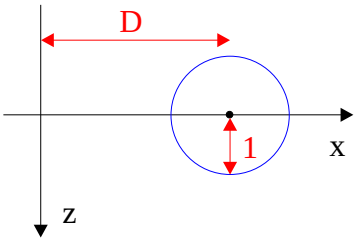
2. Methode: Vektorprodukt

$$\frac{\partial P}{\partial u} = (-x(v)\sin u, x(v)\cos u, 0)$$
$$\frac{\partial P}{\partial v} = (\dot{x}(v)\cos u, \dot{x}(v)\sin u, \dot{z}(v))$$

Kreuzprodukt:

$$\frac{\partial P}{\partial u} \times \frac{\partial P}{\partial v} = (\dot{z}(v)x(v)\cos u, \dot{z}(v)x(v)\sin u, -\dot{x}(v)x(v)) = x(v)(\dot{z}(v)\cos u, \dot{z}(v)\sin u, -\dot{x}(v))$$

Beispiel: Torus



Mittelpunkt (D,0,0), Radius A

Gleichung des Kreises:

$$c(v) = (D + A\cos v, 0, A\sin v)$$
$$P(u,v) = ((D + A\cos v)\cos u, (D + A\cos v)\sin u, A\sin v)$$

Netz des Torus:

$\{u_i\}$, $\{v_j\}$ Parameterliste

Punkte: $p(u_i, v_j)$

Normalen: $\underline{n}(u_i, v_j)$

Alternativ: Vorgabe von Punkten

$$c_i = (x_i, 0, z_i)$$
$$P_{ij} = (x_i \cos u_j, x_i \sin u_j, z_i)$$
$$u_j = \frac{2\pi i}{n}$$

Rotieren auf 3D-Kurven basierend

gegeben: $c(v)$ eine räumliche Kurve = Seele bzw. Rückgrat eines Schlauches

gesucht: Fläche des Schlauches mit Querschnitt eines Kreises

Lösung: Kreislinie $(\cos u, \sin u, 0)$ in (x,y)-Ebene, Mittelpunkt (0,0,0) und Radius 1

Transformationsmatrix

$$M = (N(v)|B(v)|T(v)|c(v))$$

homogene Matrix mit Frenetschen Koordinatensystem

Erinnerung:

$$\begin{aligned} &\dot{c}(v), \ddot{c}(v) \\ &T(v) = \frac{1}{|\dot{c}(v)|}\dot{c}(v) \\ &B(v) = \frac{1}{|\dot{c}(v) \times \ddot{c}(v)|}\dot{c}(v) \times \ddot{c}(v) \\ &N(v) = B(v) \times T(v) \end{aligned}$$

Transformation des Kreises

$$(N(v)|B(v)|T(v)|c(v)) \begin{pmatrix} \cos u \\ \sin u \\ 0 \\ 1 \end{pmatrix} = P(u,v)$$

$\rightarrow$ Parameterform des Schlauches

Ergebnis:

$$P(u,v) = c(v) + \cos u N(v) + \sin u B(v)$$

$\{u_i\}$, $\{v_j\}$ Unterteilung, Netz des Schlauches

Zusammenfassung:

- Polygonale Netze mit Datenstruktur zur Modellierung von 3D-Objekten
- Darstellung komplexer glatter Flächen durch Polygonnetze
- Behandlung der Normalenvektoren (Newell), Frenetsche Koordinaten
- Zusammenstellung einfacher Modellierungstechniken

Kapitel 8

3D-Sehen

Ziele:

- Entwicklung von Werkzeugen für die Handhabung einer Kamera
- Realisierung von Kamerafahrten durch Szenen
- Kennenlernen mathematischer Hintergründe für perspektivische Projektionen
- Realisierung in OpenGL und der OpenGL-Pipeline
- 3D-Clipping-Mechanismen

Hintergrund: Kamera-Modell liefert Grundlage für das 3D-Sehen in der Computergrafik

8.1 Das Kameramodell

Bisher: Parallelprojektion

Augpunkt: Position des Auges

view volume: Teil einer rechteckigen Pyramide deren Spitze das Auge ist

view angle: Öffnungswinkel der Pyramide

near plane, far plane: senkrechte Ebenen zur Pyramidenachse

schneiden aus Pyramide rechteckige Fenster heraus

aspect ratio: Verhältnis von Breite zur Höhe des Fensters

OpenGL: clippt Objekte, die außerhalb des view volume liegen

Projektion auf view plane

O.B.d.A. view plane = near plane

Bild des view plane: wird auf viewport transformiert und abgebildet

Setzen des view volume:

- Blickrichtung vom Auge = neg. z-Achse

- Form des view volumes wird in Projektionsmatrix abgelegt, die in Grafik-Pipeline eingeht

  `gluPerspective()` mit 4 Parametern

Vorgehensweise: (siehe auch Kopie)

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(viewAngle,aspectRatio,N,F);
```

Parameter:

viewAngle: θ wie in Skizze in Grad

aspectRatio: Verhältnis von Breite zu Höhe

N: Abstand des near plane vom Auge

F: Abstand des far plane vom Auge

Beispiel: `gluPerspective(60,1.5,0.3,50.0)`

near plane = -0.3

far plane = -50.0

Positionen einer Kamera (siehe Folie)

Ziel: Bewegen der Kamera, Augpunkt aus Normallage in vorgegebene Position

Vorgehensweise: Rotation, Translation wird Bestandteil der Modellview-Matrix

einfachste Funktion:

`gluLookAt()` in `glMatrixMode(GL_MODELVIEW)`

`gluLookAt(eye.x,eye.y,eye.z,look.x,look.y,look.z,up.x,up.y,up.z)`

Parameter:

eye: Koordinaten des Augpunktes
look: Koordinaten des interessierenden Punktes
up: Aufwärtsrichtung, default (0,1,0)

Ziel: genaue Analyse der Kamerabewegung, relative Änderung zu gegebener Position bei beliebiger Position und Bewegung der Kamera

Augkoordinatensystem: u, v, n-Achsen

Kamerakoordinatensystem: x, y, z-Achsen, die bewegt und rotiert werden

Aufgabe `gluLookAt()`

Begriffe:

pitch = richten, roll = verdrehen, yaw = gieren

Aufgabe:

eye, look, $\underline{up}$ gegeben

$\underline{n}$ parallel zu eye-look

$\underline{u}$, $\underline{v}$ senkrecht zu $\underline{n}$

$\underline{u}$ senkrecht zu $\underline{up}$

Lösung:

$$\underline{u} = \underline{up} \times \underline{n}$$
$$\underline{v} = \underline{n} \times \underline{u}$$

(Reihenfolge)

Ergebnis:

$$\underline{n} = eye - look$$
$$\underline{u} = \underline{up} \times \underline{n}$$
$$\underline{v} = \underline{n} \times \underline{u}$$

→ normiert, $\underline{up}$=(0,1,0) default

Frage: Welche Matrix übergibt `gluLookAt()` in Modellview-Matrix?

Sei V=Transformation von Weltkoordinaten in Kamerakoordinaten, M=Modelltransformation der Punkte

Modellview-Matrix=V·M

Also: `gluLookAt()` bildet V und multipliziert mit CM

Bestimmen der Matrix V

V transformiert Koordinatensystem des Augpunktes in Koordinatensystem des Ursprungs

$\underline{u}$, $\underline{v}$, $\underline{n}$ orthon. System

$$V = \begin{pmatrix} u_x & u_y & u_z & d_x \\ v_x & v_y & v_z & d_y \\ n_x & n_y & n_z & d_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} u_x \\ u_y \\ u_z \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

mit

$$\underline{u} = (u_x, u_y, u_z)$$
$$\underline{v} = (v_x, v_y, v_z)$$
$$\underline{n} = (n_x, n_y, n_z)$$
$$\begin{pmatrix} eye.x \\ eye.y \\ eye.z \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

und

$$(d_x, d_y, d_z) = (-eye \cdot \underline{u}, -eye \cdot \underline{v}, -eye \cdot \underline{n})$$

Projektion von eye

Matrix V leistet folgendes

$$\underline{u} \rightarrow \underline{i}$$
$$\underline{v} \rightarrow \underline{j}$$
$$\underline{n} \rightarrow \underline{k}$$
$$eye \rightarrow Ursprung$$

Kameras in Programmen

Überlegung: Klasse `camera`

```
cam.set(eye,look,up); //setzen der Kamera
cam.slide(-1,0,2);    //anheben
cam.roll(30);         //verdrehen
cam.yaw(20);          //aendern der Richtung
```

Routine in camera

`setModelViewMatrix()` (siehe Folie)

Bemerkungen:

- `set()` wirkt wie `gluLookAt()`

  übernimmt Parameter eye, look, $\underline{up}$

  berechnet

$$\underline{n} = eye - look$$
$$\underline{u} = \underline{up} \times \underline{n}$$
$$\underline{v} = \underline{n} \times \underline{u}$$

  (normiert)

übergibt Werte an OpenGL

- `setShape()`

  übergibt 4 Argumente an die entsprechende Kamera und ruft `gluPerspective(viewAngle,anspectRatio,N,F)` auf

- zentrale Funktionen: `slide(),roll(),yaw(),pitch()`

  Kameraflug:

  beim userdefinierten Flug mit Kamera durch die Szene, interaktiv durch Drücken von Tasten bzw. mit Menüs

  Freiheitsgrade=6

  3 Verschiebung, 3 Drehen

  einige Details:

  „sliding“: bewege Kamera in Kamerarichtung um (delu,delv,delw)

  „rotation“: Drehe Kamera um eigene $\underline{u}$-Achse

$$\underline{n}' = \cos\alpha\underline{u} + \sin\alpha\underline{v}$$
$$\underline{v}' = -\sin\alpha\underline{u} + \cos\alpha\underline{v}$$

  entsprechend: `pitch(),yaw()`

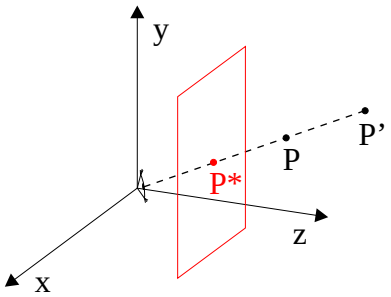
Zusammenfassung: Folie

- `myKeyboard` „slide“, „rotated“, z.B. Drücken Taste „p“: pitch um 1 Grad usw.

8.2 Die perspektivische Projektion

Ziel: Erweiterung der Grafik-Pipeline

Was ist Perspektive?



Ebene: z=-N

Probleme:

Wie projiziert man in view volume?

Wie geschieht Clippen?

Wie rechnet man mit homogenen Koordinaten?

Wie behandelt man die Tiefe eines verdeckten Punktes?

Perspektivische Projektion eines Punktes

N = Abstand vom Augpunkt

N=z Gleichung der Projektionsebene

Anwendung des Strahlensatzes

$$\frac{x^*}{P_x} = \frac{N}{-P_z}, \frac{y^*}{P_y} = \frac{N}{-P_z}$$

Ergebnis:

$$p^* = (x^*, y^*, -N)$$

mit

$$\left.\begin{aligned} x^* &= N\frac{P_x}{-P_z} \\ y^* &= N\frac{P_y}{-P_z} \end{aligned}\right\}(*)$$

Perspektivische Projektion von Geraden

Strahl:

$$\begin{aligned} &A + \underline{c}t \\ &A = (A_x, A_y, A_z) \\ &\underline{c} = (c_x, c_y, c_z) \end{aligned}$$

$P(T) = A + \underline{c}t$ ist Punkt der Geraden

Andwendung von (*): (Weglassen der z-Komponente)

$$p^*(t) = \left(N\frac{A_x + c_x t}{-A_z - c_z t}, N\frac{A_y + c_y t}{-A_z - c_z t} \right)$$

Strahl in der Ebene z=N

1. Spezialfall $A+\underline{c}t$ parallel zur Ebene: $c_z = 0$

$$p^*(t) = \frac{N}{-A_z}(A_x + c_x t, A_y + c_y t)$$

   Gerade mit dem Anstieg $\frac{c_y}{c_x}$

   Fakt: Zwei parallele Geraden, die parallel zur view plane sind, werden auf parallele Gerade abgebildet.

2. allgemein: $c_z \neq 0$, t wächst

$$P^*(\infty) = \left(N\frac{c_x}{-c_z}, N\frac{c_y}{-c_z}\right)$$

   Verschwindungspunkt der Geraden hängt nur von der Richtung (c_x, c_y, c_z) ab

   Fakt: Zwei parallele Geraden, die nicht parallel zur view plane sind, schneiden sich im selben Verschwindungspunkt.

Einarbeiten in Graphik-Pipeline

Welcher der Punkte P_1, P_2 ist näher zum Betrachter?

Idee: Einführen der Pseudotiefe

Punkt $P = (x,y,z)$ wird zugeordnet $p^* = (x^*, y^*, z^*)$, wobei x^*, y^* wie zuvor, z^* Zusatzinformation

Vorschlag:

$$p^* = \left(N\frac{P_x}{-P_z}, N\frac{P_y}{-P_z}, \frac{aP_z + b}{-P_z}\right)$$

für geeignete a,b

Wahl:

$$a = -\frac{F+N}{F-N}$$
$$b = -\frac{2FN}{F-N}$$

Größe:

$$\frac{aP_z + b}{-P_z}$$

heißt Pseudotiefe von P

Hierbei: N,F near und far plane

Bemerkung: P_z=-N

$$z^* = \frac{\frac{-F+N}{F-N}(-N) + \frac{2FN}{F-N}}{N} = -1$$

für P_z=F

z^*=1

D.h. die Pseudotiefe hat Werte aus [-1,1]

Benutzen homogener Koordinaten

$P = (P_x, P_y, P_z)$, homogene Koordinaten $P = (P_x, P_y, P_z, 1)$

identifiziere (wP_x, wP_y, wP_z, w), $w \neq 0$

Betrachte (wP_x, wP_y, wP_z, w)

Transformation:

$$\begin{pmatrix} N & 0 & 0 & 0 \\ 0 & N & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} wP_x \\ wP_y \\ wP_z \\ w \end{pmatrix} = \begin{pmatrix} wNP_x \\ wNP_y \\ w(aP_z+b) \\ -wP_z \end{pmatrix}$$

$(wNP_x, wNP_y, w(aP_z+b), -wP_z)$

repräsentiert

$$\left(N\frac{P_x}{-P_z}, N\frac{P_y}{-P_z}, N\frac{aP_z+b}{-P_z}, 1\right)$$

Koordinaten der Projektion mit Pseudotiefe

Ergebnis: Perspektive = perspektivische Transformation und orthogon. Projektion

siehe Folie

Das transformierte view volume

`viewVolume(left,top,-N)` usw. transformiert

obere Ebene $\Rightarrow$ Ebene y=top

untere Ebene $\Rightarrow$ Ebene y=bottom

linke Ebene $\Rightarrow$ Ebene x=left

rechte Ebene $\Rightarrow$ Ebene x=right

Ziel: transformiert Einheitswürfel in view volume

Matrix: Projektionsmatrix

$$\begin{pmatrix} \frac{2N}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2N}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{F+N}{F-N} & \frac{-2FN}{F-N} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

r=right, l=left, t=top, b=bottom, N=near, F=far

Ergebnis:

- Projektionsmatrix liefert perspekt. Transformation mit Skalierung und Verschiebung, die das view volume in das kanonische view volume $[-1,1]^3$ überführt

- OpenGL bildet Matrix R und multipliziert mit CM, falls `glFrustum(l,r,b,t,N,F)` aufgerufen wird

- `gluPerspective(viewAngle,aspectRatio,N,F)` leistet das selbe und ist anschaulicher

  Zusammenhang:

$$\begin{aligned} t &= N \tan\left(\frac{\pi}{360} viewAngle\right) \\ b &= -t \\ r &= t \times aspect \\ l &= -r \end{aligned}$$

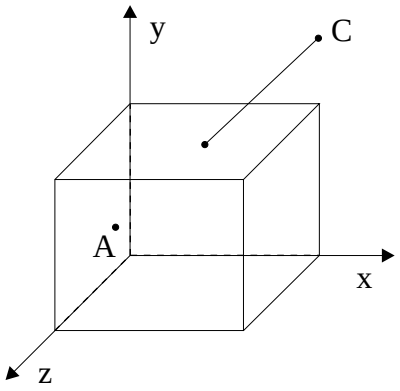
Clippen im view volume

Grafik-Pipeline:

Clippen nach Projektionsmatrix, d.h. Clippen erfolgt in $[-1,1]^3$ = canonical view volume cvv

Clippen basiert auf dem Clippen von Strecken

Cyrus-Beck-Algorithmus in 3D (tatsächlich in 4D)



A=(a_x, a_y, a_z, a_w)

C=(c_x, c_y, c_z, c_w)

O.B.d.A. $a_w > 0$

Ziel: Berechnung des Schnittpunktes

$$I = (I_x, I_y, I_z, I_w)$$

cvv=$[-1,1]^3$ wird durch 6 Begrenzungsebenen realisiert

Stahl $A + \underline{c}t$

Test für Cyrus-Beck: Teste ob Punkt außerhalb oder innerhalb der Begrenzungsebene liegt

Test mit Ebene x=-1

$\frac{a_x}{a_w} > 1$ genau dann ist A innerhalb der Ebene, d.h. genau dann wenn $a_x + a_w > 0$

BC=Randkoordinaten

Randkoordinaten	**homogene Koordinaten**	**Clipp-Ebene**
BC_0	x+w	x=-1
BC_1	w-x	x=1
BC_2	w+y	y=-1
BC_3	w-y	y=1
BC_4	w+z	z=-1
BC_5	w-z	z=1

Bemerkungen:

- Falls alle 6 Werte positiv, so liegt Pkt. innerhalb cvv
- Falls beide Punkte innerhalb, triviales Akzeptieren
- Falls *A* und *C* beide außerhalb auf derselben Seite liegen, triviales Verwerfen

- Andernfalls muss $\overline{AC}$ gegen jede Ebene individuell geclippt werden, man erhält Kandidaten für neues Intervall
- Berechne t_{hit} und aktualisiere

$$t_{in} = max(t_{in}, t_{out})$$
$$t_{out} = min(t_{out}, t_{in})$$

- Berechnung des Schnittpunktes

  Strecke $A + \underline{c}t$

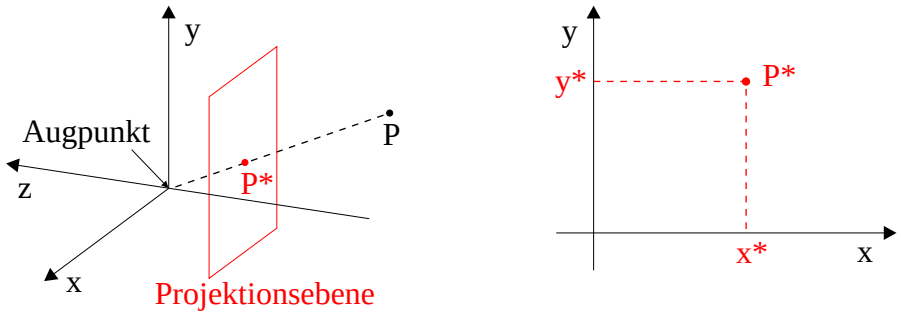
  Schnitt mit der Ebene x=1

$$\frac{a_x + (c_x - a_x)t}{a_w + (c_w - c_x)t}$$

  ergibt

$$t_hit = \frac{a_w - a_y}{(a_w - a_x)(c_w - c_x)}$$

Wiederholung: Perspektivische Projektion



$$P = (P_x, P_y, P_z) \mapsto (x^*, y^*) = \left(N\frac{P_x}{-P_z}, N\frac{P_y}{-P_z} \right)$$

Projektion einer Geraden:

Gerade $A + \overrightarrow{c}t$

$$A = (A_x, A_y, A_z)$$
$$\underline{c} = (c_x, c_y, c_z)$$

Einsetzen in Projektionsgleichung

$$P = A + \underline{c}t \vdash P^*(t) = \left(N\frac{A_x + c_x t}{-A_z - c_z t}, N\frac{A_y + c_y t}{-A_z - c_z t} \right)$$

Ergebnis: Gerade in der Projektionsebene

- geometrische Überlegung
- arithmetisch: hier nicht
- Ausnahme: $\Rightarrow$ Punkt
- wachsendes t:

$$P^*(\infty) = \left(N\frac{c_x}{-c_z}, N\frac{c_y}{-c_z} \right)$$

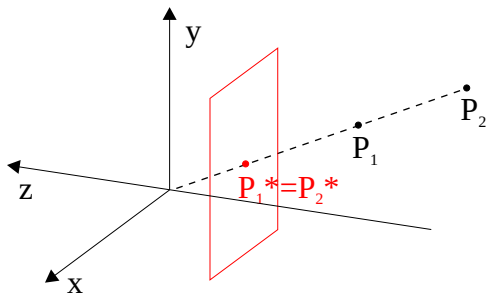
Verschwindungs- oder Fernpunkt

Ergebnis:

$c_z \neq 0$, d.h. die Gerade ist nicht parallel zur Projektionsebene, dann hängt der Verschiebungspunkt nur von $\underline{c}$ ab

Einbau in die Graphik-Pipeline

Problem:



Welcher Punkt P_1, P_2 mit $P_1* = P_2*$ ist näher zum Betrachter?

Einführung: Pseudotiefe

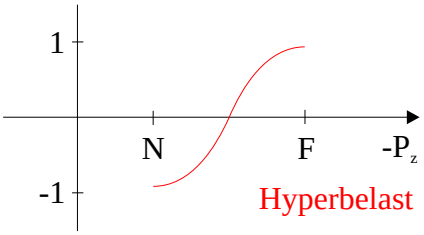
Hinzufügen einer 3. Koordinate

$$(x^*, y^*, z^*) = \left(N\frac{P_x}{-P_z}, N\frac{P_y}{-P_z}, N\frac{aP_z + b}{-P_z} \right)$$

mit

$$a = \frac{F+N}{F-N}, b = -\frac{2FN}{F-N}$$

Pseudotiefe



$$(P_x, P_y, P_Z) \mapsto \left(N\frac{P_x}{-P_z}, N\frac{P_y}{-P_z}, N\frac{aP_z+b}{-P_z} \right)$$

perspektivische Transformation

$$\left(N\frac{P_x}{-P_z}, N\frac{P_y}{-P_z}, N\frac{aP_z+b}{-P_z} \right) \mapsto \left(N\frac{P_x}{-P_z}, N\frac{P_y}{-P_z} \right)$$

Benutzung homogener Koordinaten

$$(P_x, P_y, P_z) \to (P_x, P_y, P_z, 1) = (wP_x, wP_y, wP_z, w)$$

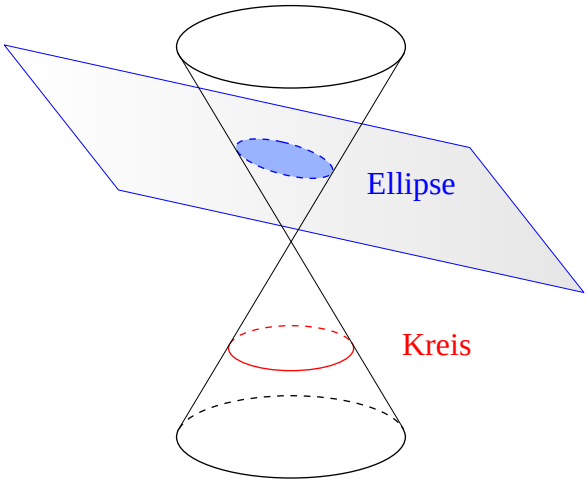
$$\mapsto (wNP_x, wNP_y, w(aP_z+b), -wP_z) : (-wP_z) = \left(N\frac{P_x}{-P_z}, N\frac{P_y}{-P_z}, N\frac{aP_z+b}{-P_z}, 1 \right)$$

Transformationsmatrix:

$$\begin{pmatrix} N & 0 & 0 & 0 \\ 0 & N & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} wP_x \\ wP_y \\ wP_z \\ w \end{pmatrix} = \begin{pmatrix} wNP_x \\ wNP_y \\ w(aP_z+b) \\ -wP_z \end{pmatrix}$$

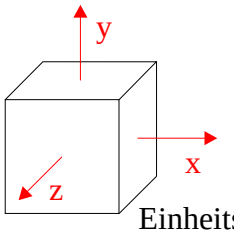
Fakt: Geraden werden in Geraden transformiert

Beispiel: Kegelschnitte



Hyperbel, Parabel und Ellipse sind perspektivische Bilder des Kreises

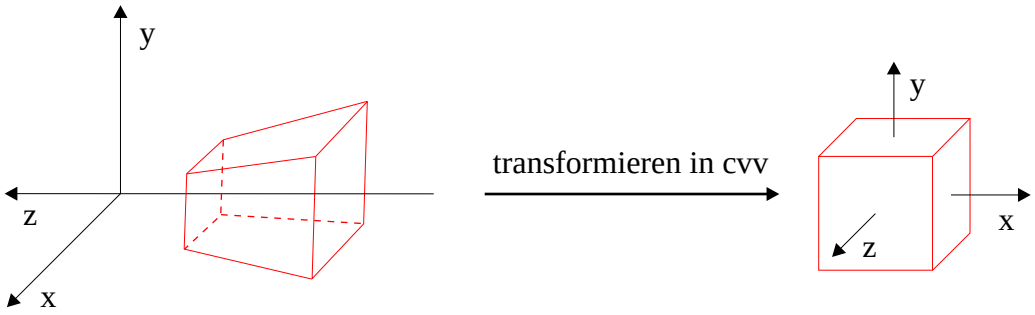
Clippen: canonical view volume (cvv)



Einheitswürfel

$$[-1,1]^3$$

Transformation:



Projektionsmatrix

$$R = \begin{pmatrix} \frac{2N}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2N}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{F+N}{F-N} & -\frac{2FN}{F-N} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

l-left, r-right, t-top, b-bottom, N-near plane, F-far plane

z.B.

$$R \begin{pmatrix} l \\ t \\ -N \\ 1 \end{pmatrix} = \begin{pmatrix} -N \\ -N \\ -N \\ N \end{pmatrix} \cong \begin{pmatrix} -1 \\ -1 \\ -1 \\ 1 \end{pmatrix}$$

usw.

Clippen im cvv

4D-Variante von Cyrus-Beck

Randkomponente	hom. Koordinaten	Clipp-Ebene
BC_0	w+x	x=-1
BC_1	w-x	x=1

entsprechend für y,z

Warum clippen in cvv?

- cvv frei von Parametern, effektives Clippen
- Ebenen von cvv sind mit Koordinatenachsen verbunden (kein Skalarprodukt)

Warum clippen mit homogenen Koordinaten?

- entspricht Graphik-Pipeline
- a_x, a_w haben mehr Info als $\frac{a_x}{a_N}$ (z.B. Vorzeichen!)

Beachte „aspect ratio“!

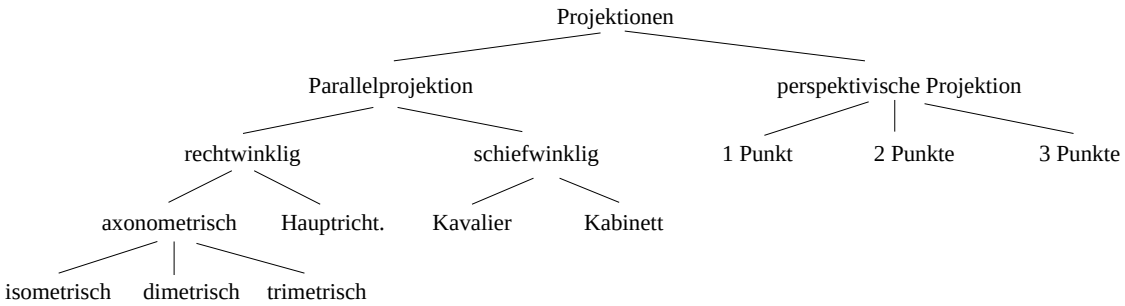
- aspect ratio wird beim Übergang zu cvv auf 1 gesetzt
- viewport-Transformation setzt aspect ratio wieder auf ursprünglichen Wert

Zusammenfassung: Graphik-Pipeline (persp.)

- Koordinaten von V um 1 erweitert $\Rightarrow$ Übergang zu homogenen Koordinaten
- (V,1) wird mit Modelview-Matrix multipliziert $\Rightarrow$ Position von V in Augkoordinaten
- Multiplikation mit Projektionsmatrix $\Rightarrow$ Quadruppel (Viertupel) in Clippkoordinaten (cvv)
- Clippen in cvv
- perspektivische Rücktransformation liefert Tripel von Koordinaten
- Multiplikation des Tripels mit Viewport-Matrix ergibt (s_x, s_y, d_z), wobei (s_x, s_y) gezeichnet wird, d_z ist Tiefeninformation

8.3 Klassifizierung von Projektionen

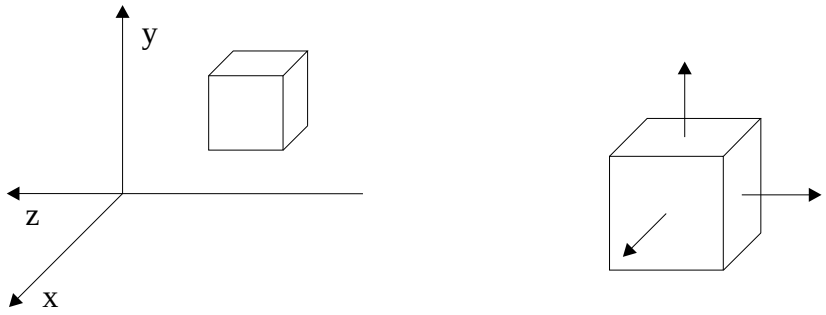
Übersicht:



Rolle der orthographischen Projektionen

Clippebenen begrenzen rechteckigen Quader

$l \leq x \leq r, b \leq y \leq t, N \leq -z \leq F$



Transformationsmatrix

$$R = \begin{pmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{2}{F-N} & \frac{F+N}{F-N} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

z.B.

$$R \begin{pmatrix} l \\ b \\ -N \\ 1 \end{pmatrix} = \begin{pmatrix} -1 \\ -1 \\ 1 \\ 1 \end{pmatrix}$$

usw.

OpenGL: `glOrtho(l,r,b,t,n,f)` multipliziert CM mit R

Spezialfall: `gluOrtho2D` = `glOrtho` mit $N = -1$, $F = 1$

Perspektive

`glFrustum()`

`gluPerspective()`

Zusammenfassung:

- Viewing-System zur Betrachtung von 3D-Szenarien aus verschiedenen Betrachtungspunkten
- Kamerabewegung durch eine Szene, Kamera als Auge modelliert
- Projektion der Szene auf viewplane entspricht Film des Kameramodells
- Details für perspektivische Transformation und Integration in OpenGL-Pipeline

Kapitel 9

Rendern für Fotorealismus

Ziele: fotorealistische Effekte für 3D-Szenen

- Klärung von Reflexionsmodellen
- Rendern polygonaler Netze unter Beleuchtung
- Glätte polygonaler Netze
- verdeckte Flächen eliminieren
- Auftragen von Texturen
- Hinzufügen von Schatten

Bemerkungen:

- Fotorealismus erfordert schnelle Schattierungsmodelle und Lichtquellen
- unterschiedliche Lichtreflektionen:
    - ambient
    - diffus
    - spekular
- Schattierungsmodelle: Phong, Gourand

9.1 Einführung

Rendern = Berechnung jedes einzelnen Pixels der Szene

Schattierung = Berechnung der Lichtstrahlung, die mit der Szene wechselwirkt

Grundlage: polygonale Netze

Flatshading: Berechnung des Lichtes, das auf eine Seite trifft, alle Punkte gleich intensiv

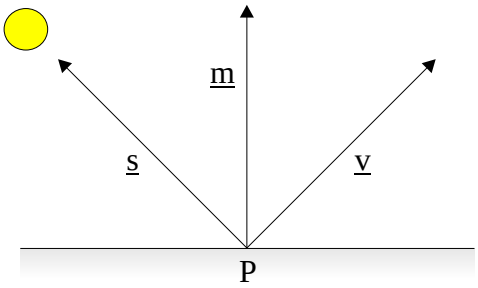
Smoothshading: verschiedene Farbwerte für jedes Pixel (durch Interpolation)

Spekularlicht: Schlaglichter

Schattierungsmodell: Beschreibung, wie Licht reflektiert und gesteuert wird

Modellvorstellung: achromatisches (ohne Farbe) Licht mit Helligkeit (Intensität)

geometrischer Ansatz



$\underline{m}$ Normale

$\underline{s}$ Vektor zur Lichtquelle

$\underline{v}$ Vektor zum Betrachter

Berechnung in Weltkoordinaten

$\underline{v} \cdot \underline{m} > 0 \Rightarrow$ sichtbar

$\underline{v} \cdot \underline{m} \leq 0 \Rightarrow$ unsichtbar

9.1.1 Berechnung der diffusen Reflektion

I_d Intensität des gestreuten Lichtes

I_s Intensität der Lichtquelle

φ_d diffuser Reflektionskoeffizient (materialabhängig)

I_d ist unabhängig von der Richtung und proportional zur Fläche eines Flächenelements

Lambertsches Gesetz

$I_d \sim I_s \cdot \cos\beta$

genauer: $I_d = I_s \cdot \varphi_d \cdot \max\left\{0, \frac{\underline{s} \cdot \underline{m}}{|\underline{s}||\underline{m}|}\right\}$

Beispiel: siehe Kopie (Kugeln links oben)

φ_d=0,0/0,2/0,4/0,6/0,8/1,0

Bemerkungen:

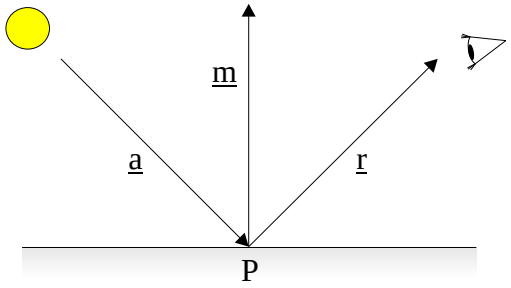
- simplifiziertes Modell, Realität weitaus komplexer
- φ_d ist von der Wellenlänge abhängig

9.1.2 Berechnung der Wellenlänge abhängig

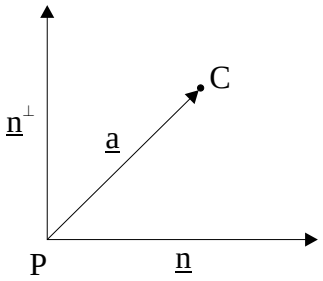
- Modell von Phong (nach Phong Bei-Fuong)
- Reflektion liefert „plastikartigen“Glanz
- Reflektion in der Umgebung der Hauptreflektionsrichtung (siehe Kugel-Kopie rechts oben)

→ **exakte spiegelnde Reflektion**

Problem: Licht trifft aus Richtung $\underline{a}$ in Punkt P auf.
Welche Richtung hat $\underline{r}$?



Wie zerlegt man $\underline{a}$ in ein Vielfaches $k \cdot \underline{n}$ und eine dazu senkrechte Komponente $m \cdot \underline{n}^{\perp}$?



$$\underline{n} = (n_x, n_y) \Rightarrow \underline{n}^{\perp} = (-n_y, n_x) \text{(in 2D)}$$

$\underline{n}^{\perp}$ nicht eindeutig bestimmt

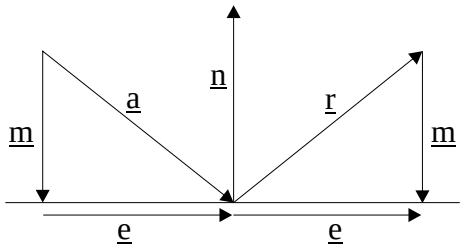
$$\underline{a} = k \cdot \underline{n} + m \cdot \underline{n}^{\perp}, k, m \in \mathbb{R}$$

Lösung für k und m:

$$\underline{a} \cdot \underline{n} = k \cdot \underline{n} \cdot \underline{n} + m \cdot \underline{n}^{\perp} \cdot \underline{n} \Rightarrow k = \frac{\underline{a} \cdot \underline{n}}{|\underline{n}|^2} \tag{9.1}$$

$$\underline{a} \cdot \underline{n}^{\perp} = v \cdot \underline{n} \cdot \underline{n}^{\perp} + m \cdot \underline{n}^{\perp} \underline{n}^{\perp} \Rightarrow m = \frac{\underline{a} \cdot \underline{n}^{\perp}}{|\underline{n}|^2} \tag{9.2}$$

Anwendung: Reflektion



$\underline{a} = \underline{m} + \underline{e}$

$\underline{r} = \underline{e} - \underline{m}$ (Symmetrie)

$\underline{e} = \underline{a} - \underline{m}$

$$\underline{r} = (\underline{a} - \underline{m}) - \underline{m} = \underline{a} - 2\underline{m} \tag{9.3}$$

$\underline{m}$ ist orthogonale Projektion von $\underline{a}$ auf $\underline{n}$

→ Einsetzen von (9.1) und (9.2):

$$\underline{a} = \frac{\underline{n} \cdot \underline{a}}{|\underline{n}|^2} \underline{n} + \frac{\underline{e} \cdot \underline{a}}{|\underline{n}|^2} \underline{e}$$

mit (9.3): Richtung des reflektierten Strahls

$$\underline{r} = \underline{a} - 2\frac{\underline{n} \cdot \underline{a}}{|\underline{n}|^2} \underline{n}$$

mit normierten Vektoren:

$$\underline{r} = \underline{a} - 2(\underline{a} \cdot \underline{n}_0) \cdot \underline{n}_0$$

exakte Reflektion:

$$\underline{r} = -\underline{s} + 2\frac{\underline{m} \cdot \underline{s}}{|\underline{m}|^2} \underline{m}$$

(siehe Kugel-Kopie, Bild c)

→ Welchen Anteil des reflektierten Lichts ein Betrachter aus Richtung $\underline{v}$ sehen kann, ist komplizierte Funktion in φ (siehe Kopie links unten)

⇒ Vereinfachnung: Berechne den Anteil aus $\cos^f(\varphi) = \left(\frac{\underline{r}}{|\underline{r}|} \cdot \frac{\underline{v}}{|\underline{v}|} \right)$

- f wird in Experiment bestimmt, $f \in [0,200]$, $f \in \mathbb{R}$
- $f \to \infty$: exakter Spiegel

$$I_{sp} = I_s \cdot \varphi_{sp} \cdot \left(\frac{\underline{r}}{|\underline{r}|} \cdot \frac{\underline{v}}{|\underline{v}|} \right)^f$$

$\to$ vereinfache weiter: $\underline{h} = \underline{s} + \underline{v}, \beta = 2\varphi$

Gleiche den hierbei gemachten Fehler durch Anpassen von f aus

$$I_{sp} = I_s \cdot \varphi \cdot \max \left\{ 0, \left(\frac{\underline{n}}{|\underline{n}|} \cdot \frac{\underline{m}}{|\underline{m}|} \right)^f \right\}$$

Beispiel: Kugel-Kopie rechts oben

I_{sp}: f=3, 6, 9, 25, 200

0,25

0,5

0,75

9.1.3 Berechnung der ambienten Reflektion

- ambientes Licht ist Hintergrundlicht
- hebt den plastischen Eindruck von Objekten hervor
- überwindet unrealistisch schwarze Schatten
- ambientes Licht ist uniform vorhanden und benötigt keine Lichtquelle

  I_a Intensität
  φ_a Reflektionskoeffizient

- Beispiel auf Kopie rechts unten:
  - ambiente und diffuse Lichtquelle haben Intensität 1.0, I_a=0.04, φ_a=0.0/0.1/0.3/0.5/0.7

9.1.4 Gesamtintensität

$$I = I_a \cdot \varphi_a \cdot \max \left\{ 0, \frac{\underline{s} \cdot \underline{m}}{|\underline{s}||\underline{m}|} \right\} + \max \left\{ 0, \frac{\underline{n} \cdot \underline{m}}{|\underline{n}||\underline{m}|} \right\}$$

9.1.5 Hinzufügen von Farbe

φ_d, φ_{sp}, φ_a sind abhängig von der Wellenlänge

$\Rightarrow$ getrennte Berechnung für R,G,B

$$I_{\{R,G,B\}} = I_{a_{\{R,G,B\}}} \cdot \varphi_{a_{\{R,G,B\}}} + ...$$

also für jeden Farbwert obige Berechnung

drei Typen von Farbe:

ambient I_{aR}, I_{aG}, I_{aB}

diffus I_{dR}, I_{dG}, I_{dB}

spekular I_{spR}, I_{spG}, I_{spB}

$\rightarrow$ verschiedene Werte für Materialeigenschaften und Licht führen zu sehr vielen verschiedenen Möglichkeiten, siehe Tabelle auf Kopie

Bemerkungen:

- ambientes, diffuses Licht basiert auf der Farbe der Fläche
- Farbe der Fläche = Farbe des reflektierten Lichtes, falls die Beleuchtung weiß
- Farbe des spekularen Lichtes = Farbe des reflektierten Lichtes
  z.B. spekulares Licht eines roten Apfels, der gelb angestrahlt wird, ist gelb
- Objekte aus verschiedenen Materialien
  $\varphi_a, \varphi_d, \varphi_s$ in verschiedenen Lichtarten erlauben die Simulation von Materialeigenschaften
  $\Rightarrow$ Tabelle der Folie

Lichtquellen in OpenGL

Erzeugen einer Lichtquelle (bis 8):

`GL_LIGHT0`

Beispiel: Lichtquelle in (3,6,5) in Weltkoordinaten setzen

```
GLfloat myLightPosition={3,6,5,0};
glLightv(GL_LIGHT0,GL_POSITION,myLightPosition);
glEnable(GL_LIGHTING);                          //Aktivieren
glEnable(GL_LIGHT0);                            //Einschalten
```

Bemerkungen:

- $(x,y,z,1)$ Lichtquelle in homogenen Koordinaten, Position in (x,y,z)

- $(x,y,z,0)$ unendlich ferne Lichtquelle in Richtung (x,y,z), parallele Strahlen, z.B. Sonne

Licht: 3 Typen von Licht für jede Lichtquelle

```
GLfloat ambv[]={0.2,0.4,0.6,1.0};
GLfloat diffv[]=...;
GLfloat spec[]=...;
//Lichtquelle
```

vierte Komponente: α-Blending

RGBA-Werte

Übergabe der Parameter

`glLightfv(GL_LIGHT0,GL_AMBIENT,ambv);` usw.

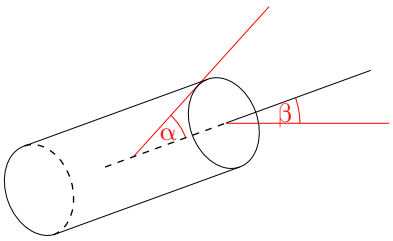
default-Werte

ambient=(0,0,0,1) schwarz

diffus=(1,1,1,1) weiß

spekular=(1,1,1,1) weiß

Spot-Lights



Richtung $\underline{d}$

Öffnungswinkel α

$\cos^{\varepsilon}(\beta)$ Faktor für Lichtquelle um Achse

```
glLightf(GL_LIGHT0,GL_SPOT_CUTOFF,45.0); //α = 45°
glLight(GL_LIGHT0,GL_SPOT_EXPONENT,4.0); //ε = 4.0
GLfloat dir[]=
glLightfv(GL_LIGHT0,GL_SPOT_DIRECTION,dir);
```

default-Werte: $d=(0,0,-1)$, $\alpha=180°$, $\varepsilon=0$

Dämpfung des Lichtes mit Abstand

Ansatz:

$$atten = \frac{1}{K_c + K_e D + K_q D^2}$$

D=Abstand, K_c, K_e, K_q Konstanten

Ansatz erlaubt variable Gestaltung der Dämpfung

Aufruf:

```
glLight(GL_LIGHT0,GL_CONSTANT_ATTENUATION,2.0);
```

entsprechend

```
GL_LINEAR_ATTENUATION
GL_QUADRATIC_ATTENUATION
```

default-Werte $K_c = 1, K_e = K_q = 0$

Bemerkungen:

- 3 Parameter zum Setzen und Spezifizieren des Beleuchtungsmodells
- Farbe des globalen ambienten Lichtes unabhängig von Lichtquelle
- Variation des „viewpoint“benutze „half way“Vektor
- richtige Beleuchtung von Vorder- und Rückseite eines Polygons (Orientierung mit Normalenvektor)

Erweiterung des Lichtmodells = Berechnung der Lichtintensität

Formel für r-Komponente

$$I_r = e_r + I_{mr}\varphi_{mr} + \sum_i atten_i \times spot_i \times (I_a r^i \varphi_{ar} + I_{dr}\varphi_{dr} \times lambert_i + I^i_{sr}\varphi_{sr} \times phong_i^f)$$

Hierzu:

e$_r$ emmitiertes Licht

I$_m r$ globales, ambientes Licht

φ_{mr} Konstante -“-

$\sum$ Summe über alle Lichtquellen

Dämpfung der i-ten Quelle:

$$atten_i = \frac{1}{K^i_c + K^i_e D + K^i_q D^2}$$

Abschwächung des Spotlights der i-ten Quelle:

$$spot_i = \cos^{\varepsilon}(\beta_i)$$

i-te Quelle:

$$lambert_i = max\left(0, \frac{\underline{s} \cdot \underline{m}}{|\underline{s}||\underline{m}|}\right)$$
$$phong_i = max\left(0, \frac{\underline{h} \cdot \underline{m}}{|\underline{h}||\underline{m}|}\right)$$

$I^i_{ar}, I^i_{dr}, I^i_{sr}$ ambientes, diffuses, spekulares Licht der i-ten Quelle

$\varphi_{ar}, \varphi_{dr}, \varphi_{sr}$ Materialkonstanten vom Objekt

entsprechende Formel für I_g, I_b

9.2 Schraffierungen = Shading

Aufgabe: Informationen zum Normalenvektor werden zur Schraffierung benutzt

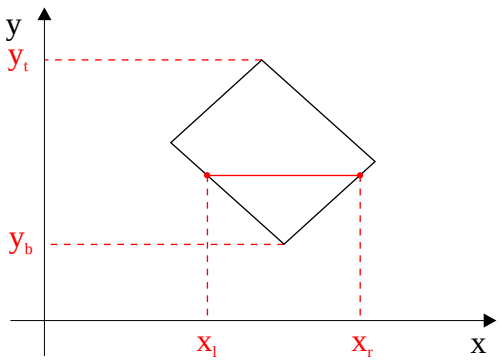
Unterschiede:

- flat shading: Schraffierung jedes einzelnen Polygons
- smooth shading: Glättung beim Übergang an Kanten zweier Polygone

Schraffieren einer Seite:

Dachdecker (Scan Line)

- pixelweise Zeile für Zeile von links nach rechts
- zeilenweise von unten nach oben



Prozedur:

```
for(int y=yb;y<=yt;y++)        //zeilenweise
{
    finde xl und xr
    for(int x=xl;x<=xr;x++)    //scan line
    {
        finde Farbe c fuer das Pixel;
        setze c fuer Pixel (x,y);
    }
}
```

flat shading:

Seitenfläche eben, Licht relativ weit entfernt

diffuses Licht ändert sich wenig beim Übergang von Punkt zu Punkt

d.h. setzen der Farbe außerhalb der Schleife

Ergebnis:

- starkes Hervortreten der Kanten
- keine spekularen Lichteffekte

OpenGL: `glShadeMode(GL_FLAT)`

smooth shading:

Idee: Unterdrückung der Kantenübergänge

Gouraud-Schraffierung (OpenGL-unterstützt)

Phong-Schraffierung

Hierzu: lineare Interpolation zweier Punkte

A, B Punkte

Strecke $P(t) = A + (B - A)t$, $t \in [0,1]$

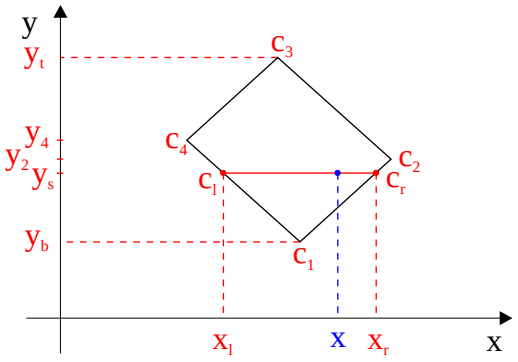
komponentenweise für jede Koordinate

Funktion:

$$lerp(a,b,t) = a + (b-a)t$$

Gouraud-Schraffierung

= berechne verschiedene Werte c für jedes Pixel durch lineare Interpolation



Vorgehensweise:

- finde c_l, c_r in der Scan Line

$$c_l = lerp(c_3, c_4, f), f = \frac{y_s - y_t}{y_4 - y_t}$$

(für jede Farbkomponente)

$$c_r = lerp(c_1, c_2, f), f = \frac{y_s - y_b}{y_2 - y_b}$$

- finde c_x

$$c_x = lerp\left(c_l, c_r, f = \frac{x - x_l}{x_r - x_l}\right)$$

Vereinfachung: inkrementelles Fortschreiten

$$c_{x+1} = c_x + \frac{c_r - c_l}{x_r - x_l}$$

Vorteil: $\frac{c_r - c_l}{x_r - x_l}$ wird nur einmal berechnet

Bemerkung:

- rechenintensives Verfahren
- OpenGL `glShadeModel (GL_SMOOTH)`

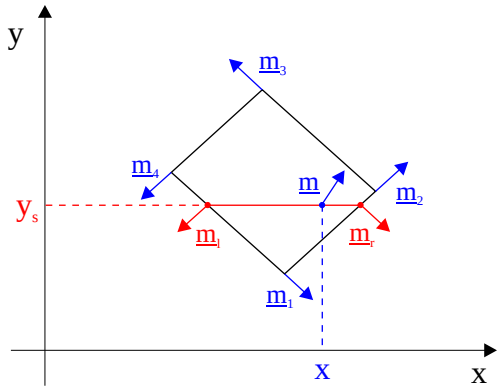
Effekte:

- Interpolation glättet Lichteffekte
- Schlaglichter treten deutlich hervor

Phong-Schraffierung

bessere Approximation des Normalenvektors Phong, Bei-Huong

- finde den Normalenvektor an jedem Punkt der Seitenfläche
- berechne den Farbwert für jedes Pixel



- Bestimme $\vec{m}_l, \vec{m}_r$

$$\vec{m}_l = lerp\left(\vec{m}_1, \vec{m}_4, \frac{y_s - y_1}{y_4 - y_1}\right)$$
$$\vec{m}_r = lerp\left(\vec{m}_2, \vec{m}_3, \frac{y_s - y_2}{y_3 - y_2}\right)$$

- Bestimme $\vec{m}$

$$\vec{m} = lerp\left(\vec{m}_r, \vec{m}_l, \frac{x - x_l}{x_r - x_l}\right)$$

Ergebnis: Übergabe von $\vec{m}$ an das Beleuchtungsmodell

Vergleich:

- Phong-Schraffierung liefert bessere Plastizität der Figuren
- gute Schlaglichteffekte
- nachteilig gegenüber Gouraud-Schraffierung ist 6-8 mal höhere Rechenzeit
- Reihe von Modifikationen zur Verbesserung der Rechenzeit
- OpenGL unterstützt Phong-Schraffierung nicht

(Hierzu: Anwendung des Schraffierungsmodells erfolgt einmal pro Ecke nach Modellview-Transformation. Normalenvektoreninformationen bleiben nicht erhalten beim Rendern nach persp. Transformation
Alternativen: Schlaglicht-Texturierung)

9.3 Elimination verdeckter Flächen

Voraussetzung: genügend Speicherplatz

Idee:

Tiefen- bzw. Z-Buffer

einfache Methode, die in Graphik-Pipeline passt

Alternativen: Computergraphik II

Nachteil Z-Buffer:

Rendern von Objekten, die sich später als überflüssig erweisen

Vorgehensweise:

für Pixel p[i][j] speichere Tiefenbuffer b-Bit-Größe d[i][j]

b zwischen 12 und 30 Bits

Scan-Prozess:

- Ausfüllen einer Seite erreicht man Pixel p[i][j]
- Test, ob Pseudotiefe der aktuellen Seite kleiner als Pseudotiefe d[i][j] ist, die im Speicher abgelegt ist
- falls „ja“: die Farbe der näheren Fläche ersetzt p[i][j], kleinerer Wert in d[i][j] ersetzt
- falls „nein“: keine Änderung

Bemerkungen:

- Vorteil ist, dass jede Fläche bearbeitet werden kann
- Nachteil ist hoher Speicherbedarf
- d[][] wird mit 1.0 initialisiert, dem größten Wert der Pseudotiefe
- Framebuffer wird mit schwarzem Hintergrund initialisiert

Berechnung der Pseudotiefe:

Ansatz: $P = (P_x, P_y, P_z)$ Ecke

wird in Pseudotiefe übergeben, wird transformiert

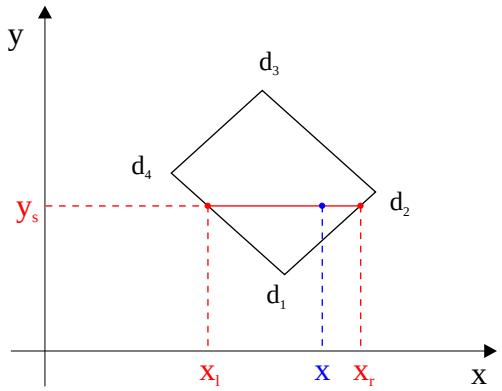
wird

$$(x, y, z) = \left(\frac{P_x}{-P_z}, \frac{P_y}{-P_z}, \frac{aP_z + b}{-P_z} \right)$$

bzw. skalierter oder verschobener Wert hiervon

$\frac{aP_z+b}{-P_z}$ Pseudotiefe

Skizze:



Pseudotiefe der Ecken d_1,d_2,d_3,d_4 sind bekannt

Problem: Wie erhält man die korrekte Tiefeninformation für (x,y_s)?

Antwort: lineare Interpolation gibt die Lösung nicht her

Lösung: hyperbolische Interpolation

geg.:

$$P_1 = (x_1, y_1, z_1)$$
$$P_2 = (x_2, y_2, z_2)$$
$$P = lerp(P_1, P_2, g) = (x_1 + (x_2 - x_1)g, y_1 + (y_2 - y_1)g, z_1 + (z_2 - z_1)g)$$

Pseudotiefen:

$$d_1 = \frac{az_1 + b}{-z_1}$$
$$d_2 = \frac{az_2 + b}{-z_1}$$
$$d = \frac{a(z_1 + (z_2 - z_1)g) + b}{-(z_1 + (z_2 - z_1)g)}$$

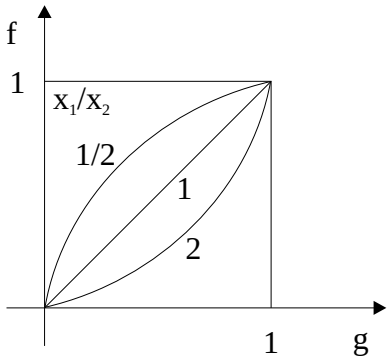
Problem: Welcher Zusammenhang besteht zwischen d und d_1,d_2? Ist etwa d=lerp(d_1,d_2,f) für einen geeigneten Wert f?

Lösung:

$$d = lerp(d_1, d_2, f) \quad frf = \frac{g}{lerp\left(\frac{x_1}{x_2}, 1, g\right)} = \frac{g}{\frac{x_1}{x_2} + \left(1 - \frac{x_1}{x_2}\right) g}$$

Bsp.:

$g = 0$	$P = P_1$	$f = 0$	$d = d_1$
$g = 1$	$P = P_2$	f=1	$d = d_2$
$g = \frac{1}{2}$			$f = \frac{x_2}{x_1 - x_2}$



Vorgehensweise für Scanline-Verfahren:

Scanline y=y_s

Berechne:

$$d_l = lerp(d_1, d_2, f)$$
$$d_r = lerp(d_1, d_2, h)$$
$$d_{(x,y)} = lerp(d_l, d_r, k)$$

wobei f,h,k entsprechend der hyperbolischen Interpolation gewählt werden

Pseudocode: Ergänzung der Gouraud-Schraffierung zur Elimination verdeckter Flächen

```
for(int y=yb;y<=yt;y++)
{
    finde xl,xr
    finde dl,dr
    finde cl,cr
    for(int x=xl,c=cl,d=dl;x<=xr;x++,c+=cinc,d+=dinc)
         if(d<d[x][y])
         {
            setze c auf Pixel (x,y);
```

```
                d[x][y]=d;              //update Tiefeninfo
            }
    }
```

Arbeit in OpenGL

- unterstützt Tiefenbuffer
- Erzeugen eines Buffers

  `glutInitDisplayMode(GLUT_DEPTH|GLUT_RGB)`

- Test mit

  `glEnable(GL_DEPTH_TEST)`

- Erzeugen neuer Bilder: neu initialisieren

  `glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT)`

weitere photorealistische Phänomene:

Texturen, Schatten

zurückgestellt (Raytracing)

Zusammenfassung:

- Verbessern des realistischen Aussehens durch Beleuchtungseffekte, Oberflächenfärbung, Weglassen verdeckter Polygonteile
- Modellierung komplexer Wechselwirkungen des Lichtes mit Materialien und der Umgebung
- Modellbildungsvarianten
  Lambert, Gouraud, Phong
- Kennenlernen des Einflusses von Materialkonstanten für Beleuchtungsphänomene
- Rendern von polygonalen Objekten pixelweise mit Scanline-Verfahren
  (inkrementelles Aufbereiten zur Verringerung der Rechenzeit)

Kapitel 10

Einführung in das Raytracing

Ziele:

- Konzept des Raytracing
- Entwicklung mathematischer Algorithmen
- realistische Effekte, die auf Transparenz und Brechung beruhen
- Szenen mit Kugeln, Quadern, Kegelstümpfen
- Entwickeln von Werkzeugen für Texturen und Bitmaps

Raytracing: (raycasting)

> Strahlverfolgung und deren Wechselwirkung mit verschiedenen Körpern, Aussenden eines Strahls vom Auge und Aufsammeln der Lichtintensität

10.1 Einführung

Grundidee: Array des Bildschirms, für jedes Pixel wird die Frage gestellt, was sieht man durch dieses Pixel?

Realität: Prozess umgekehrt

Bemerkungen:

- HSR (hidden surface removal) ist nicht notwendig, fällt mit ab
- Lichtquellen in Modell einbeziehen, Licht trifft auf Körper und wird vom Auge aufgenommen
- Reflexion, diffuse Beleuchtung integrieren
- Integration komplexer Gebilde als Polygonnetze

Bereitstellung geometrischer Grundlagen

> eye = Kamera bzw. Augpunkt
>
> N = near plane = Ebene des Schirms

Gestalt der Kamera:

Blickwinkel θ
Höhe des Schirms: $H = N \cdot \tan \frac{\theta}{2}$
Breite des Schirms: $W = H \cdot aspect$

Viewport-Transformation:

Transformation von N in den Viewport

Vorgaben:

nCols= # Spalten

nRows= # Zeilen

Zählung 0,...,nCols-1 bzw. 0,...,nRows-1

(r,c)-Pixel: Pixel in r-ter Zeile und c-ter Spalte

(u_c, v_r) = Koordinaten des linken unteren Endpunktes des Pixels mit

$$u_c = -W + \frac{W \cdot 2c}{nCols}, c = 0, ..., nCols - 1$$
$$v_r = -H + \frac{H \cdot 2r}{nRows}, c = 0, ..., nRows - 1$$

Aufgabe:

Bestimme Gleichung des Strahls durch den Punkt (u_c, v_r)

Lösung: Richtung zum Augpunkt

$$eye - N\underline{n} + u_c\underline{u} + v_r\underline{v}$$

Gleichung des Strahls

$$r(t) = eye(1-t) + (eye - N\underline{n} + u_c\underline{u} + v_r\underline{v})t = eye + \underline{dir}_{rc} \cdot t$$

$\underline{dir}_{rc}$ Richtung vom Augpunkt durch das (r,c)-Pixel

$$\underline{dir}_{rc} = -N\underline{n} + W\left(\frac{2c}{nCols} - 1\right)\underline{n} + H\left(\frac{2r}{nRows} - 1\right)\underline{v}$$

Definition: Strahl mit dieser Gleichung soll rc-Strahl heißen

Bemerkung:

$t = 0$ Augpunkt

$t = 1$ Pixel

t variiert bis Strahl ein Objekt der Szene schneidet

Fakten:

1. Falls zwei Objekte t_a,t_b geschnitten werden, liegt derjenige näher am Augpunkt, dessen t-Wert kleiner ist.
2. Tiefensortierung der Objekte entspricht der Sortierung der Schnittpunkte t_{hit}, in denen der Strahl die Objekte schneidet
3. Negative t_{hit} werden ignoriert, liegen hinter dem Augpunkt

Raytracing-Algorithmus (Grundgerüst):

definiere Objekte der Szene, Lichtquellen
setze Augpunkt

```
for(int r=0;r<nRows;r++)
    for(int c=0;c<nCols;c++)
    {
        1. rc-Strahl berechnen
        2. berechne Schnitte des rc-Strahls mit Objekten
        3. bestimme nächstgelegenen Schnitt
        4. bestimme Koordinaten des Schnittes und Normalenvektor
        5. bestimme Farbe
        6. setze Farbwert in (r,c)-Pixel
    }
```

Bemerkungen:

- Schritte 3,4,5 neu, detaillierte Besprechung folgt
- t_{hit} = Schnittzeit, Wert des Parameters, an dem (r,c)-Strahl das Objekt trifft
- falls alle Objekte getestet sind, wird kleinster positiver t_{hit}-Wert gesetzt
- berechne Schnittpunkt und Normalenvektor
- Beschreibung der Objekte mit Liste von Objekten

10.2 Schnittberechnungen

Hinweis: Klassen programmieren

Objekt einer Szene: definiert und in der Szene abgelegt

Objekt = Beispiel einesgenerisch abgelegten Objektes (Kugel,Kegel,Würfel usw.)

Transformation: Wie wird das generische Objekt in der Szene abgelegt?

Zeichnen: OpenGL: `drawOpenGL()` für jedes graphische Primitiv

Schnittberechnung:

implizite Form der Flächen

z.B. generische Kugel

$$F(x,y,z) = x^2+y^2+z^2-1$$
$$F(P) = |P|^2-1$$

generischer Zylinder

$$F(x,y,z) = x^2+y^2-1, 0 \leq z \leq 1$$

Bemerkung:

- Vorteil der generischen Situation, spezielle Lage eines Zylinders im Raum, skalierte verschobene Kugel ergibt unklare implizite Gleichungen
- implizite Form zur Schnittberechnung

  S Augpunkt
  $\underline{dir}_{rc} = \underline{c}$ Richtung
  Gleichung des Strahls

  $$r(t) = S + \underline{c}t$$

  Fläche (implizit gegeben): $F(P) = 0$
  Gleichung: $F(r(t)) = 0$ wird gelöst: $F(S+\underline{c}t) = 0$

Beispiele:

1. Ebene

   generische Ebene

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

   Strahl

   $$r(t) = S + \underline{c}t$$

Einsetzen:

$$s_z + c_z t_{hit} = 0$$
$$t_{hit} = -\frac{s_z}{c_z}$$

falls $c_z \neq 0$

falls $c_z = 0$ parallel zur Ebene $z = 0$

Ergebnis $P_{hit} = S - \frac{s_z}{c_z}\underline{c}$

2. generische Kugel

$$F(P) = |P|^2 - 1 = 0$$

Strahl

$$r(t) = S + \underline{c}t$$

Einsetzen:

$$|S + \underline{c}t|^2 - 1 = 0$$

Umformen

$$At^2 + 2Bt + C = 0$$

mit

$$A = |\underline{c}|^2$$
$$B = \underline{s} \cdot \underline{c}$$
$$C = |S|^2 - 1$$

Lösung:

$$t_{hit} = -\frac{B}{A} \pm \frac{\sqrt{B^2 - AC}}{A}$$

Lösungsdiskussion:

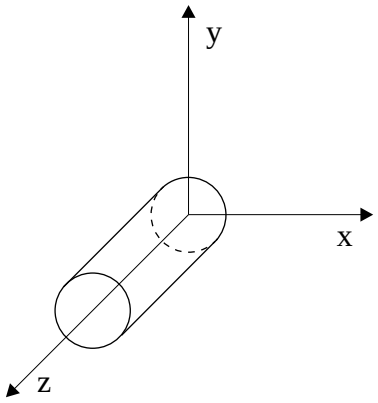
a) $B^2 - AC < 0$ kein Schnittpunkt, Strahl geht an Kugel vorbei

b) $B^2 - AC = 0$ Berührungspunkt $t_{hit} = -\frac{B}{A}$

c) $B^2 - AC > 0$ zwei Schnittpunkte t_1, t_2

3. generischer Zylinder

$$F(x,y,z) = x^2 + y^2 - 1 = 0, 0 \leq z \leq 1$$



Strahl

$$r(t) = S + \underline{c}t$$

Einsetzen von x,y-Koordinaten //(des Strahls) liefert

$$At^2 + 2Bt + C = 0$$

berechne Lösung t_{hit} wie zuvor

beachte $0 \leq z \leq 1$ für P_{hit}

Schnitte mit transformierten Objekten

T Transormation(smatrix)

z.B. Kugel in Ellipsoid

W'$\rightarrow$ W

Lösung des Schnittproblems für transformierte Flächen

$G(S + \underline{c}t)$ für transformierte Fläche

Lösung der transformierten Gleichung

$$F(T^{-1}(S + \underline{c}t)) = 0$$

Anwendung:

$T^{-1}S + T^{-1}\underline{c}t$ führt zu dem modifizierten Schnittproblem (hom. Koordinaten)

$$(*)\tilde{r}(t) = M^{-1}\begin{pmatrix} S_x \\ S_y \\ S_z \\ 1 \end{pmatrix} + M^{-1}\begin{pmatrix} c_x \\ c_y \\ c_z \\ 0 \end{pmatrix} t$$

Zusammenfassung:

Schnitt eines umgekehrt transformierten Strahls mit generischem Objekt, anstelle Schnitt eines Strahls mit transf. gener. Objekt

Vorgehensweise:

- jedes Objekt der Objektliste hat eigene Transformation (Transf. des gen. Objektes in Szene)
- Berechnung des Schnittes mit transf. Objekt sind folgende Schritte erforderlich:
  1. mit der umgekehrten Transformation M^{-1} wird Strahl transformiert (*)
  2. Berechnung von t_{hit} des transformierten Strahls mit dem gen. Objekt
  3. Benutze t_{hit} in $S + \underline{c}t$ um den aktuellen Schnittpunkt zu erhalten

Vorteil:

- Schnittberechnung erfolgt ausschließlich mit gen. Objekten
- erlaubt einfaches Programmieren, rechenzeitintensiv

10.3 Raytracer-Anwendungen

Ziel: Pseudocode des Raytracers soll ausgefüllt werden

Hilfsmittel: Klassen zur Beschreibung komplexer Szenen, Objektlisten mit affinen Transform. für jedes Objekt

- objektor. Zugang sollte definieren welche Objekte betrachtet werden, welche Aktionen diese betreffen, welche Datentypen unterstützt
- Informationen zu den Objekten, damit diese die gewünschten Aktionen unterstützen

Klasse `camera` (Ausführungen gemacht)

```
void camera::raytrace(Scene& scn,int blockSize)
```

Aufgaben:

- Übergabe der Szene

- Festlegung der Pixelblockgröße
- generiert (r,c)-Strahl
- bestimmt Farbwert in P_{hit}
- zeichnet Pixel

Codefragment

```
void display(void)
{
     glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
     cam.drawOpenGL(scn);                              //Vorschau
     cam.raytrace(scn,blockSize);
}
```

Modifikation

OpenGL zum Zeichnen

„preview“vor eigentlichem Raytracing

Preview:

- korrekte Kameraposition
- Szene wird pixelweise geändert
- Entdecken von Fehlern beim Raytracing

Details: Bestandteile von `raytrace();` Klasse Ray

```
class Ray
{
    public:
        Point3 start;
        Vector3 dir;

        void setStart(Point3& p)
        {
            start.x=p.x;...
        }

        void setDir(Vector3& v)
        {
            dir.x=v.x;...
        }
```

```
        //weitere
};
```

Skelett des Raytracers: Folie

Bemerkungen:

- theRay: setzt Startpunkt auf Augpunkt und berechnet (r,c)-Strahl für jedes Pixel
- scn: Szene Objekt, berechnen der Farbe, die zurückgegeben wird

  ```
  clr=scn.shade(theRay)
  ```

- `shade()`: Hauptarbeit

  bestimmt Schritte, berechnet Farbe und Licht, gibt Farbwert zurück

  Pixelgröße: blockSize

  Idee ist Zusammenfassung von einzelnen Pixeln zu Pixelblöcken
  $\Rightarrow$ Beschleunigung des Raytracing für grobes, erstes Rendern

  Blockgröße 4 = 16-fache Beschleunigung

Bemerkungen:

- `raytrace()` initialisiert Matrizen zum Zeichnen der Pixelgröße
- „modelview“-Matrix = Identity, Projektionsmatrix skalieren

Grundlagen zu `shade()` Aufgaben:

- Bestimme, ob Objekt geschnitten wird
- wenn ja, welches Objekt

zur Routine `shade()`

- `getFirstHit(ray,best)` gibt Daten über ersten Schnitt (genannt „best“) zurück
- falls Informationen über ersten Schnitt verfügbar setzt `shade()` mit dem Auffinden der Farbe fort
- falls kein Objekt geschnitten wird, wird Hintergrundfarbe zurückgegeben
- falls Objekt geschnitten wird, sammelt `shade()` Beiträge zur Variablen color auf

1. Glühstrahlung, ambientes, diffuses und spekulares Licht laut Lichtmodell
2. reflektiertes Licht, gebrochenes Licht von anderen Oberflächen

   Routine `getFirstHit()`

Routine `getFirstHit()`

- `getFirstHit(ray,best)` gibt Daten über Array der Schnitte, genannt best, zurück
- falls Info über Schnitt verfügbar, setzt `shade()` mit Auffinden der Farbe fort
- falls kein Schnitt verfügbar, wähle Hintergrundfarbe

Implementierung: Liste der Schnitte

Klasse `intersection`

```
class intersection
{
    public:
        int numHits;
        HitInfo[];
        ...
};
```

Klasse mit zwei Feldern

der Schnittparameter

Liste mit Daten

Bemerkungen:

- t_{hit} nur zulässig, falls $t_{hit} > 0$
- t_{first} von spez. Interesse, wenn Strahl erstmals ein Objekt schneidet

Frage: Warum enthält Liste Infos über alle t_{hit} und nicht t_{first}?

Antwort:

- Behandlung Boolscher Objekte, z.B. $(A \cap B \setminus C) \setminus D$ wobei A, B, C, D Körper
- mehrfaches Durchdringen eines Strahls in einen Körper

Klasse `HitInfo`

Daten für Schnittpunkt mit Fläche

```
class HitInfo
{
    public:
        double hitTime;       //Schnittparameter
        GeomObj hitObj;       //geschnittenes Objekt
```

```
        bool isEntering;      //eintreffend?
        int surface;          //welche Fl"ache
        Point3 hitPoint;      //Schnittpunkt
        Vector3 hitNormal;    //Normale
        ...
};
```

Beispiel: `getFirstHit`

- scannt Objektliste
- testet auf Schnitt
  Objekte haben eigene `hit()`-Routine
  Rückgabe `hit()` true falls legitimer Schnitt vorliegt
- Vergleich ersten positiven Schnitt mit t_{best}
- eventuell aktualisieren
- initialisiere best. `numHits` auf 0
- `getFirstHit()` vernachlässigt Komplexität der Schnittberechnung auf Routine `hit()`
  - größere Effizienz bei Kenntnis des Objekts
  - Beispiel zur Benutzung von Polymorphismen
  - effiziente, robuste Herangehensweise
  - `hit()` ist virtuelle Methode der GeomObj-Klasse, von der alle aktuellen shape-Klassen abgeleitet sind

Beispiel: `hit()` für Klasse `Sphäre`

Erläuterungen:

- Transformation des Strahls in gen. Koordinaten der Kugel
- Umkehrabbildung: `xfrmRay()`

$$\tilde{r}(t) = M^{-1} \begin{pmatrix} S_x \\ S_y \\ S_z \\ 1 \end{pmatrix} + M^{-1} \begin{pmatrix} c_x \\ c_y \\ c_z \\ 0 \end{pmatrix} t$$

  `xfrmRay()` implementieren

- Koeffizienten A, B, C der gen. Situation

$$At^2 + 2Bt + C = 0$$
$$A = |\underline{c}|^2$$
$$B = \underline{s} \cdot \underline{c}$$
$$C = |\underline{s}|^2 - 1$$

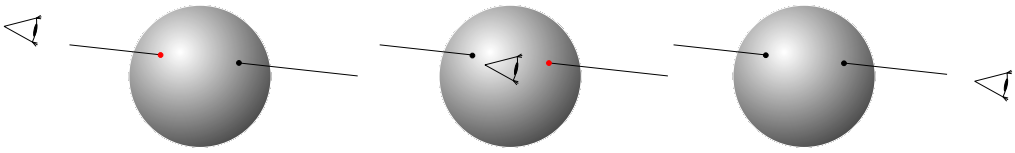
  werden bestimmt und $B^2 - AC$ getestet

- Falls $B^2 - AC < 0$: kein Schnittpunkt

  `hit()` gibt false zurück, `inter()` wird nicht aufgerufen

- Falls $B^2 - AC > 0$: zwei Schnittparameter

  3 Möglichkeiten



$0<t_1<t_2$ $t_1<0<t_2$ $t_1<t_2<0$

- Falls $t_1 > 0$: inter.hit[0] enthält Schnittdaten und num:=1

- Falls $t_2 > 0$: inter.hit[num]
  Falls erster Schnitt <0, wird t_2 in hit[0] abgelegt

- Koordinaten der Punkte (in gen. Koordinaten) an denen Strahl Kugel schneidet: hitPoint

  `Point3 rayPos(Ray& r,float t)`
  gibt Position des Strahls zum Zeitpunkt t zurück

- Normale in Schnittpunkt = Richtungsvektor des Punktes (in gen. Koordinaten)

Raytracer für leuchtende Kugeln

Szene bestehend aus Kugeln, leuchtend

10.4 Schnitte mit weiteren Objekten

Ziel: `hit()` für weitere Primitive

Vorgehensweise:

- Transformation des Strahls in gen. Koordinaten des Objekts
- Berechnung der Schnitte mit Objekt
- detaillierte Prozedur für jeden Objekttyp

Quadrat generisches Quadrat

$z = 0, -1 \leq x,y \leq 1$

Implizite Gleichung:

$$F(P) = P_z, |P_x| \leq 1, |P_y| \leq 1$$

(Transformation in jedes ebene Parallelogramm)

`hit()` findet Strahl in gen. Koordinaten, testet, ob Schnitt vorliegt

Kegelstumpf

Implizite Gleichung

$$(*)F(x,y,z) = x^2 + y^2 - (1 + (s-1)z)^2, 0 \leq z \leq 1$$

Spezialfälle:

s=1 Zylinder

s=0 Kegel

`hit()` für Kegelstumpf

Problem: verschiedene Typen von Strahlen, die Objekt unterschiedlich schneiden

1. zwei Schnittpunkte
2. Schnitt des Mantels und Vorderseite
3. Schnitt des Mantels und Hinterseite
4. Schnitt Vorder- und Hinterseite
5. kein Schnitt

(if-else-Tests!)

Schnittanalyse

$S + \underline{c}t$ Strahl einsetzen in (*)

$$(S_x + c_x t)^2 + (S_y + c_y t)^2 - (1 - (s-1)(S_z + c_z t))^2 = At^2 + Bt + C = 0$$

mit

$$A = c_x^2 + c_y^2 - d^2, d = (s-1)c_z$$
$$B = S_x c_x + S_y c_y - F d, F = (1 + (s-1)S_z)$$
$$C = S_x^2 + S_y^2 - F^2$$

Vorgehensweise:

- Falls $B^2 - AC < 0$: kein Schnitt
- Falls $B^2 - AC > 0$: Strahl schneidet den Kegelmantel bestimmt t_1, t_2
- Teste, ob echte Schnitte mit Kegelstumpf vorliegen:
  - Bestimme z-Koordinaten des Schnittpunktes
  - Strahl schneidet Kegelstumpf, falls $0 \leq z \leq 1$ ist
  - Test für Grundfläche z=0 ergibt (x,y,0) liegt innerhalb der Grundfläche, falls $x^2 + y^2 \leq 1$
  - Test für Deckfläche z=1 ergibt(x,y,1) liegt innerhalb der Deckfläche, falls $x^2 + y^2 \leq s^2$

Typifizierung der Flächen

Mantel 0

Grundfläche 1

Deckfläche 2

surface-Feld der Hit-Liste

Würfel (bzw. allgemeine konvexe Polyeder)

konv. Polyeder: durch Stützebenen begrenzte Körper

Würfel: spezielles Polyeder

gen. Würfel

Eckpunkte ($\pm 1, \pm 1, \pm 1$)

Ebenden der Seiten: x=1, x=-1, y=1, y=-1, z=1, z=-1

Hinweis:

Name, Gleichung, nach außen gerichtete Normale, Pkt =spot der Ebene

Folie: `hit()`-Prozedur

Bedeutung des gen. Würfels

- Modellierung von Quadern
- gen. Würfel als Umhüllung im Sinne einer bounding-Box zur effektiven Schnittberechnung
- effiziente Algorithmen

Schnittalgorithmus für gen. Würfel

- Cyrus-Beck-Algorithmus wird modifiziert
- teste gegen jede Seite und bestimmt Schnittparameter

Kandidatenintervall CI (current interval) = Strecke, die nach bisherigen Tests innerhalb liegt: CI=$[t_{in}, t_{out}]$

Pseudocode:

`if(Strahl dringt bei` t_{hit} `ein)`

$t_{in} = max(t_{in}, t_{hit})$

`else if(Strahl verlässt bei` t_{hit}`)`

$t_{out} = min(t_{out}, t_{hit})$

Schnittalgorithmus für konvexe Polyeder

`hit()` für konvexe Polyeder

Idee: Benutze Stützebenen

i-te Ebene:

B_i Punkt

$\underline{m}_i$ nach außen gerichteter Normalenvektor

Modifiziere Algorithmus für gen. Würfel

benutze das teurere Skalarprodukt

$\underline{m}_i$(B_i-s)

$\underline{m}_i \underline{c}$

Schnittalgorithmus für Polygonnetze

Wiederholung:

Polygonnetz: Datenstruktur
Liste von Seitenflächen:
Listen von Ecken der Seite
Listen der Normalen

Idee: Behandle jede Seitenfläche wie Begrenzungsebene

Schnittalgorithmen für graphische Primitive

1. Würfel

   Transformation in gen. Koordinaten

   Pseudocode:

   `if(Strahl dringt bei` t_{hit} `ein)`

   $t_{in} = max(t_{in}, t_{hit})$

   `else if(Strahl verlässt bei` t_{hit}`)`

   $t_{out} = min(t_{out}, t_{hit})$

   Schnittberechnung:

   Strahl: $S + \underline{c}t$

   Seitenebene: $F(P) = \underline{m}(P - B) = 0$

   $\underline{m}$ nach außen gerichtete Normale

   B Stützpunkt der Ebene

   Schnittparameter

   $\underline{m}(S + \underline{c}t - B) = 0$

   $t_{hit} = \frac{Zähler}{Nenner}$

   Zähler: $\underline{m}(B - S)$

   Nenner: $\underline{m} \cdot \underline{c}$

   Überlegung:

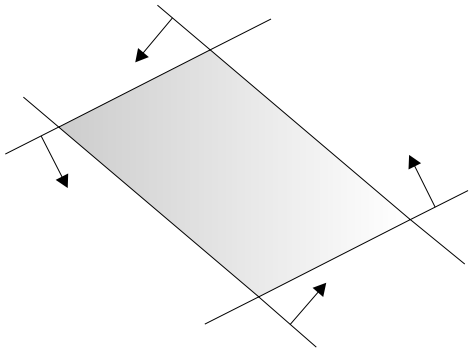
   - Strahl geht in äußeren Halbraum: Nenner>0, d.h. $\measuredangle(\underline{m}, \underline{c}) < 90°$
   - Strahl geht in inneren Halbraum: Nenner<0, d.h. $\measuredangle(\underline{m}, \underline{c}) > 90°$
   - Strahl parallel: Nenner=0

   Klassifikation:

Srahl	**Bedingung**
hinein	Nenner<0
heraus	Nenner>0
völlig innen	Nenner=0 Zähler> 0
völlig außen	Nenner=0 Zähler<0

2. konvexes Polyeder

   Beschreibung durch Stützebenen



Ziel: `hit()`für konvexe Polyeder

Beschreibung: N Stützebenen

i-te Ebene

$\underline{m_i}$ nach außen gerichteter Normalenvektor

B_i Stützpunkt

Algorithmus des Würfels modifizieren

```
for(int i=0;i<N;i++) //für jede Ebene
{
    Zähler=dot3(mi,Bi-S);
    Nenner=dot3(mi,c);
    if(tabs(Zähler)<eps)...
}
```

3. Polygonale Netze

OpenGL: `drawOpenGL()` Prozedur zum Zeichnen polygonaler Netze

Ziel: `hit()` für polygonale Netze
Liste Seitenflächen,Liste Ecken,Liste Normalen

Idee: behandle jede Seite wie Begrenzungsebene

Problem: konkave Körper (nicht korrekt), Einschränkung: konvexe polygonale Netze

`hit()`-Prozedur

Zähler, Nenner in jedem Schritt

benutze 0-ten Eckpunkt der Seite pt[face[f].vert[0].vertIndex] als Stützpunkt

nach außen gerichteter Normalenvektor norm[face[f].vert[0].normIndex] entsprechend einarbeiten in Pseudocode

4. Weitere graph. Primitive

implizite Fläche: F(P)=0

betrachte $F(S+\underline{c}t)$=:d(t) Abstandsfuktion

Diskussion:

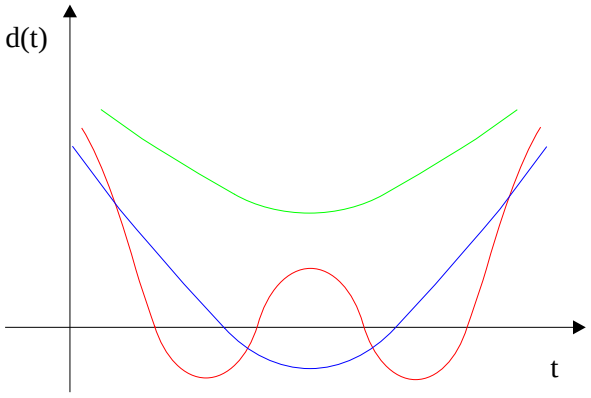
- d(t)>0: P(t) außerhalb
- d(t)<0: P(t) innerhalb
- d(t)=0: P(t) auf F

  Ergebnis: d(t)=0 charakterisiert t-Werte zu Schnittpunkt, d.h. d(t)=0 muss gelöst werden

Beispiel: Torus

$$F(P) = \left(\sqrt{P_x^2 + P_y^2} - d\right)^2 + P_z^2 - 1$$
$$F(S+\underline{c}t) = d(t) = 0$$

führt auf eine Gleichung vierten Grades für t, komplexe Numerik



Beispiel: Superquadric

$$(x^n 1 + y^n)^{\frac{m}{n}} + z^m - 1 = 0$$
$$d(t) = ((S_x + c_x t)^n + (S_y + c_y t)^n)^{\frac{m}{n}} + (S_z + c_z t)^m - 1 = 0$$

Newton-Iteration

Zusammenfassung:

`hit()` kann für hinreichend viele graph. Primitive implementiert werden

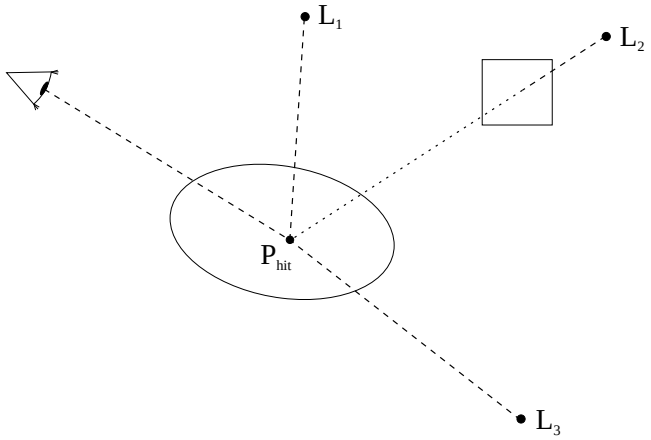
10.5 Rekursives Raytracing

Ziel: besserer Realismus der gerenderten Szene

Raytracing erlaubt mit geringem Zusatzaufwand, Schraffierungs- und Beleuchtungstechniken

Nachteil: erhöhter Rechenaufwand

Berechnung der Lichtintensität: P_{hit} wird von verschiedenen Lichtquellen beleuchtet



P_{hit} wird von weiteren Lichtquellen beleuchtet bzw. verdeckt (nur ambientes Licht der Quelle)

zusätzliche Strukturen: Schattenfühler

Routine `isInShadow()`

true = ein weiteres Objekt liegt zwischen P_{hit} und L

false = anderenfalls

Hierzu: neuer Strahl

$P_{hit} \rightarrow L$ mit $P_{hit}+(L-P_{hit})t$

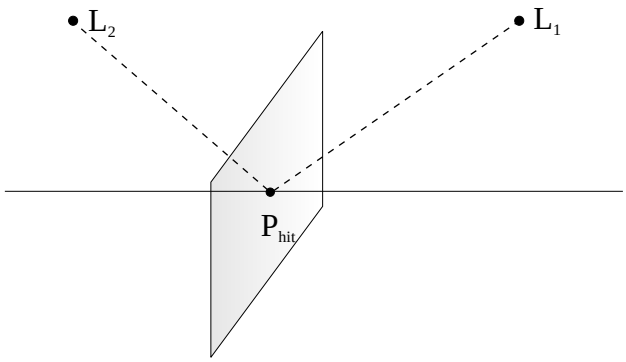
@ t=0: P_{hit}

@ t=1: L

Test:

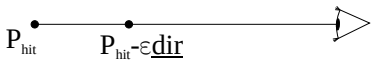
- Objekt der Objektliste wird ausgewählt
- Schnitttest mit Schattenfühler
- Falls $t_{hit} \in [0,1]$ gefunden wird, gib „true“zurück
- anderenfalls „false“

Problem: Schattenfühler mit t_{hit}=0 liefert Probleme



beide Schattenfühler geben „true“zurück

Lösung: Betrachte $P_{hit} - \varepsilon \underline{dir}$



Einbeziehen von Schattenfühlern in Code

`Scene::shade(ray)`

`getFirstHit()` gibt t_{hit} zurück

berechne P_{hit}, Normalenvektor in P_{hit}

ambientes Licht zu zugehörigem Strahl wird berechnet

Schattenfühler in $P_{hit} - \varepsilon \underline{dir}$

recurselevel=1

Lichtquelle L

Richtung des Fühlers: L.pos-feeler.start

`isInShadow(feeler)` aufgerufen

falls Hindernis auftritt, ignoriere diffuses und spekulares Licht der Quelle

Pseudocode:

```
feeler.start=hitPoint-epsilon.ray.dir;
feeler.recurselevel=1;
color=ambientes Teil des Lichtes;
for(jede Lichtquelle L)
{
```

```
    feeler.dir=L.pos-hitPoint;
    if(isInShadow(feeler)) continue;
    color.add(diffuses Licht);
    color.add(spekulares Licht);
}
```

Pseudocode:

```
boot.Scene::isInShadow(Ray& f)
{
    for(GeomObj* P-obj;P;P=P->next)
         if(P->hit(f)) return true;
                 returne false;
}
```

Hinweise:

- Routine scannt alle Objekte der Liste nach Schnitten
- falls kein Schnitt, gibt „false“zurück
- anderenfalls true

Unterschiede zu `hit()`:

- nur $t_{hit} \in [0,1]$ werden akzeptiert
- falls ein $t_{hit} \in [0,1]$ berechnet wird, wird es sofort zurückgegeben, ohne Berechnung von Daten von P_{hit}

Einbeziehen von Reflexion und Transparenz

Vorteil des Raytracing:

erlaubt die Behandlung von Reflexion und Transparenz → realistische Szenen

Nachteil:

mehrfache Reflexionen und Berechnungen erhöhen den Rechenaufwand dramatisch

Beispiel: Folie

Augpunkt in Richtung $\underline{dir}$

P_n Schnitt mit Oberfläche

Lichtintensität

$$I = I_{amb} + I_{diff} + I_{spec} + I_{refl} + I_{trans}(*)$$

Berechnung von I_{refl}

Richtung des reflektierten Strahls -$\underline{r}$

$\underline{r} = \underline{dir} - 2(\underline{dir} \cdot \underline{m})\underline{m}$

$\underline{m}$ = normalisierter Normalenvektor in P_n

Berechnung von I_{trans}

gebrochene Komponente

Richtung -$\underline{t}$ gebrochen in Richtung -$\underline{dir}$

explizite Formel: anschließend

Fakt: I_{refl}, I_{trans} setzen sich wieder wie in (*) aus 5 Komponenten zusammen
$\Rightarrow$ Rekursion des Raytracing

Beispiel: Folie

I = Summe von I_{R_1},I_{T_1},I_{L_1}

I_{L_1} lokale Komponente

I_{L_1} = Summe von I_a,I_d,$I_s$in P_n

lokale Komponenten = kommen direkt vor Lichtquelle

I_{R_1} = Summe von I_{R_3},I_{T_3},I_{L_3}

I_{T_3} = Summe von I_{R_4},I_{T_4},I_{L_4}

Rekursion: Darstellung des Baumes

jeder Knoten addiert lokale Komponente hinzu

Pseudocode:

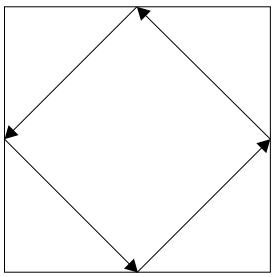
`shade(Ray& r)` aus vohergehendem Kapitel

rekursiver Aufruf

```
shade(refl);shade(trans);
```

- z.B. reflexivity 0.8: Test if(reflexivity>0.6)...
- z.B. transparency: Test if(transparency>0.5)...

Bemerkung: perfekte Spiegel



Einschränkung durch Rekursionslevel

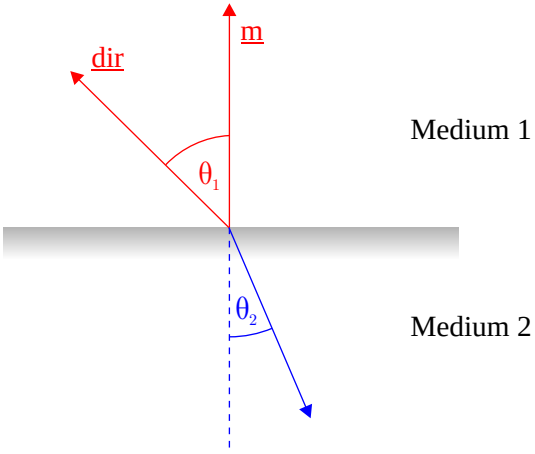
recurselevel = 0 entspricht dem Strahl vom Augpunkt

maxRecursionDepth: 4,5,6 geben sehr realistische Szenen wieder

Zur Lichtbrechung:

Durchgang durch transparente Medien

OpenGL RGBA-Wert,α-blendig charakterisiert Transparenz



Brechung:

- $\underline{t}$ liegt in der durch $\underline{dir}$ und $\underline{m}$ definierten Ebene
- Brechungsgesetz

$$\frac{\sin\theta_2}{c_2} = \frac{\sin\theta_1}{c_1}$$

  wobei c_i, i=1,2, Lichtgeschwindingkeit im Medium i
- es gilt: $\theta_1 = 0 \Leftrightarrow \theta_2 = 0$
- Lichtgeschwindigkeiten in Abhängigkeit der Lichtgeschwindigkeiten im Vakuum

Luft	99,97%
Glas	52%-59%
Wasser	75%
30%-ige Zuckerlösung	72,5%
Alkohol	73,5%
Diamant	41,3%

$\frac{c_1}{c_2}$ Brechungsindex

- kritischer Winkel

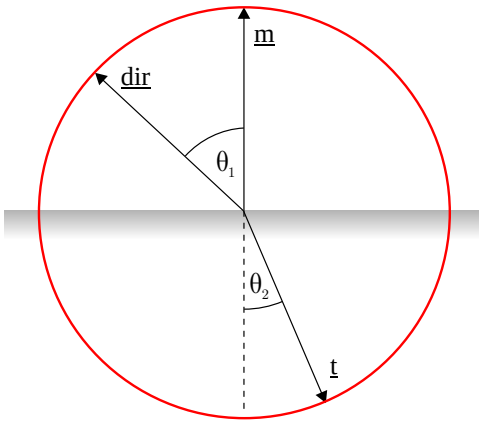
  $\frac{c_2}{c_1} < 1$ Übergang vom „schnellen“zum „langsamen“Medium

  $$\sin\theta_2 = \frac{c_2}{c_1}\sin\theta_1$$

  folgt $\theta_2 < \theta_1$
  kritischer Winkel $\theta_2 \approx 90°$
  Übergang von θ_1 zu $\theta_2 \approx 90°$
  totale innere Reflexion
  Beispiel: Lichtübergang von Wasser zu Luft: $\frac{c_1}{c_2} = 0,75$

  Wie groß ist kritischer Winkel?
  Brechung hängt ab von der Wellenlänge

Aufg.: Richtung des gebrochenen Strahls berechnen



Voraussetzung:

Vektoren$\underline{m}$, $\underline{dir}$, $\underline{t}$ sind normiert

$$\underline{n} = \cos\theta_1\underline{m} + \underline{dir}$$

Länge:

$$|\underline{n}| = \sin\theta_1$$

normieren:

$$\underline{u}_n = \frac{1}{\sin\theta_1}\underline{n} = \frac{\cos\theta_1}{\sin\theta_1}\underline{m} + \frac{1}{\sin\theta_1}\underline{dir}$$

Bestimmen von $\underline{t}$

$$\underline{t} = \sin\theta_2 \underline{u_n} - \cos\theta_2 \underline{m}$$
$$\underline{t} = \frac{c_2}{c_1}\underline{dir} + \frac{c_2}{c_1}(\cos\theta_1 - \cos\theta_2)\underline{m}$$
$$\cos\theta_2 = \sqrt{1-\sin^2\theta_2} = \sqrt{1-\left(\frac{c_2}{c_1}\right)^2 \sin^2\theta_1}$$

Ergebnis:

$$\underline{t} = \frac{c_2}{c_1}\underline{dir} + \left(\left(\frac{c_2}{c_1}\right)(\underline{mdir}) - \cos\theta_2\right)\underline{m}$$
$$\cos\theta_2 = \sqrt{1-\left(\frac{c_2}{c_1}\right)\sin^2\theta_1}$$

Bemerkung: Falls 1-$\left(\frac{c_2}{c_1}\right)^2 \sin^2\theta_1 < 0$, liegt totale Reflexion vor

Prozedur: `transmitDirection()`

Hinweis:

- Raytracing mit transparenten Objekten berücksichtigt nicht unterschiedliche Brechungsindizes für verschiedene Wellenlängen
- alternatives Vorgehen extrem rechenaufwendig, nicht nur drei Farbkomponenten, sondern diskretes Spektrum des Lichtes

Realisierung der Brechung in `shade()`

Ansatz: $\frac{c_2}{c_1}$ muss in jedem Schritt bestimmt werden

Vorgehensweise:

Feld zum Stahl, der Zeiger Objekt hält, durch das der Strahl gerade läuft

Verschiedene Modellierungsansätze

1. keine zwei transparenten Objekte durchdringen sich

   a) sei Strahl außerhalb aller Objekte und vor dem Schnitt mit Objekt A
   A ist „transparent“genug:
   `shade()` berechnet $\underline{t}$, Richtung des gebrochenen Strahls mit c_1=1 für Luft und c_2 vom Objekt A, recurselevel wird erhöht, Strahlzeiger auf A, `shade()` wird rekursiv aufgerufen bis color wiedergegeben wird, danach skaliert mit transparency von A und zu color addiert

b) Strahl innerhalb eines Objektes A, schneidet weiteres Objekt, Strahl trifft auf Luft
⇒ Normale wechselt Vorzeichen, keine amb.,diff.,spekulare Komponente wird berechnet, gebrochener Strahl wird mit c_1 aus Objekt A eingelesen, c_2=1 (falls Einfallswinkel<kritischer Winkel)
neuer Strahl wird definiert (Zeiger auf B)

2. transparente Objekte schneiden sich

komplizierte Strategie:

streckenweise geht der Strahl durch verschiedene Objekte

Liste von Objekten, Rückgabe in welchem Objekt der Strahl läuft

`shade()` komplexe Prozedur

Setzen einer Prioritätsliste

Liste=∅ Strahl außerhalb aller Objekte

zu jedem Zeitpunkt ist der Strahl innerhalb einer Kollektion von Objekten, die von der Liste zurückgegeben wird

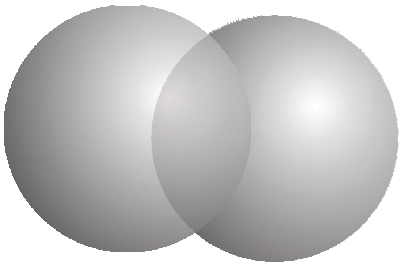
Entsprechend a),b) aus 1. modifizieren

10.6 CSG und Raytracing

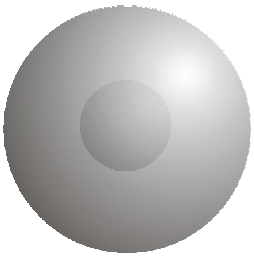
CSG = constructive solid geometry

CSG erlaubt Konstruktion abgeleiteter Primitive mit Hilfe Boolscher Operatoren

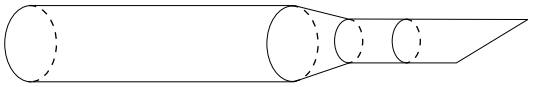
Beispiel:



Schnitt zweier Kugeln: $S_1 \wedge S_2$ Linse

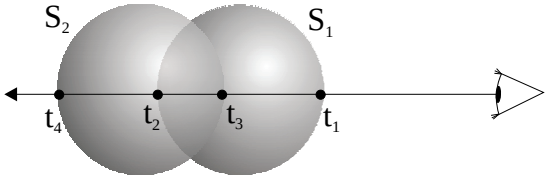


$B = (S_1 \setminus S_2) \setminus C$



$C_1 \cup C_2 \cup C_3 \cup C_4$ Vorteil von CSG: wird durch Raytracing unterstützt

Raytracing für zusammengesetzte Objekte



Prozedur `hit()` gibt

für S_1 t_1,t_2 zurück

für S_2 t_3,t_4 zurück

Strahl innerhalb der Linse [t_2,t_3], nur echter Schnitt falls $t_3 < t_2$

entsprechend: übrige Beispiele

Ziel: Erweiterung des Raytrace-Algorithmus zur Behandlung von CSG-Objekten

Vorgehensweise:

- zwei Objekte A,B
- betrachte festen Strahl
- bilde Listen t_{hit} für A,B
- Konstruktion von Listen t_{hit} für A∪B,A∩B,A\B,B\A

  T=Innenmenge eines Objektes, also T(A),T(B)
- bilde T(A∘B)=T(A)⊙T(B) für $\odot = \cup, \cap, \setminus$

Zusammenfassung:

Innenmenge der CSG-Objekte wird aus den Innenmengen der Basisobjekte mit Hilfe der boolschen Operationen gebildet

Hilfsmittel:

- `hit()` für jedes Objekt trägt alle Informationen zu den Schnittdaten
  inter.hit[] Array, zeitsortierte Liste

- erster positiver t_{hit}-Wert der kombinierten Liste gibt ersten Schnitt für das CSG-Objekt zurück

Datenstrukturen: Folie

$$((B_1 \cup B_2) \cup C_2) \cup ((S_1 \setminus S_2) \setminus B_3) \setminus C_1$$

Syntax-Baum:

innerer Knoten repräsentiert einen Operator

Blatt: geometrische Primitive

Baum: Kombination von Teilbäumen (nicht eindeutiger Baum)

Implementierung:

`sphere, cube` abgel. Klasse von `shape`

`shape` abgeleitet von `GeomObj`

Idee: neue Klasse

`Boolean` abgeleitet von `GeobObj`

`BooleanObj` Binärbaum mit Zeiger links bzw. rechts auf `GeomObj`

Vorgaben:

```
class Boolean::public GeomObj
{
    GeomObj *left, *right;                          //Zeiger auf Kinder
    Boolean() { left=right=Null; }                  //Konstruktor
    virtualboolhit(Ray& r, Intersection& inter);
     ...
}
```

Hinweise:

- spezielle `hit()`-Routine für `Booleans`, Schnitt eines Strahls mit `Booleans`
- verschiedene Klassen

  `UnionBool, IntersectionBool, DifferenceBool`
  z.B.

```
class UnionBool::public Boolean
{
    public:
        UnionBool() { Boolean(); }
        virtualboolhit(Ray& r,Intersection& inter);
          ...
}
```

Bemerkungen:

`Booleans` besitzen keine eigene affine Transformation, nur shapes

Transformation für `Booleans` sind auf den Blättern lokalisiert

Schnitt von Strahlen mit boolschen Objekten

Skizze: `hit()` für CSG-Obj.

Lauf über Baum

- Schnitt über linken Teilbaum
- Schnitt über rechten Teilbaum
- Kombination beider Teilbäume entsprechend der boolschen Operation

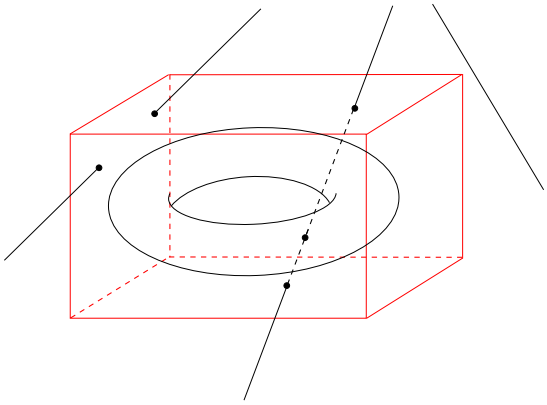
Ergebnis: Raytracing unterstützt CSG-Objekte

10.7 Beschleunigung für Raytracing

Raytracing: oft sich wiederholende Techniken, Schnittberechnungen

Ziel: Suche nach Techniken zur Reduktion der Berechnungen

Begriff: Umgebung eines Objekts = graph. Primitiv, das Körper vollständig umgibt



Idee:

- Falls ein Strahl die Umgebung nicht schneidet, kann er Objekt ebensowenig schneiden
- Umgebung mit einfacher Gestalt liefert billigen Schnitttest

Beispiel: Torus mit Quaderumgebung

Schnittverhalten eines Strahls

- Strahl trifft die Umgebung nicht
- Strahl trifft die Umgebung aber nicht den Torus (falscher Alarm)
- Strahl trifft Umgebung und den Torus

Ansatz: Schnitttest für Objekt im Allgemeinen viel teurer als Schnitttest für Umgebung

Kostenanalyse:

Schnitt mit Umgebung: T Zeiteinheiten

Schnitt mit Torus: mT Zeiteinheiten

N Strahlen für Rendering erforderlich

f Anteil der Strahlen, die den Quader trifft

D.h.

N Tests mit Umgebung mit NT Kosten
fN Tests mit Torus mit $fNmT$ Kosten
Ergebnis: totale Kosten $NT(1+fm)$
Alternative: ohne Umgebungstest
N Tests mit NmT Kosten
Verhältnis:

$$\frac{KostenohneUmgebungstest}{KostenmitUmgebungstest} = \frac{m}{1+fm}$$

Beispiel: m=20, f=$\frac{1}{40}$

Objekt einer typischen Szene überdeckt nur einen kleinen Teil der Gesamtszene

$$\Rightarrow Verhältnis = \frac{20}{1+\frac{1}{40}20} = \frac{40}{3} \approx 13$$

13-fache Beschleunigung durch Umgebungstests

Frage: Einbau von Umgebungstests in Raytracing-Algorithmus

verschiedene Ansätze:

hier Modifikation von `hit()`

`getFirstHit()` testet den aktuellen Strahl gegen jedes Objekt mit `hit()`-Prozedur für spezielles Objekt

Vorgehensweise:

erweitere Test in `hit()` zur Beschleunigung für jedes Objekt

Ziel: schnelles Entscheiden, ob der Strahl die Umgebung des Objekts schneidet, bevor langwierige Berechnungen des t_{hit} vorgenommen werden

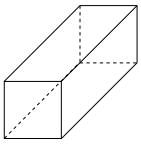
Alternativen:

- leicht zu implementierende Verfahren
- komplexe Ansätze mit hoher Performance

hier: Quader- und Kugelumgebungen

Kugelumgebung: c Mittelpunkt, r Radius

Quaderumgebung: Spezifikation



l,b,r,t

Bemerkung:

- Schnittberechnung eines Strahls mit einem Quader (achsenparallel) ist effektiv, 6 Ebenen, Testintervall t_{in}, t_{out}
- Schnittberechnung mit Kugelumgebung, quadr. Gleichung und Test mit Vorzeichen der Diskriminante

Frage: Umgebungstest in Weltkoordinaten oder generischen Koordianten?

Gegenüberstellung:

- generische Koordinaten:

  transf. Strahl in gen. Koordinaten schneidet gen. Strahl mit gen. Objekt bzw. der gen. Umgebung

  falls der Strahl Umgebung schneidet, besteht keine Notwendigkeit zur Berechnung des generischen Strahls

  Umgebung um generisches Objekt legen und testen, falls Test positiv, `hit()` weiterverfahren

- Weltkoordinaten:

in Weltkoordinaten lassen sich Umgebungen nicht besonders gut einfügen (rechenintensiv)

Ergebnis:

Kompromiss zur Benutzung von Welt- bzw. gen. Koordinaten

- Testen in Weltkoordinaten auf Schnitt = schnell, inverse Transformation nicht erforderlich
  Umgebung in Weltkoordinaten schwierig zu berechnen
- Testen in gen. Koordinaten = erfordert Berechnung des gen. Strahls (teurer), Umgebung in gen. Koordinaten: einfach abgelegt

Frage: Kugel- oder Quaderumgebungen?

schwierige Entscheidung, hängt von der Szene ab

z.B.: 100 zufällig generierte Kegelstümpfe, verringert Kugelumgebung Raytracing um 27%

Beispiel:

Umgebungstest für Kegelstumpf

Kugelumgebungen in Welt- und gen. Koordinaten

(Raytracer wird natürlich nur einen Test benutzen)

Pseudocode: Folie

Bemerkungen zum Pseudocode:

- zuerst Aufruf in Weltkoordinaten, falls Strahl Umgebung schneidet, wird die inverse Transformation aufgerufen
- zweiter Aufruf für gen. Koordinaten
- Modifikation für `rayHitsBoxExtend` für Quaderumgebungen ist möglich
- Auswahl der Tests in `hit()` für unterschiedliche Flächenklassen, z.B. für Kugeln wird man keine Kugelumgebung wählen
- es gibt keine Umgebung für Ebenen
- Quadrat besitzt Umgebung, aber Test einfach, dass man auf Umgebung verzichtet
- Polygonnetze $\Rightarrow$ Vielzahl von Polygonen, Umgebungstest bringt große Beschleunigung

Zur Implementierung von Umgebungstests

Frage:

- Wie bildet man Umgebungen für jede geom. Form?

- Wie wird der Test durchgeführt?

Antwort: Klasse `GeomObj`

Hinzufügen von

```
SphereInfo, genSphereExtent, WorldSphereExtend
Cuboid, genBoxExtent, WorldBoxExtend
```

Anwendug:

- nachdem Objekte gebildet sind, vor dem Raytracing, werden Daten für jedes Objekt in Objektlisten eingetragen
- während des Raytracing werden Kombinationen von

```
rayHitsSphereExtent()
rayHitsBoxExtent()
```

aufgerufen

Konstruktion von Kugel- und Quaderumgebungen

Aufgabe: Angabe von Kugel- bzw. Quaderumgebungen für Objekt

Unterschied:

- gener. Koordinaten: einmal für jeden Typ
- Weltkoordinaten: für jedes Objekt separat
  z.B. Wie findet man Kugelumgebung eines Zylinders, der skaliert, verschoben und gedreht wurde?

Idee: Punkt-Cluster für jedes Objekt

= Menge von Punkten, deren konvexe Hülle das Objekt umschließt

Punkt-Cluster liefert Umgebungssphäre bzw. Umgebungsbox

Konstruieren von Punkt-Clustern:

- für Quader und Quadrate ist Punkt-Cluster kein Problem
- Polygonnetz: leicht
  Punkt-Cluster ergibt sich aus der Eckenliste (=Eckenliste)
- Kegelstumpf, Sphäre: Hilfsmittel Symmetrieüberlegungen

Datenstruktur:

```
PointCluster
```

num = Feld für die Anzahl

pt[] = Array mit den Punkten

`makeExtentPoints()`

Annahme: Punkt-Cluster ist verfügbar

Gesucht: Umgebung

- Berechnung der Quaderumgebung

$$left = \min_i(pt[i].x)$$
$$right = \max_i(pt[i].x)$$

usw.

- Berechnung der Kugelumgebung

Berechnung des Schwerpunktes = Mittelpunkt

Radius = längster Abstand vom Mittelpunkt zu Punkten des Clusters

Kugel- und Quaderumgebungen in gen. Koordinaten einmal berechnen und ablegen

bisher: Umgebungen im Objektraum betrachtet

Projektionsumgebungen

alternative Umgebungsobjekte mit enormem Beschleunigungseffekt

Idee: rechteckige Umgebung auf dem Bildschirm

Projektionsumgebung:

Spezifikation: {r,l,t,b}

Vorteil: einfache Benutzung von Projektionsumgebungen

Beschleunigung des Raytracing

Umgebung eines Objekts

- Umgebung im Bildraum
- Umgebung im Projektionsraum

Idee: rechteckige Umgebung

Vorteil:

- einfache Handhabung von Projektionsumgebung
- betrachtet (r,c)-Strahl der Schirm passiert
- falls (r,c)-Strahl Umgebung nicht schneidet, wird er nicht weiter behandelt

Nachteil:

Anwendung der Projektionsumgebung erfolgt nur für Augstrahl

rekursives Raytracing wird nicht unterstützt

Konstruktion einer Projektionsumgebung

- vorbereitender Prozess:

  `IntRectScreenExtend` wird geometr. Objekt zugeordnet

- Erweiterung der ray-Klasse um (r,c)-Durchgangspixel auf Schirm
- Hinzufügen eines Parameters

  Rekursionstiefe erlaubt Klassifizierung eines Strahls als Aug-Strahl

  Augstrahl = level = 0

- neuer Test am Beginn der Prozedur `hit()` für jeden Objekttyp, z.B. Kegelstumpf siehe Folie

Aufgabe: Berechnung der Projektionsumgebung

- Hilfsmittel: Punkte-Cluster
  → Punkte-Cluster des generischen Objektes wird in Weltkoordinaten transformiert mit Hilfe der Objekttransformation
- Zwischenergebnis:
  - Punktewolke P[0], P[1], ...
  - P[i] wird auf die Near Plane der Kamera transformiert
  - P Punkt der Szene
  - P' Ortsvektor des Augpunktes
  - P wird auf Schirm projiziert
    (r,c)-Pixel
- Ergebnis:

$$r = \frac{nRows}{2}\left(1 - \frac{NP_v}{HP_n}\right)$$

$$c = \frac{nCols}{2}\left(1 - \frac{NP_u}{HP_n}\right)$$

$$P_u = (P - P')\underline{u}$$

$$P_v = (P - P')\underline{v}$$

$$P_n = (P - P')\underline{n}$$

$\underline{u}, \underline{v}, \underline{n}$ Koordinatenvektoren des Kamerakoordinatensystems

Strahl

$$r(t) = P' + \underline{dir}_{rc} t$$
$$\underline{dir}_{rc} = N\underline{n} + W\left(\frac{2c}{nCols} - 1\right)\underline{u} + H\left(\frac{2r}{nRows} - 1\right)\underline{v}$$

Schnitt

$$K(P - P') = \underline{dir}_{rc}$$

Auflösung nach (r,c), Berechnung für alle i

Ergebnis: P[i] wird projiziert (r_i,c_i)

Suche über (r_i,c_i)

kleinstes und größtes r_i

kleistes und gröstes c_i

$\Rightarrow$ {l,r,t,b} Projektionsumgebung

$\rightarrow$ Bild Folie

10.8 Schraffierung und Raytracing

Lichtmodell

$$I = I_a\rho_a + I_d\rho_d \times lambert + I_s\rho_s \times (phong)\rho$$
$$lambert = max\left(0, \frac{\underline{s}\cdot\underline{m}}{|\underline{s}||\underline{m}|}\right)$$
$$phong = max\left(0, \frac{\underline{h}\cdot\underline{m}}{|\underline{h}||\underline{m}|}\right)$$

f = ganzzahliger Exponent

$\underline{h} = \underline{s} + \underline{v}$ Halbweg-Vektor

$\rightarrow$ Aufsummieren verschiedener Lichtquellen

$\rightarrow$ Aufspalten in jede RGB-Komponente

Hier:

- P_{hit} berechnen

- Kenntnis $\underline{h}, \underline{s}, \underline{m}$

zur Berechnung von $\underline{m}$:,

- $\underline{m}$ ergibt sich einfach in generischen Koordinaten

  $\Rightarrow$ Transformation in Weltkoordinaten $\underline{m}' = (M^{-1})^T \underline{m}$

Vorgehensweise:

- Normalenvektor von P_{hit} in generischen Koordinaten
- Transformation gibt Normalenvektor in Weltkoordinaten zurück

Oberflächenmodellierung

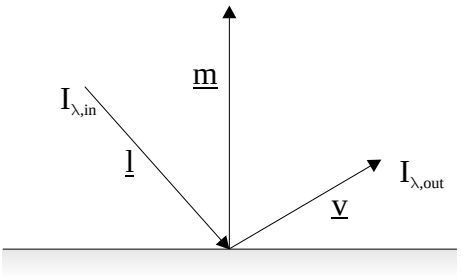
- reflektiertes Licht = Kombination von amb., diff., spek. Komponenten
- Modellierungsparameter $\rho_{a_x}, \rho_{d_x}, \rho_{s_x}$ jeweils für x=r,g,b
- Übergabe von Materialparametern mtrl.ambient usw.

Alternatives Lichtmodell: Cook Torrance Phong-Schraffierung hat Nachteile: → Objekte erscheinen plastikartig

Ansatz von Cook-Torrance:

- Objekte bestehen aus einer Vielzahl von kleinen Mikrofacetten
- stärkere Berücksichtigung der Oberflächenstruktur

Bidirektionale Betrachtung



$\underline{l}, \underline{m}, \underline{v}$ sind Einheitsvektoren

Bidirektionale Reflexionsfunktion:

BRIDF = bidirectional reflected intensity function

mit Parameter

- Einfallsrichtung des Lichtes $\underline{l}$

- Richtung des Betrachters $\underline{v}$
- Wellenlänge λ
- Eigenschaften der Fläche (Normalenvektor $\underline{m}$

→ Ansatz:

$$BRIDF(\underline{l}, \underline{v}, \lambda) = \frac{I_{\lambda,out}}{I_{\lambda,in}}$$

(→ Verhältnis beschreiben)

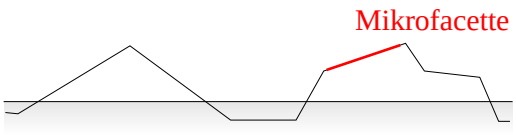
→ Vorteil:

- beliebige Einfalls- und Betrachtungsrichtungen
- realisiert rauhe Oberflächen mit Rinnen

→ Lösung:

- physikalische Simulation, teuer und zeitaufwendig
- math. Modell analog zum Phong-Modell

→ Hier: math. Modell nach Cook-Torrance



→ Ansatz von Cook-Torrance

- feste Wellenlänge

$$I = I_a + I_d + I_s = \underbrace{\rho_a I_a^{in} + \rho_d I_d^{in}}_{\text{wie bisher}} + \rho_s I_s^{in}$$

- neu:

$$I_s = \frac{\underline{m} \cdot \underline{l}}{\underline{m} \cdot \underline{v}} \cdot S \cdot F \cdot G \cdot D \cdot I_s^{in}$$

hierbei s=skalar, Helligkeit, F,G,D skalare Faktoren

→ Begründung für den Ansatz:

$$\frac{I_s(\underline{m} \cdot \underline{v})}{I_s^{in}(\underline{m} \cdot \underline{l})}$$

= Verhältnis von reflektierter zu einfallender Intensität

→ Rolle der skalaren Faktoren:

1. D=D($\underline{l},\underline{v}$) = Maß für Verteilung der Mikrofacetten = Verhältnis von korrekt reflektiertem zu tatsächlich reflektiertem Licht von $\underline{l}$ nach $\underline{v}$
   Verteilungsfunktion:

$$\sigma = \frac{\theta - \varphi}{2}$$

$$D(\sigma) = \frac{1}{4m^2 \cos^2 \sigma} exp\left(-\left(\frac{\tan\sigma}{m}\right)^2\right)$$

   (Normalverteilung)

   m=Maß für Rauheit

   m≈ 0.2 glatt

   m≈ 0.6 rauh

2. G=G($\underline{l},\underline{v}$) = Maß für die Verringerung der Reflexion der Mikrofacetten, die durch Schatten bzw. Verdeckung gebildet wird
   a) gesamtes einfallendes Licht wird reflektiert, dann G=1
   b) Teil des reflektierten Lichtes wird in die Facette gestreut, G_s
   c) Teil des Lichtes fällt nicht in die betrachtete Facette G_m

   Festsetzung $G := min(1, G_m, G_s)$

   Wie betrachtet man G_m,G_s?
   Antwort:

$$G_s = 2\frac{(\underline{mh})(\underline{mv})}{\underline{hv}}$$

$$G_m = 2\frac{(\underline{mh})(\underline{ms})}{\underline{hs}}$$

3. F=Fresnel-Koeffizient

   Ansatz: Fresnel-Koeffizient beschreibt welcher Anteil des Lichtes in Richtung des Betrachters spekular reflektiert wird
   siehe Bilder Folie

   - Gleichung:

$$F(\underline{l},\underline{v},\lambda) = \mathscr{F}(\varphi,\eta)$$
$$\varphi = acos(\underline{l},\underline{h})$$
$$\eta = Brechungsindex$$

   - Physik:

$$\mathscr{F}(\varphi,\eta) = \frac{1}{2}\left(\frac{g-c}{g+c}\right)^2\left(1+\left(\frac{c(g-c)-1}{c(g+c)+1}\right)^2\right)$$

mit

$$c = \cos\varphi, g = \sqrt{\eta^2 + c^2 - 1}$$

- Hinweis: η variiert mit Wellenlänge
- Bemerkungen:
  - Berechnung von η
  - rechtwinklig einfallendes Licht φ=0, c=1, g=η

$$F_0 = \mathscr{F}(\varphi, \eta) - \frac{(\eta - 1)^2}{(\eta + 1)^2}$$

Auflösen:

$$\eta = \frac{1 + \sqrt{F_0}}{1 - \sqrt{F_0}}$$

→ Cook Torrance ist viel teurer als Phong, liefert aber besseren Realismus

Zusammenfassung:

- Raytracing ist konzeptionell einfacher, uniformer Zugang für fotorealistisches Rendern
- Raytracing = Strahlverfolgung mit rekursivem Aufruf (Lichtquellen, Brechung, Reflexion)
- Features:
  - Elimination verdeckter Flächen wird realisiert
  - Schattenbrechung
  - Einbeziehung von Materialeigenschaften
- Raytracing mit Basisobjekten
  (umfangreiche Schnittprozedur) komplexe Szenen mit CSG-Techniken
- Beschleunigungsverfahren durch Umgebungen in Objekt- bzw. Projektionsraum
- zusätzlich Cook-Torrance-Beleuchtung

Literaturempfehlungen: [12]

Literaturverzeichnis

[1] James D. Foley, Andries VanDam u.a. *Computergraphics: Principles and Practice.* Addison Wesley, 1989. (Nachschlagewerk)

[2] Edward Angel. *Interactive Computer Graphics.* Addison Wesley, 2002. (Vorlesungsbegleitend)

[3] Hans-Joachim Bungartz, Michael Griebel, Christoph Zenger. *Einführung in die Computergraphik.* Vieweg, 2002. (Lernen!)

[4] Alan Watt. *3D-Computergrafik.* Pearson Studium, 2001 (deutsch). (Nachschlagewerk, Übersetzung ins Deutsche)

[5] Alan Watt. *3D-Computer Graphics.* Addison Wesley, 1994.

[6] Jackie Neider, Tom Davis, Manson Woo. *OpenGL Programming Guide.* Addison Wesley, 1993.

[7] http://www.opengl.org

[8] Rainer Barth, Ekkehard Beier, Bettina Pahnke. *Graphikprogrammierung mit OpenGL.* Addison Wesley, 1996.

[9] Ute Claussen. *Programmieren mit OpenGL.* Springer-Verlag, 1997.

[10] Gerd Fischer. *Analytische Geometrie. Eine Einführung für Studienanfänger* . Vieweg, 2001.

[11] Franco P. Preparata, Michael I. Shamos. *Computational Geometry. An Introduction (Monographs in Computer Science)* . Springer, 1999.

[12] Andrew Glassner. *An introduction to raytracing.* Academic Press, 1989.

[13] Francis S. Hill. *Computer Graphics Using OpenGL.* Prentice Hall, 2000.