Codierung von Grafik-Modellen

Speicherung eines Grafik-Objekts (z.B. eines Würfels):

```
cube.cgf ; Objekt-Name - CGF Version 0.0
8 ; Anzahl Punkte
-1., -1.,  1. ; Koord. Pkt[0] etc.
 1., -1.,  1.
 1.,  1.,  1.
-1.,  1.,  1.
-1., -1., -1.
 1., -1., -1.
 1.,  1., -1.
-1.,  1., -1.
6 ; Anzahl Flaechen
4, 4, 4, 4, 4, 4 ; Pkte je Flaeche
0, 1, 2, 3 ; Pkt-Idx ab Flaeche[0]
1, 5, 6, 2
2, 6, 7, 3
3, 7, 4, 0
4, 7, 6, 5
5, 1, 0, 4
```

Geometrie der Objektpunkte (ihre räumliche Lage)

Topologie der Objektpunkte (Beziehungen zwischen ihnen)

```
        (7)-----(6)
       / |     / |
      /  |    /  |
     /  (4)--/--(5)
(3)------(2)   /
 |    /    |  /
 |   /     | /
(0)------(1)
```

(Xface?)

Codierung von Grafik-Modellen

Anmerkung zur Codierung von Grafik-Objekten (z.B. Würfel):

Grundsätzlich zwei technisch realisierbare Codierungs-philosophien für die (4D-)Flächenecken (z.B.: **`facePnt`**):

1. Individuelle Speicherung aller Flächenecken-Koord., d.h. (6 Flächen x 4 Punkte/Fläche x 4Koord./Punkt=) 96 Werte:

```
float facePnt[6][4][DIM];
```

```
#define DIM 4
```

⇨ Unnötig erhöhter Speicherbedarf

Bei Formänderungen (z.B.: Erhöhung einer Kuppelspitze) ist zu berücksichtigen, daß jeder Objektpunkt in mehreren (beim Würfel: in 3) Flächen vorkommt:

⇨ Unnötig erhöhter Rechenzeitbedarf

⇨ Kaum wartbare Fehler-Quelle

➥ nicht gebräuchlich, nicht zu empfehlen (quick & dirty)

2. Alternative: Trennung von Objekt-Geometrie u. -Topologie

Codierung von Grafik-Modellen

2. Geometrie-Codierung von 3D-Objekten durch (einmalige) Speicherung der Koordinaten – z.B. Würfel:

```
float cubVrtx[8][DIM];          /*32 Koordinaten*/
```

Zwei gebräuchliche Verfahren zur Topologie-Erfassung:

2a) Erfassung der Punkt-Indizes für jede Fläche – z.B.:

```
int facePnt[6][4];                  /*24 Indizes*/
```

Bsp. Zuordnung Flächen- / Objektpunkt:

```
facePnt[5][3]=7;  /*4.Pkt d.6.Flaeche≡8.ObPkt*/
```

⇨ Unhandliche, fehleranfällige Ausdrücke, z.B.:

```
float x;  /*...*/  x=cubVrtx[facePnt[5][3]][0];
```

➥ Gebräuchlich eher in„Rapid-Prototyping“-Programmen

2b) Erfassung der Punkt-Adressen für jede Fläche – z.B.:

```
float *facePnt[6][4], x; /*...*/
facePnt[5][3]=cubVrtx[7];    x=facePnt[5][3][0];
```

➥ Intuitive, übersichtliche, robuste Ausdrücke

Beispiel: „Harte“ Codierung eines Würfels

```
#include <conio.h>
#include <stdio.h>
```

```
int main()
{ int     j1=0;
  float  cubVrtx[8][DIM]; //Koordinaten der Wuerfelecken
  float *facePnt[6][4];  //Flaechen-Def.: Punkt-Adressen
  /*Koordinaten-Zuweisung fuer Wuerfel-Punkt (z.B.):*/
  for(j1=0;j1<DIM;j1++) cubVrtx[7][j1]=4.f-j1;//=4,3,2,1
  /*Verknuepfung Flaechen-Ecke / Wuerfel-Punkt (z.B.):*/
  facePnt[5][3] = cubVrtx[7];  /* = &cubVrtx[7][0];   */
  for (j1=0; j1<DIM; j1++)     /*Koordinaten-Ausgabe: */
  { printf ("\n\r facePnt[5][3][%d]=%5.2f"
     " = cubVrtx[7][%d]=%5.2f",
     j1, facePnt[5][3][j1],
     j1, cubVrtx[7][j1]);

     printf (" =%5.2f \n\r",*(*(*(facePnt+5)+3)+j1));
  } _getch();  return 0;  }
```

C:\WINDOWS\system32\cmd.exe

```
facePnt[5][3][0]= 4.00 =cubVrtx[7][0]= 4.00 = 4.00
facePnt[5][3][1]= 3.00 =cubVrtx[7][1]= 3.00 = 3.00
facePnt[5][3][2]= 2.00 =cubVrtx[7][2]= 2.00 = 2.00
facePnt[5][3][3]= 1.00 =cubVrtx[7][3]= 1.00 = 1.00
```

s.Projekt: 01OObjCod

Codierung von Grafik-Modellen

- Besonders interessanter Spezialfall: Zuweisung von phys. Eigenschaften (z.B. Farbe) einem bestimmten Objektpunkt

  ⇨ Ermittlung des Objektpunkt-Index aus der Objpkt.-Adresse

  6 Startadressen (**`facePnt[i]`**) von Eckpunkt-Listen mit je 4 Startadressen (**`facePnt[i][j]`**) von Startadressen (**`cubVrtx[k]`**) von Koordinaten-Listen

  Beispiel:

```
#define DIM 4
float  cubVrtx[8][DIM];
float *facePnt[6][4]; /*...*/
facePnt[5][3]= facePnt[4][1]= cubVrtx[7];
```

  Gemeinsamkeit aller Instanzen **`facePnt`** eines Objektpunktes **`cubVrtx`** auf unterschiedlichen Flächen:

  Adressen-Abstand zu **`cubVrtx[0]`** – z.B.:

```
printf("Idx=%d\n",(facePnt[5][3]-cubVrtx[0])/DIM);
printf("Idx=%d\n",(facePnt[4][1]-cubVrtx[0])/DIM);
```

  (Ausgabe beider Anweisungen: „**`Idx=7`**“)

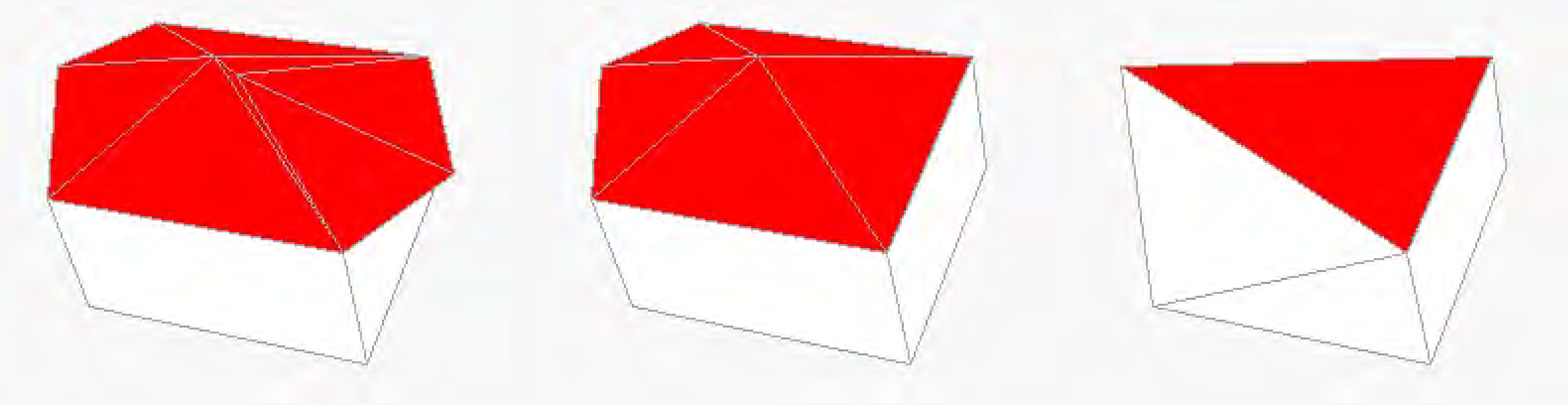
s.Projekt: `ObjCod`

Codierung von Grafik-Modellen

Veränderte Aufgabenstellung in der Praxis:

- Anzahl von Punkten u. Flächen erst beim Laden bekannt (⇨ Speicherplatz-Belegung zur Laufzeit)
- Anzahl von Punkten pro Fläche zudem i.d.R. ungleich (⇨ z.T. verkettete Listen statt Feldern)
- Je nach Anwendung:

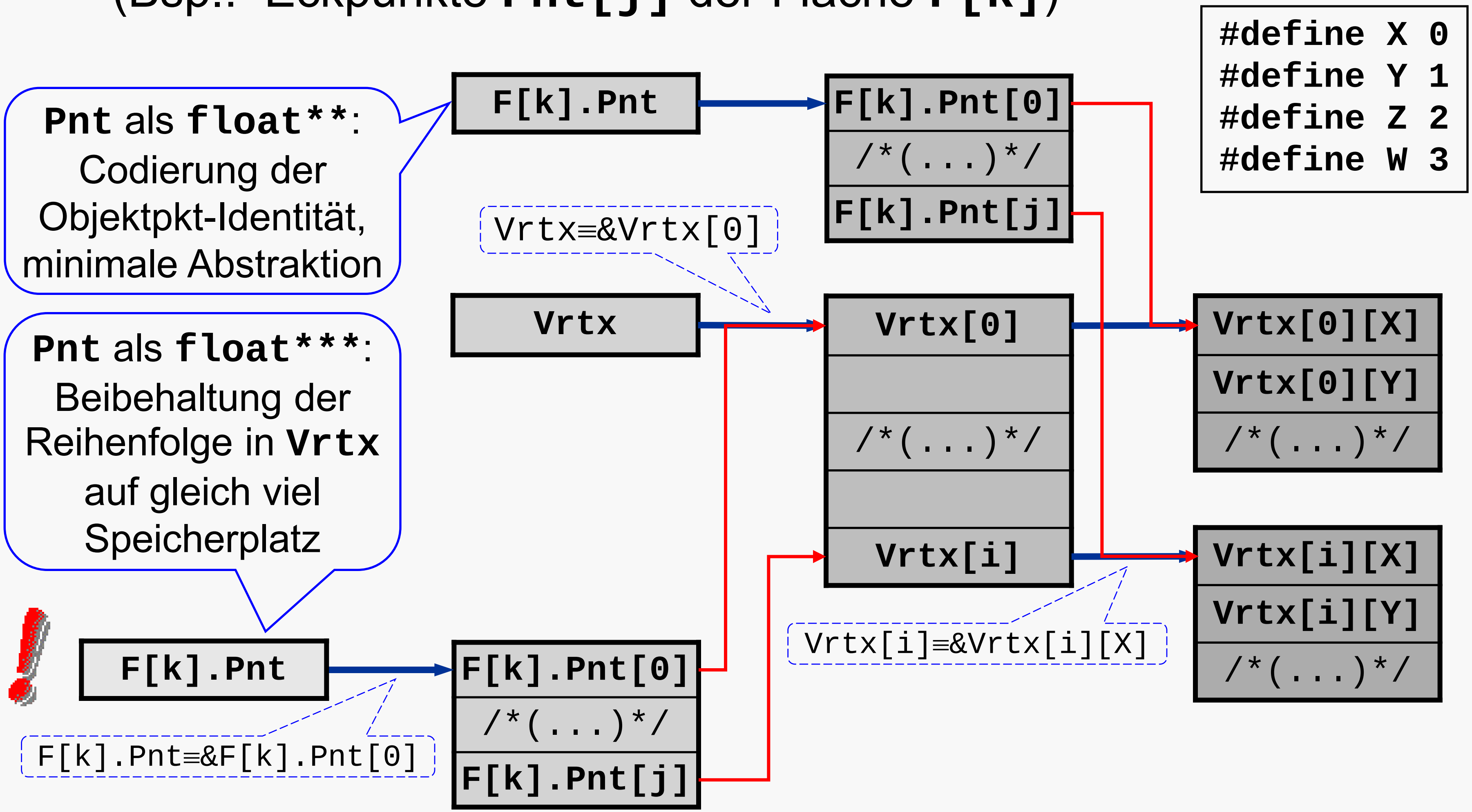
  Anzahl von Objektpunkten und -flächen, aber auch Identität von Flächen-Eckpunkten variabel



Codierung von Grafik-Modellen

Codierung d. Objekt-/Eckpkt-Zuordnung prinzipiell wählbar:

(Bsp.: Eckpunkte **Pnt[j]** der Fläche **F[k]**)



Codierung von Grafik-Modellen

Codierung von Geometrie (**`Vrtx`**) u. Topologie (**`Pnt`**) in der Praxis:

- Speicherreservierung für Objektpunkte je nach Objektgröße:

```
Vrtx = (float**) calloc (nVrtx,sizeof(float*));
for (j=0; j<nVrtx; j++)
  Vrtx[j] = (float*)calloc(NDIM,sizeof(float));
```

- Speicherreservierung für Flächenpkte je nach Codiergskonzept: Flächenpunkt als **`float***`** vs. *`float**`*:

```
for (j1=0; j1<nFace; j1++)
{ Face[j1].Pnt =
 (float***)calloc(Face[j1].nPnt,sizeof(float**));
  (float**)calloc(Face[j1].nPnt,sizeof(float*));
 for (j2=0; j2<Face[j1].nPnt; j2++) { /*idx=...*/
    Face[j1].Pnt[j2] = &Vrtx[idx];
    Face[j1].Pnt[j2] =  Vrtx[idx]; }
}
```

Allgemeine programm-interne Darstellg. eines 3D-Objektes:

```
typedef struct
{ int       nPnt;   //Anzahl Eckpunkte
  float ***Pnt;    //Punkt-Indizes
  char      symbol;//Zeichn f.Konsole
 } Polygon;
```

```
typedef char  String[LENGTH];
```

stellv. für Flächen-Eigensch.: Farbe, Textur, Rauhigkeit,...

```
#define DIM       4
#define LENGTH   80
```

Als Zeiger: Größen, die zur Compilierungszeit nicht bekannt sein können

```
typedef struct
{ String   Name;           //Obj.-Name, Header
  int      nVrtx;          //Anzahl Objektpkte
  int      nFace;          //Anzahl Flaechen

  float  **Vrtx;           //Objektpunktkoord.

  Polygon *Face;           //Flaechenliste

  float    posMat[DIM*DIM];//Positionsmatrix
} CGFobject;
```

Übung: Codieren und Laden grafischer Modelle

- Codierung eines Dreieck-Modells Speicherplatz-Reservierung, Belegung der Struktur, Erweiterung um Rückseite
- Routine zum Laden aus formatierter Datei

```
C:\WINDOWS\system32\cmd.exe
Objekt-Name: Triangle CGF Version 0.1
Objektpunkt-Koordinaten:
   -1.00   -1.00   -1.00    1.00
    1.00   -1.00   -1.00    1.00
    0.00    1.00   -1.00    1.00


Flaechen-Punkt-Indizes u. -Koordn. (2 Flaeche/n):
Flaeche[0] (3 Punkte):
(obj.Face[0].Pnt[0] - &obj.Vrtx[0])/(&obj.Vrtx[1] - &obj.Vrtx[0])==  0
Flaeche[0] Eckpkt[0] = Obj.-Pkt[0]:
   -1.00,    -1.00,    -1.00
(obj.Face[0].Pnt[1] - &obj.Vrtx[0])/(&obj.Vrtx[1] - &obj.Vrtx[0])==  1
Flaeche[0] Eckpkt[1] = Obj.-Pkt[1]:
    1.00,    -1.00,    -1.00
(obj.Face[0].Pnt[2] - &obj.Vrtx[0])/(&obj.Vrtx[1] - &obj.Vrtx[0])==  2
Flaeche[0] Eckpkt[2] = Obj.-Pkt[2]:
    0.00,     1.00,    -1.00
Symbol: █


Flaeche[1] (3 Punkte):
(obj.Face[1].Pnt[0] - &obj.Vrtx[0])/(&obj.Vrtx[1] - &obj.Vrtx[0])==  2
Flaeche[1] Eckpkt[0] = Obj.-Pkt[2]:
    0.00,     1.00,    -1.00
(obj.Face[1].Pnt[1] - &obj.Vrtx[0])/(&obj.Vrtx[1] - &obj.Vrtx[0])==  1
Flaeche[1] Eckpkt[1] = Obj.-Pkt[1]:
    1.00,    -1.00,    -1.00
(obj.Face[1].Pnt[2] - &obj.Vrtx[0])/(&obj.Vrtx[1] - &obj.Vrtx[0])==  0
Flaeche[1] Eckpkt[2] = Obj.-Pkt[0]:
   -1.00,    -1.00,    -1.00
Symbol: ▒


Ende!
```

`LoadSetCGF.exe`

Typische Vorgänge beim Laden eines grafischen Objektes:

- Bereitstellung von (Struktur-)Variablen für bekannte Merkmale
- Belegung der (Struktur-)Variablen mit Werten aus Datei
- Sukzessive Speicherplatzreservierg. für restliche Merkmale – z.B. (in Funktion mit CGF-Objekt **`obj`**):

```
obj->Vrtx=(float**)calloc(obj->nVrtx, sizeof(float *));
```

- Berechnung (i.d.R. auch Speicherung) wiederholt benötigter, abhängiger Merkmale – z.B. (vgl. Übungen)
  - **Bounding Box** (umhüllender Quader, zur schnellen Überprüfg. von Sichtbarkeit im Sichtvolumen, Verdeckung, Kollision etc.)
  - **Flächennormalen** (zur Ermittlung der Lage gegenüber Lichtquellen oder dem Augenpunkt, z.B. zur Eliminierung abgewandter Objektflächen – engl. *back face culling*)

3D-Punkte, 3D-Vektoren

In den Naturwissenschaften häufig: Rechnen mit 3D-Vektoren und -Punkten –z.B.: Stärke, Richtung d. Gravitation g(x,y,z) an einem bestimmten Punkt im Weltraum p(x,y,z)

Darstellung von 3D-Vektoren und -Punkten meist identisch: kartesische Koordinaten als Zahlentripel (x, y, z) – aber:

- Vektoren haben Betrag und Richtung, aber keine Position.
- Punkte haben Position, aber weder Betrag noch Richtung.

Typische Umgehung der Unwegsamkeit: Darstellung eines Punktes über seinen Ortsvektor, d.h. über seinen *Versatz* gegenüber d.Koordinaten-Ursprung (*V*.-Betrag, *V*.-Richtung!)

Verbleibendes Problem: Darstellung bei Verwendung mehrerer Koordinatensysteme.

⇨ Erweiterung des Begriffs des (3D-)Koordinatensystems:

3D-Punkte, 3D-Vektoren

Ein *Coordinate Frame* (*CF*, Koord.rahmen,-netz) besteht aus

- 3 senkrecht zueinander stehenden Einheitsvektoren $\underline{\mathbf{i}}$,$\underline{\mathbf{j}}$,$\underline{\mathbf{k}}$
- einem besonderen Punkt φ, dem Ursprung (engl. *origin*).

($\underline{\mathbf{i}}$,$\underline{\mathbf{j}}$,$\underline{\mathbf{k}}$,φ können nur über andere CF spezifiziert werden!)

„Homogene" Vektor- u. Punkt-Darstellung (4 Komponenten):

Vektor $\underline{v} = v_1\,\underline{\mathbf{i}} + v_2\,\underline{\mathbf{j}} + v_3\,\underline{\mathbf{k}} \qquad = [\,v_1, v_2, v_3, 0\,]^T$

Punkt $\underline{p} = p_1\,\underline{\mathbf{i}} + p_2\,\underline{\mathbf{j}} + p_3\,\underline{\mathbf{k}} + \varphi \quad = [\,p_1, p_2, p_3, 1\,]^T$

Übereinstimmend mit bisherigen Feststellungen (vgl. Ortsv.):

- Die Differenz zweier Punkte ist ein Vektor.
- Die Summe eines Punktes und eines Vektors ist ein Punkt.
- Die Summe zweier Vektoren ist ein Vektor.
- Skalierung eines Vektors ist sinnvoll.
- Addition von Punkten ist nicht sinnvoll / nicht zulässig.
- Jede Linearkombination von Vektoren ergibt einen Vektor.

3D-Punkte, 3D-Vektoren

Zur Erinnerung:

- **Linearkombination** von Vektoren = Summe skalierter Vektoren:

  $\underline{v} = \alpha_1\ \underline{v}_1 + \alpha_2\ \underline{v}_2 + \ldots\ (\alpha_i \in \mathbf{R})$

- **Affine Kombination** von Vektoren = Summe skalierter Vektoren mit Summe der Skalierungsfaktoren =1:

  $\underline{v} = \alpha_1\ \underline{v}_1 + \alpha_2\ \underline{v}_2 + \ldots\ (\alpha_i \in \mathbf{R},\ \Sigma\alpha_i = 1)$

- **Konvexe Kombination** von Vektoren = Summe skalierter Vektoren mit Summe der Skalierungsfaktoren =1 und mit nichtnegativen Skalierungsfaktoren :

  $\underline{v} = \alpha_1\ \underline{v}_1 + \alpha_2\ \underline{v}_2 + \ldots\ (\alpha_i \in \mathbf{R},\ \Sigma\alpha_i = 1,\ \alpha_i \geq 0)$

Affine u. konvexe Kombinationen von Punkten sind zulässig!

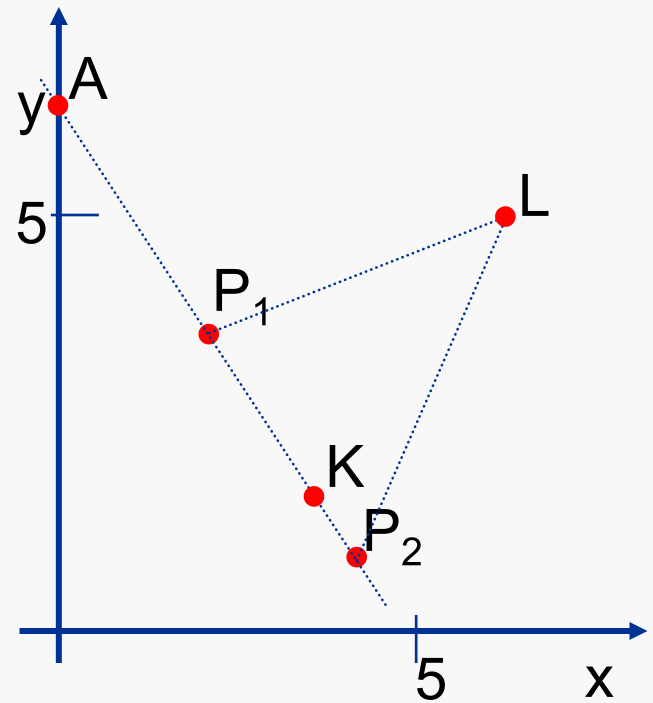
3D-Punkte, 3D-Vektoren

Beispiel: Konvexe (K), affine (A) und lineare (L) Punkt-Kombinationen von $P_1=[2, 4]^T$ und $P_2=[4, 1]^T$:

$$K = 0.25\ P_1 + 0.75\ P_2 = [0.5, 1]^T + [3, 0.75]^T = [3.5, 1.75]^T$$

$$A = 2\ P_1 - 1\ P_2 = [4, 8]^T - [4, 1]^T = [0, 7]^T$$

$$L = 1\ P_1 + 1\ P_2 = [2, 4]^T + [4, 1]^T = [6, 5]^T$$



3D-Punkte, 3D-Vektoren

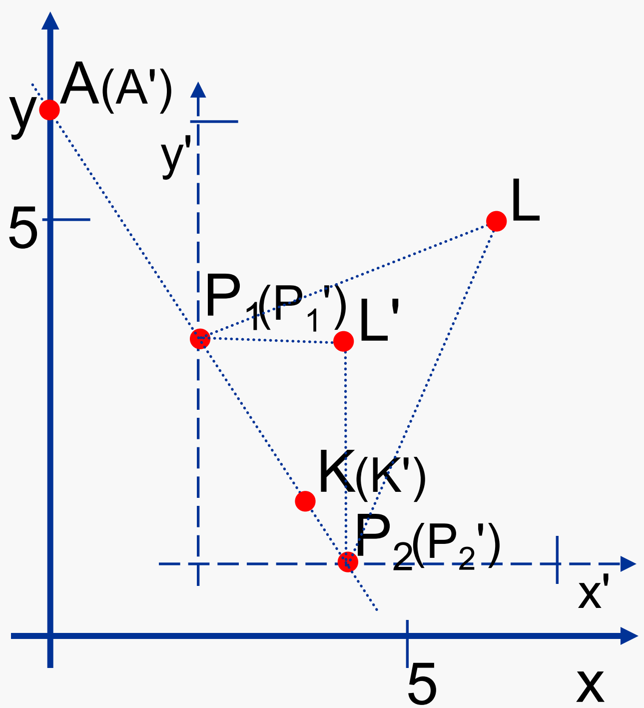
TECHNISCHE HOCHSCHULE MITTELHESSEN
Prof. Dr. Aris Christidis • SS 2017

Beispiel: Konvexe (K), affine (A) und lineare (L) Punkt-Kombinationen von $P_1=[2, 4]^T$ und $P_2=[4, 1]^T$ bei Wechsel des Koordinatensystems zu $P_1'=[0, 3]^T$ und $P_2'=[2, 0]^T$:

$$K' = 0.25\, P_1' + 0.75\, P_2' = [0, 0.75]^T + [1.5, 0]^T = [1.5, 0.75]^T$$

$$A' = 2\, P_1' - 1\, P_2' = [0, 6]^T - [2, 0]^T = [-2, 6]^T$$

$$L' = 1\, P_1' + 1\, P_2' = [0, 3]^T + [2, 0]^T = [2, 3]^T$$



⇨ Nur affine (u. somit auch konvexe) Punkt-Kombinationen sind unabhängig von der Wahl des Koordinatensystems!