Geometrische Constraints in der Softwaretechnik

1 Constraints im 2D CAD Systemen (RelCAD)

1.1 Konstruktive Modellierung

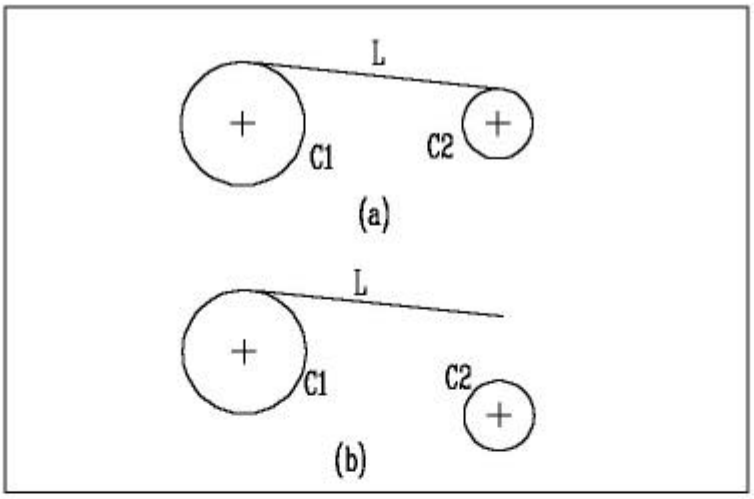


Abb. 1

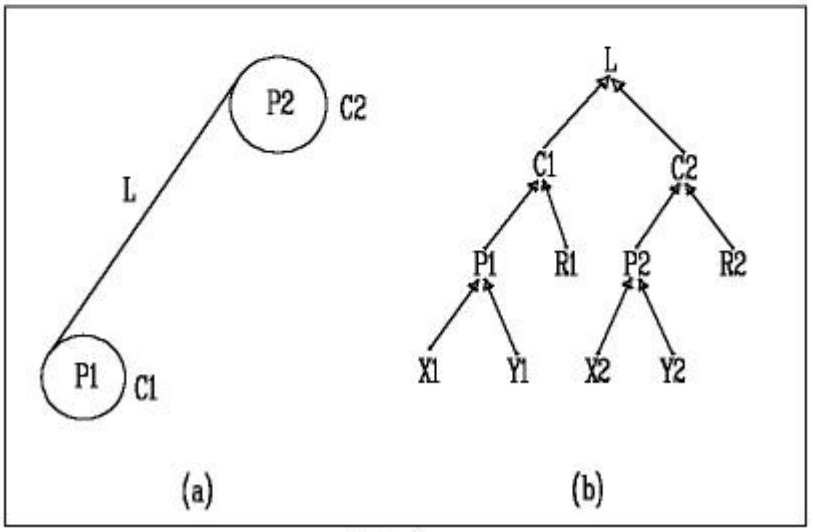


Abb. 2

Constraintbasierte Geometrische Modelle:

- **algebraische, d.h. die Beziehungen zwischen den Elementen sind durch algebraische Gleichungen festgelegt [Light,Gossard]**
- **die geometrischen Beziehungen sind definiert als Fakten und Regeln [Veroust,Schonek,Roller],**
- **eventuell auch durch Prolog Klauseln [Arbab,Wang].**

Im RelCAD System wird ein konstruktiver Ansatz benutzt. Dadurch kann die Anzahl der Gleichungen, die gemeinsam gelöst werden müssen minimiert werden.

Jedes Element des Geometriemodells ist:

- **absolut**
- **ist definiert durch seine Beziehungen zu anderen Elementen.**

Absolute geometrische und nichtgeometrische Objektklassen:

- **Werte z.B. Koordinaten, Längen, Winkel oder auch Materialwerte.**
- **Punkte**
- **Linien**
- **Kreise**
- **Bogen**

Abgeleitete Klassen:

- **Werte:**

Koordinaten von Punkten, Distanzen, Winkel, Formeln

- **Punkte:**

Schnittpunkte, Tangentialpunkte, Mittelpunkte, Endpunkte, Relativpunkte

- **Linien:**

Tangentiallinien, Parallele, Lot, Achsenparallele Linien

- **Kreise:**

Tangential zu 2 Linien oder Kreisen und Radius, Tangential zu 3 Elementen

- **Bogen:**

wie bei Kreisen

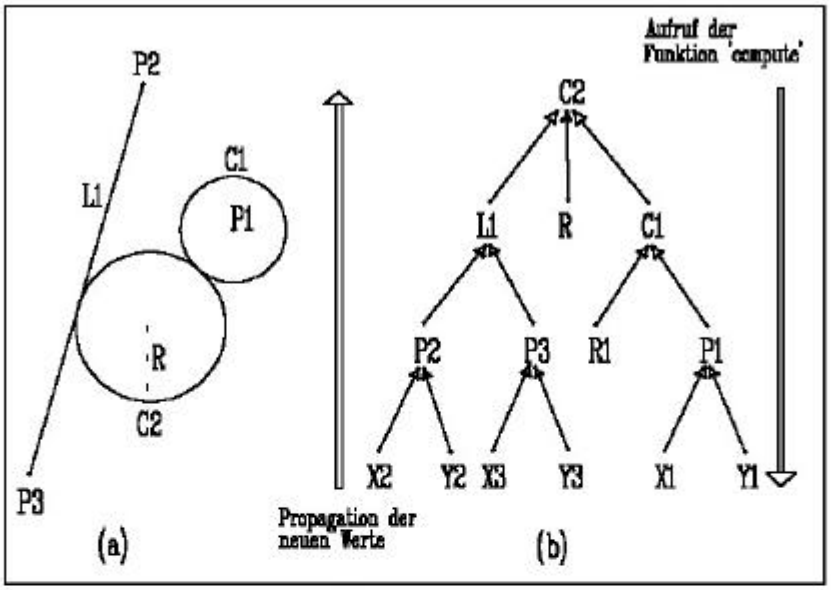


Abb. 3

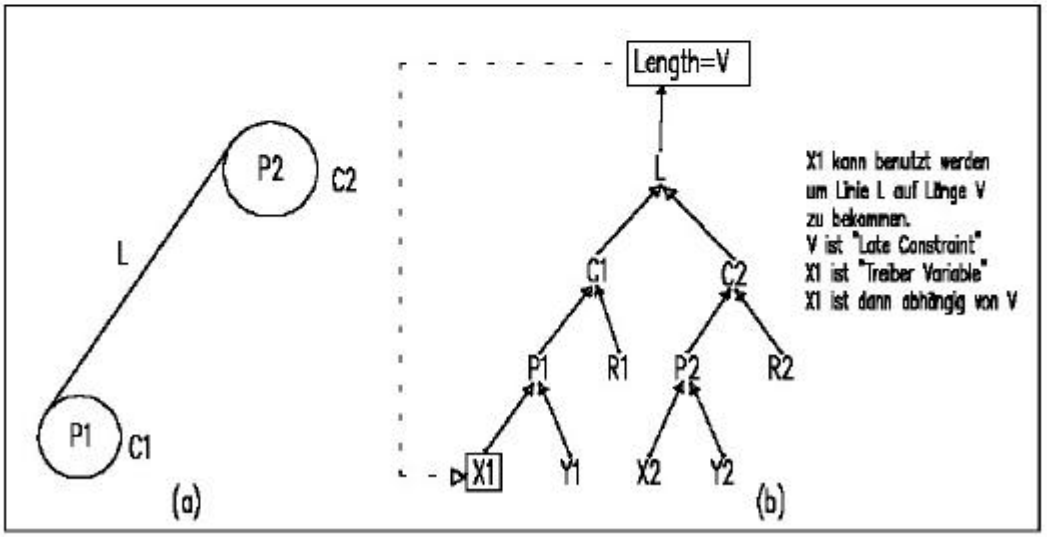


Abb. 4

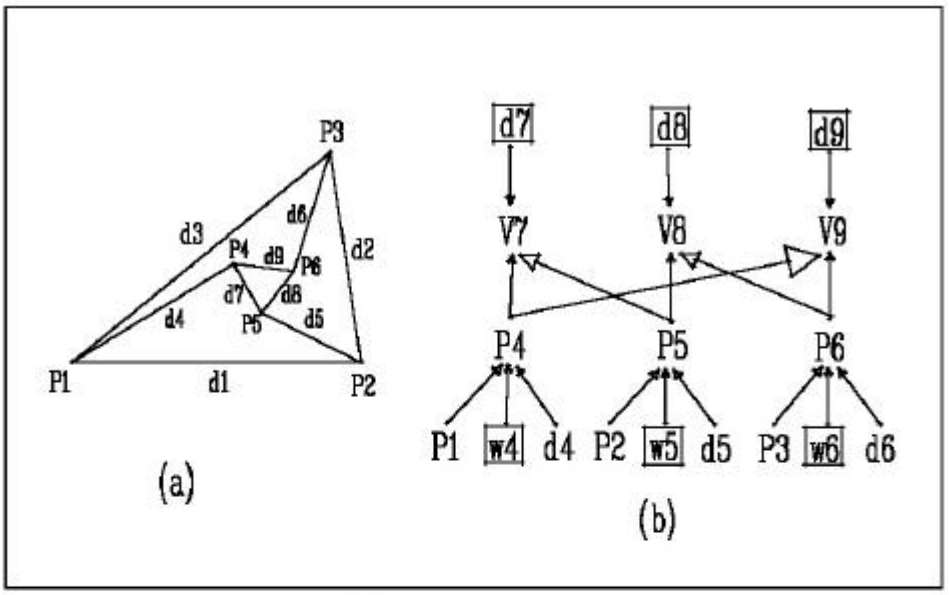

P3
d3
d6
d2
P4
d9
P6
d7
d8
d4
P5
d5
P1
d1
P2
(a)
d7
d8
d9
V7
V8
V9
P4
P5
P6
P1
w4
d4
P2
w5
d5
P3
w6
d6
(b)

Abb. 5

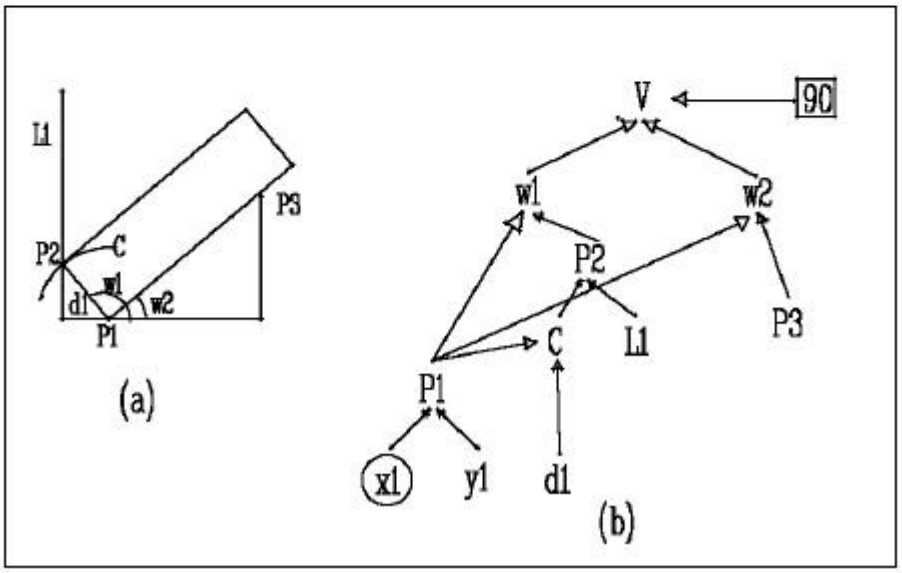

L1
P3
P2
C
w1
d1
w2
P1
(a)
V
90
w1
w2
P2
P3
C
L1
P1
x1
y1
d1
(b)

Abb. 6

1.2 Deklarative Modellierung

Modellierung der Constraints durch einen zunächst ungerichteten Graphen (Berling)

Die Relationen werden durch eine Menge von Gleichungen beschrieben, die nach jeder Variablen aufgelöst werden können.

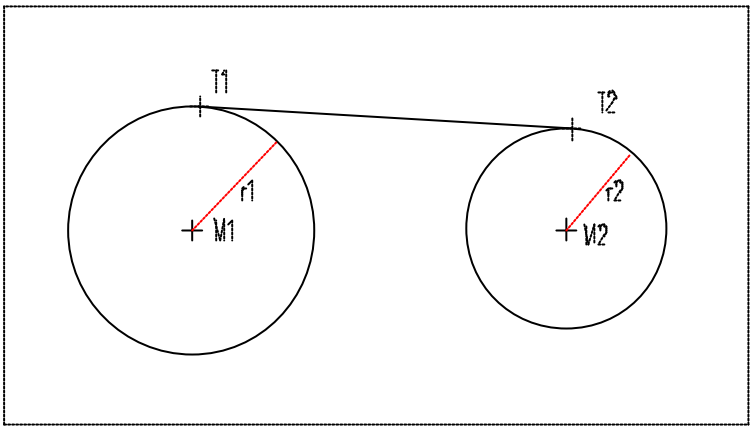
Geometrische Basiselemente:

- **Segment(P1, P2): Line von Punkt P1 nach P2**
- **Circle(M, r): Kreis Mittelpunkt M und Radius r**
- **Arc(P1, P2, M, r): Bogen von Punkt P1 nach P2 mit Mittelpunkt M und Radius r**

Constraint Relationen:

- **dp(P1, P2, d): Abstand zwischen Punkt P1 und P2 ist d**
- **dl(P, Pa, Pe, d): Abstand zwischen Punkt P und Linie von Pa nach Pe ist d**
- **a3(P1, P2, P3, a): Der Winkel zwischen der Linie P1 nach P2 und der Linie P1 nach P3 ist a**
- **a4(P1, P2, P3, P4, a): Der Winkel zwischen der Linie P1 nach P2 und der Linie P1 nach P4 ist a**
- **equ(<expression>): Der Wert von <expression> ist 0. Wenn der Ausdruck Variablen enthält muss jede Variable aus den übrigen berechenbar sein.**
- **fix(<value>): Ein Constraint ohne Eingangswert mit genau einem Ausgangswert.**

Beispiel:



Modellbeschreibung:

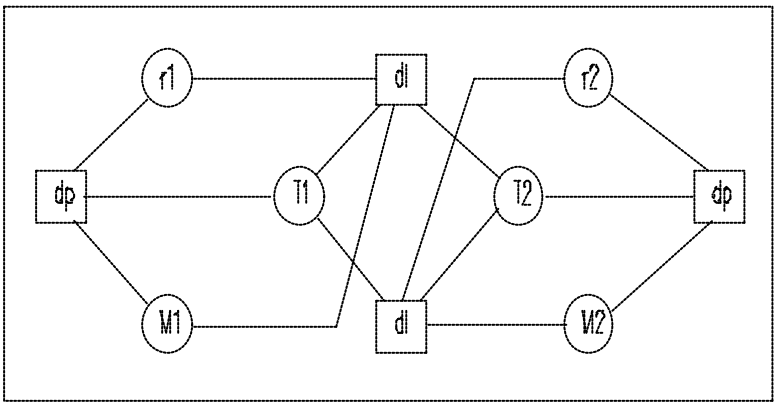
dp(M1,T1,r1),

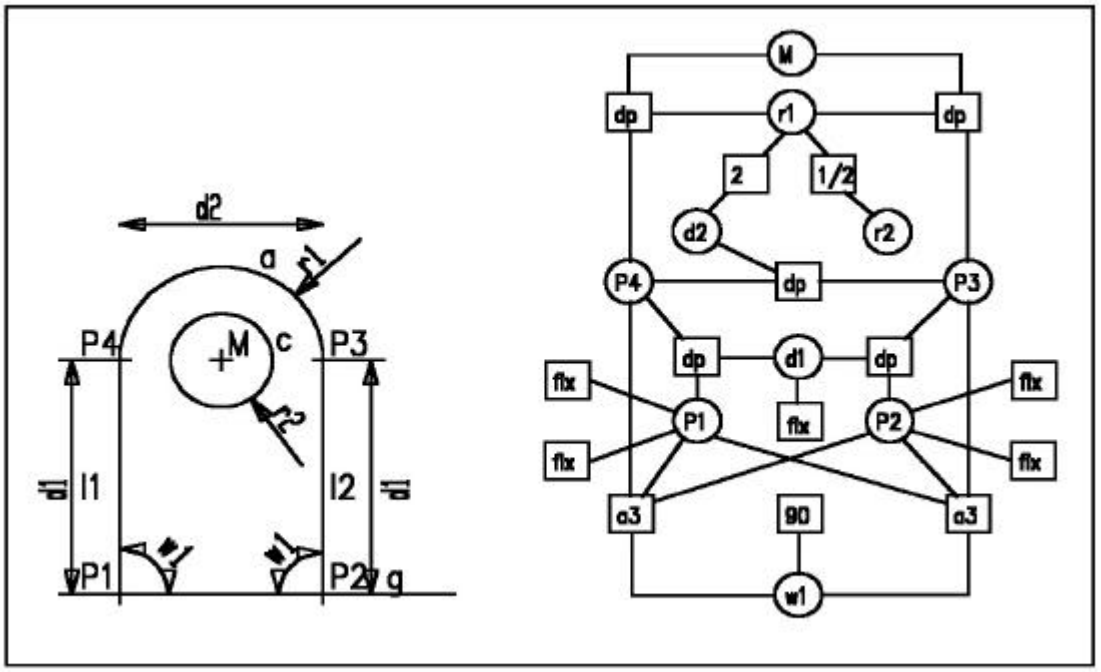
dp(M2,T2,r2),

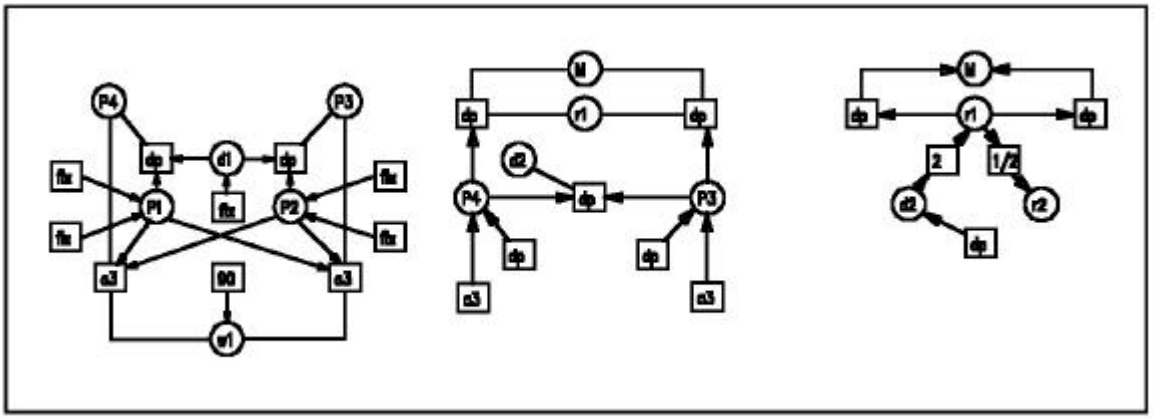
dl(M1,T1,T2,r1),

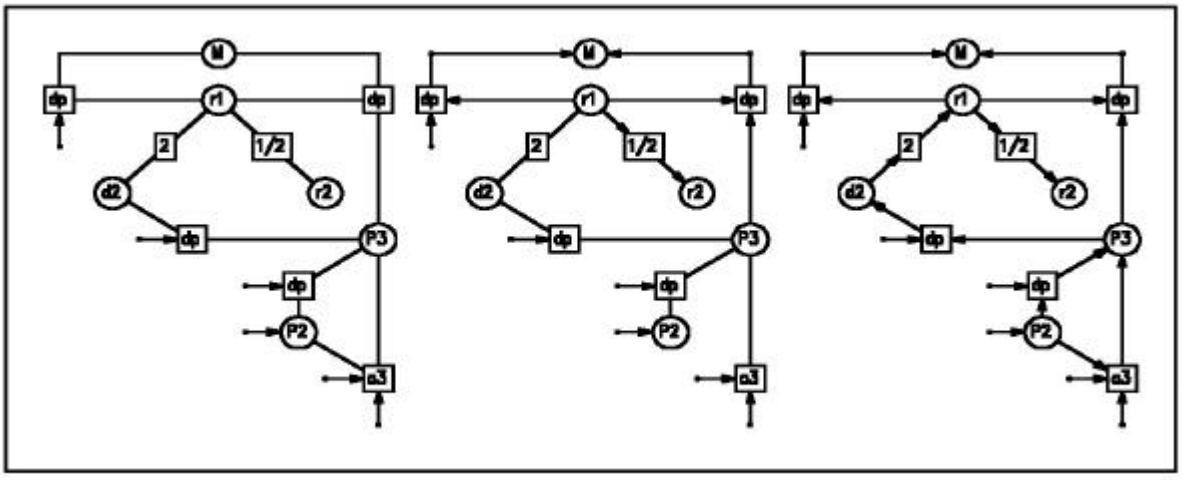
dl(M2,T2,T1,r2),

circle(M1,r1), circle(m2,r2), line(T1,T2)




d2
a
r1
P4
M
c
P3
r2
d1
l1
l2
d1
P1
P2
g
M
dp
r1
dp
2
1/2
d2
r2
P4
dp
P3
dp
d1
dp
flx
flx
P1
flx
P2
flx
flx
a3
90
a3
w1


P4
P3
dp
d1
dp
flx
flx
flx
flx
P1
P2
a3
90
a3
w1
M
dp
r1
dp
d2
P4
dp
P3
dp
dp
a3
a3
M
dp
r1
dp
2
1/2
d2
r2
dp

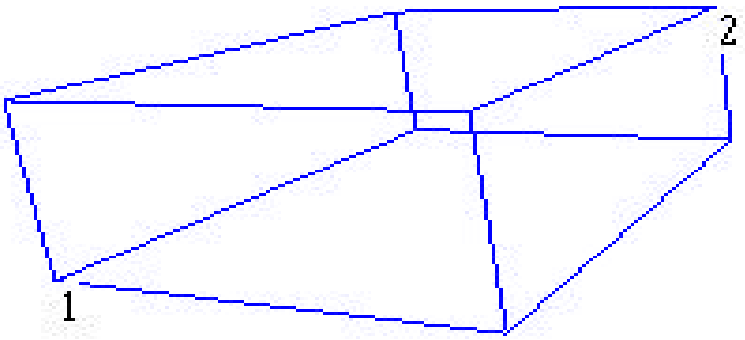

M
dp
r1
dp
2
1/2
d2
r2
dp
P3
dp
P2
a3
M
dp
r1
dp
2
1/2
d2
r2
dp
P3
dp
P2
a3
M
dp
r1
dp
2
1/2
d2
r2
dp
P3
dp
P2
a3

Constraints im 3D CAD Systemen (RelCAD3D)

Parametrische 3D Systems meist history based

Auch hier das Problem: Ändern von Punkten oder Werten die nicht als Parameter gegeben sin sondern von Parametern abhängen

Beispiel:

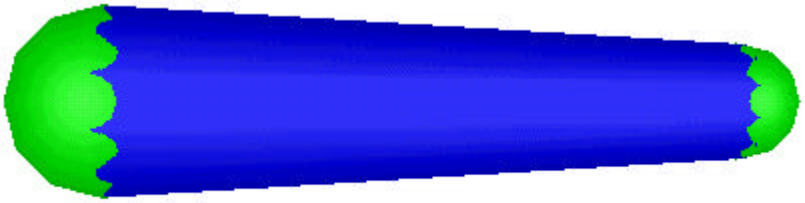


Quader definiert durch 2 Punkte
Wenn einer der anderen Punkte geändert wird, müssen die x, y, z Koordinaten von Punkt 1 oder Punkt 2 auch geändert werden.

Das sollte auch möglich sein, wenn der Quader nicht orthogonal ausgerichtet ist, z.B. nach einer Drehung

Einige Elemente analog zum 2D
The same is aimed in 3D

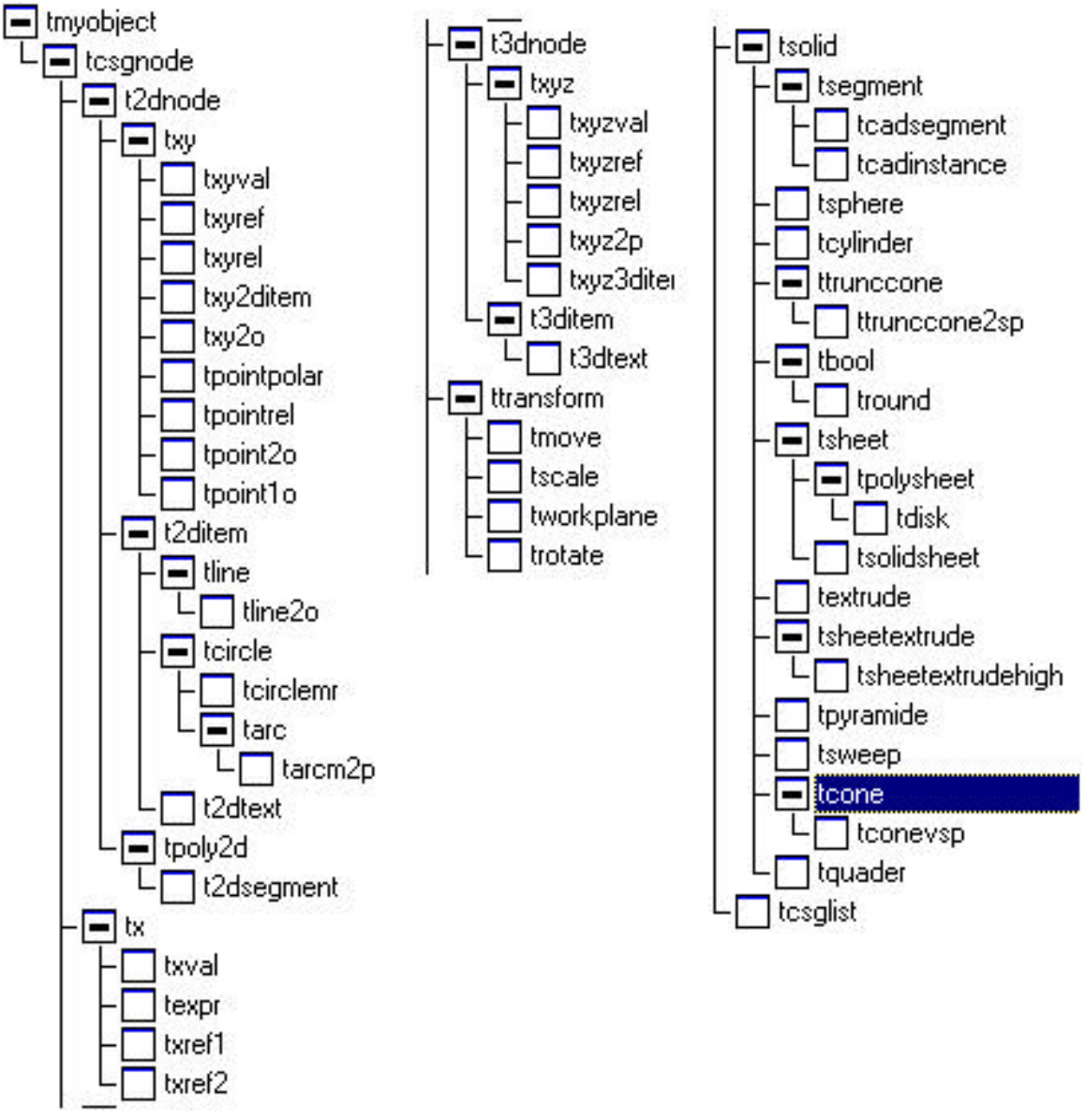
Beispiel: Kegelstumpf tangential an 2 Kugeln



Eine Menge von 3D Solids sind Sweeps einer 2D Fläche
Hier kann das 2D Konzept auch benutzt werden
Im 3D hat man wesentlich mehr Basiselemente und eine komplexere Modellstruktur:

- **Values**
- **2D Points**
- **2D Items**
- **3D Points**
- **3D Items mit Boundary Representation und CSG- Modell**

Die variational geometry basiert hauptsächlich auf dem CSG Modell.
Die Klassen des Modell sind:



Basisklassen: Tcsgnode

- **Werte (1 Dim.): Tx**
- **2- Dim. Elemente: T2dnode**
- **2-Dim. Elemente: T3dnode**

For 2 and 3 D Punkte und Items

2D:

- **Punkt: Txy**
- **Items: Tline, Tcircle, Tarc, Tpoly2d**

3D:

- **Punkt: Txyz**
- **Items: T3ditem**

T3ditem enthält den solid und optional eine Transformation.

```
type t3ditem=class(t3dnode)
 attr:tattr;
 solid:tsolid;
 t:ttransform;
```

tsolid enthält das Brep und das CSG Modell
t enthält die Transformation.

Nur die nicht transformierten Daten sind im Brep Modell enthalten.

Bei Zeichnen mit OpenGL wird

```
glpushmatrix;
glmultmatrixd(@t.m);
```

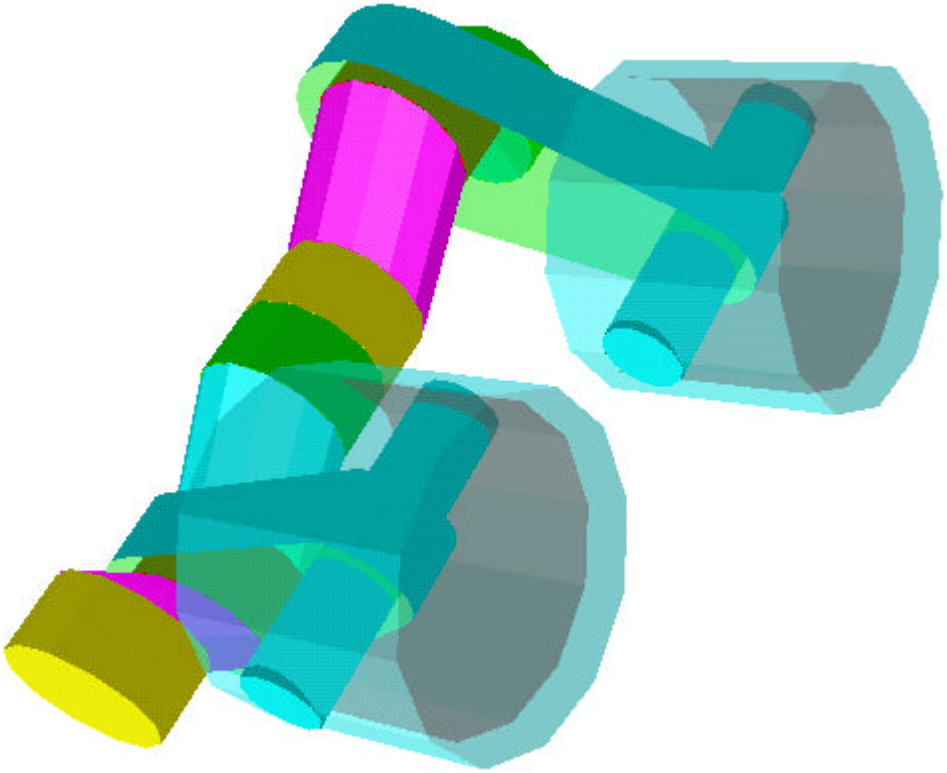
benutzt.

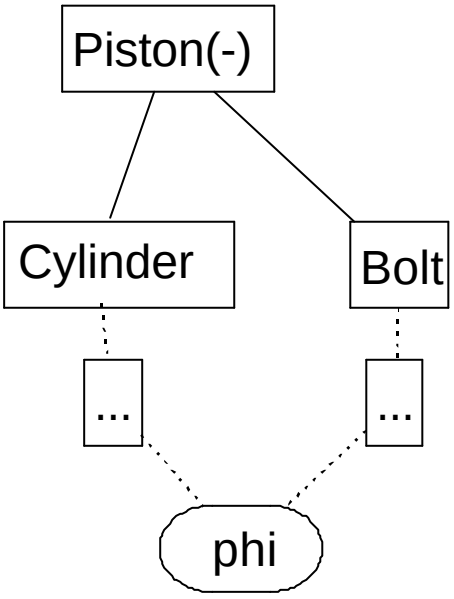
Wenn ein transformiertes item benötigt wird (z.B..für eine boolsche Operation) werden die Punkte der Brep- Darstellung transformiert

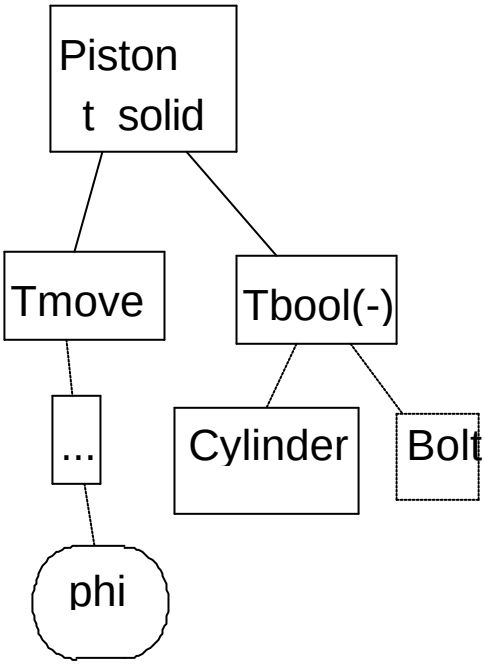
Tsolid hat die Subklassen:
Tsphere, Tcylinder, Tcone, Ttrunccone, Tbool, Tsheet, Textrude, Tpyramide, Tquader, Tsegment

Hierdurch werden keine Boolschen Operationen benötigt, wenn das Modell bewegt wird.

Beispiel:

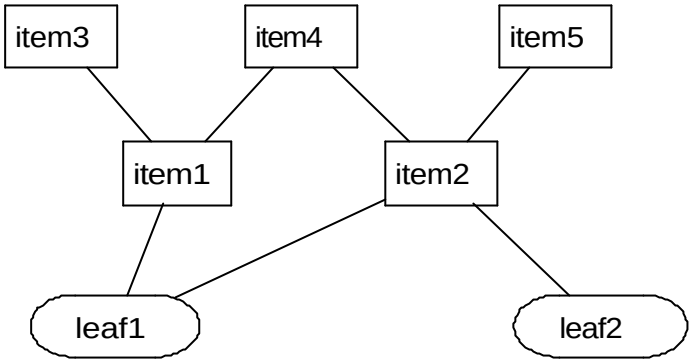



Piston(-)
Cylinder
Bolt
...
...
phi


Piston
t solid
Tmove
Tbool(-)
...
Cylinder
Bolt
phi

Bei Änderung eines Modells:

Recompute: Neuberechnung von den Blättern her.
Transform, Movepoint: Propagieren eines Modells zu den Blättern



Recompute nach Änderung von leaf2

Welche Knoten sind abhängig von leaf2?: item2, item4, item5
In welcher Reihenfolge muss recompute aufgerufen werden?:

Um ein Modell zu traversieren wird die Klasse `traverse` abgeleitet von `twriter`, benutzt.

`Twriter` hat 2 Methoden:
`twriter.write` Schreiben der Daten
`twriter.put` wird für jeden Zeiger auf ein Object in einer Instanz aufgerufen.
Put ruft die store Methode der Klasse auf.

In `ttravese` werden diese Methoden überschrieben:

```
ttraverse=class(tmywriter)
 post:boolean;f:tdoprocedure;
 constructor crea-
te(apost:boolean;af:tdoprocedure);
 procedure put(p:tmyobject);override;
 procedure write(const x;n:integer);override;
```

`ttravese.write`: macht nichts

```
procedure ttraverse.put(p:tmyobject);
begin
if p<>nil then begin
 i:=pointerlist.indexof(p);
 if i<0 then begin
  pointerlist.add(p);
  if post then begin
   p.store(self);
   f(p);
  end else begin
   f(p);
   p.store(self);
  end;
end;
end;
end;
```

`ttraverse.put`: ruft für jeden erreichbaren Knoten genau einmal eine vorgegebene Prozedur auf:
Das kann z.B. sein: Hinzufügen des Knoten zu einer Liste. Dann enthält am Ende die Liste alle erreichbaren Knoten.

Tgetref ist abgeleitet von ttraverse.

```
type tgetrefs=class(ttraverse)
 stack:tintarray;
 refs:tmylist;
 vars:tmylist;
 csgonly:boolean;

constructor create(avars:tmylist;var a-
refs:tmylist; acsgonly:boolean);
begin
pointerlist:=tmylist.create;
stack:=tintarray.create;
vars:=avars;
if arefs=nil then arefs:=tmylist.create;
refs:=arefs;
csgonly:=acsgonly;
end;

procedure put(p:tmyobject);
if p<>nil then begin
 donode:=(not csgonly)or(p is tcsgnode);
 if donode then stack.add(0);
 i:=pointerlist.indexof(p);
 if i<0 then begin
  pointerlist.add(p);
  p.store(self);
 end;
 if donode then begin
  if stack[stack.count-1]>0 then refs.add(p);
  dec(stack.count);
  if(vars.indexof(p)>=0)or(refs.indexof(p)>=0)
  then begin
   for j:=0 to stack.count-1 do stack[j]:=1;
  end;
 end;
end;
```

In tgetref.put enthält die Variable stack den Weg von der Wurzel zum aktuellen Knoten.
Wenn der Knoten inder vars Liste oder schon in der refs Liste ist, werden alle Knoten aus dem Weg markiert.

If ein Knoten markiert wird, wird er zur `refs` Liste hinzugefügt.

```
trgetref:=tgetref.create(vars,refs,true);
trref.put(list);
```

Danach enthält refs alle Knoten auf einem Weg von einem Knoten aus list zu einem Knoten aus vars.

```
procedure
listrecompute(list:tcsglist;vars:tmylist);
var trref:tgetrefs;
 refs:tmylist;
 n:tcsgnode;
begin
refs:=nil;
trref:=tgetrefs.create(vars,refs,true);
trref.put(list);
trref.free;
for i:=0 to refs.count-1 do begin
 n:=tcsgnode(refs[i]);
 n.compute;
end;
```

Aufruf:listrecompute((item3,item4,item5),(leaf2))

Nach trref.put(item3,item4,item5):

refs enthält (leaf2,item2,item4,item5)

Nun wird für jedes Element aus refs die Prozedur compute aufgerufen, und so das geänderte BRep-Model erzeugt. Die Elemente in refs sind von unten nach oben sortiert. Dadurch werden die compute automatisch in der richtigen Reigenfolge aufgerufen.

Wenn ein Item transformiert wird (Translation, Rotation, Scalierung, Ausrichten an einer Arbeitsebene) oder ein Punkt bewegt wird müssen die Änderungen zu den Blättern propagiert werden.

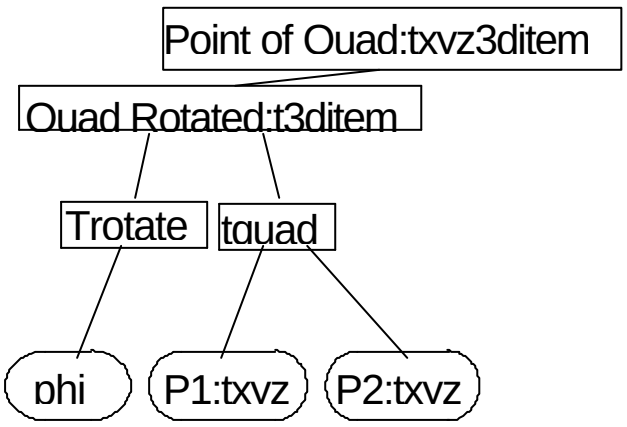
In RelCAD2D wurde Newton Iteration benutzt.
Im 3D benötigt man schon eine Gleichung mit 3 Variablen, wenn ein Punkt bewegt wird.
Deswegen werden hier Standard Methoden zu Propagation der Punktbewegung bzw. zur Transformation jeder 3D item Klasse definiert.

```
Tsolid.movepoint(anr:integer; dx:tvertex);
Tsolid.transform(am:tmatrix);
```

- **Bewegung eines Punktes**

Beispiel:
Ein Punkt in einem gedrehten Quader, der nicht notwendig einer der definierenden Punkte ist, soll verschoben werden:

```
type txyz3ditem=class(txyz)
{nr>=0 node in the brep of the solid,
 nr<0 defining point of the solid}
 it:t3ditem;
 nr:integer;
 l:tintarray;
 {it points to the item on the highest level
 l is a if indices to the lower levels if it is a seg-
ment or a boolean solid}
```

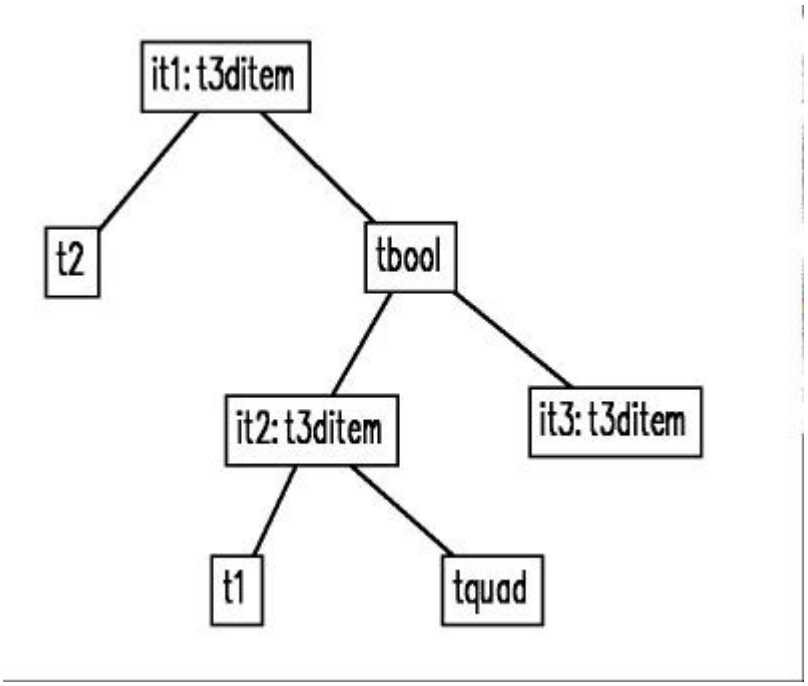


```
procedure txyz3ditem.move(const dx:tvertex);
it2:=it;m:=idmatrix;li:=0;
while it2<>nil do begin
 if it2.t<>nil then begin
  m2:=it2.t.m;
  for i:=0 to 2 do m2[3,i]:=0;
  if invertmat(m2,m1) then m:=matxmat(m1,m);
 end;
```

Weil der Punkt des gedrehten Quaders bewegt wird, muss die inverse Drehung auf den Verschiebevektor angewandt werden, um die korrekte Transformation des nicht gedrehten Quaders zu erreichen.

```
 so:=it2.solid;
 if so<>nil then begin
  if so is tsegment then begin
   se:=so as tsegment;
   it2:=t3ditem(se.list[l[li]]);inc(li);
  end else if so is tbool then begin
   bo:=so as tbool;
   if l[li]=0 then it2:=t3ditem(bo.op1) else
   it2:=t3ditem(bo.op2);inc(li);
  end else break;
 end else break;
end;
d:=vecxmat(dx,m);
so:=it2.solid;
so.movepoint(nr,dx);
compute;
```

Wenn der Punkt eines item durch eine boolsche Operation erzeugt wurde, werden die Parameter der boolschen Operation transformiert und `tbool` wird neuberechnet.



Ein Punkt in it1 soll auf eine vorgegebene Position bewegt werden.

Wie muss der Punkt im tquad bewegt werden?

Gegeben:

x sei die Orginal Position in tquad

M sei die Transformation zur neuen Position in it1

Gesucht:

Transformation des Punktes in tquad

Es gilt::

x M' T1 T2 = x T1 T2 M

Daraus ergibt sich:

M' = (T1 T2) M ($T2^{-1}$ $T1^{-1}$)

Der Translationsvektor in tquad ist dann:

D' = x M' - x

So kann die Bewegung eines gegebenen Punktes propagiert werden zu einer Bewegung eines Punktes in dem orthogonal ausgerichteten Quader.

Diese Bewegung muss dann so definiert werden, dass der Quader erhalten bleibt.

```
procedure tquader.movepoint(anr:integer;const
dx:tvertex);
if anr=-1 then v1.move(dx)
else if anr=-2 then v2.move(dx)
else begin
 x0:=vertex(idmatrix,anr);
 d1:=nullvertex;d2:=nullvertex;
 for i:=0 to 2 do begin
  if abs(x0[i]-v1[i])<eps then begin
   d1[i]:=dx[i];
  end;
  if abs(x0[i]-v2[i])<eps then begin
   d2[i]:=dx[i];
  end;
 end;
 if not vertexeq(d1,nullvertex)then v1.move(d1);
 if not vertexeq(d2,nullvertex)then v2.move(d2);
end;
compute;
```

Die definierenden Punkte v1,v2 werden so bewegt, dass der zu bewegende Punkt die gewünschten Koordinaten erhält, die Flächen aber planar bleiben.

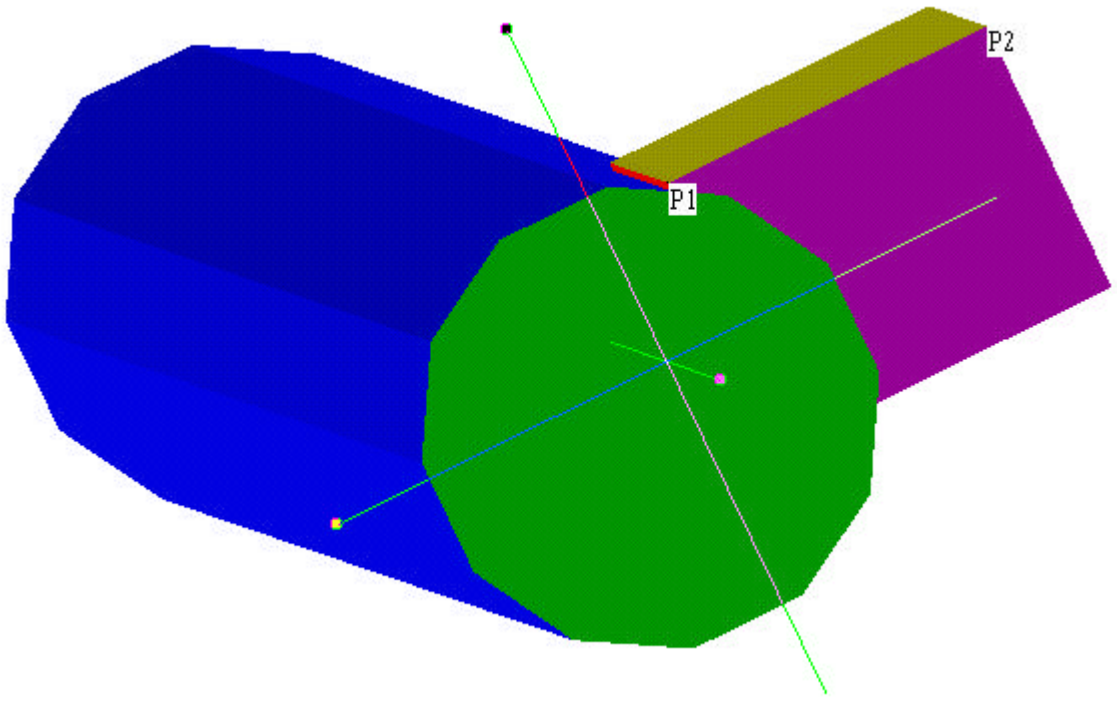
Ähnliche Methoden werden für die anderen Subklassen von tsolid definiert. Z.B. die Bewegung eines Punktes auf der Oberfläche einer Kugel führt zu einem neuen Radius.

Transformation (Translaton, Rotation) eines 3D-Items.

Wenn ein 3D Item transformiert wird, ändert sich seine Gestalt nicht.

Es gibt zwei Möglichkeiten für die Propagation:

- **Der Solid bleibt unverändert, nur die Transformation ändert sich.**
- **Die Transformation wird propagiert zu den definierenden Punkten des Items.**
  Z.B..
  dem Mittelpunkt einer Kugel,
  den Achsenpunkten eines Zylinders oder Kegels,
  den definierenden Punkten eines Quaders.



3 Geometrische Constraints bei Diagrammen

Hier sollen Diagramme aus Knotenartigen Objekten (Box, Kreis, Frame usw.) und Kantenartigen Objekten (Linien, Manhattanlinien, Orthogonale Verbindungen) betrachtet werden.

```
type tany=class(tmyobject)
 p:pdb;
 computed:boolean;
 fixed:byte;

titem=class(tany)
 super:tfram;

tnod=class(titem)
 constr,edges:tmylist;
 public x,y:real;
 typ:string;
 text:tmystringlist;
```

egdes: die adjazenten Kanten
constr: die Constraints
typ: die Art des Knotens (Box, Kreis, Oval)
text: der zugeordnete Text.
super: der Frame in dem der Knoten liegt bzw. nil

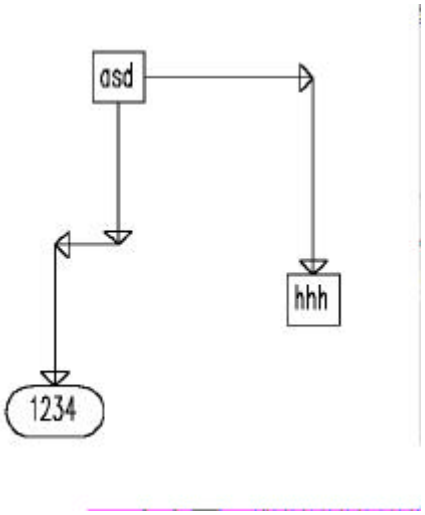
```
tedg=class(titem)
 public
 n1,n2:tnod;
 stuetz:tmylist;
 typ:string;
 richtung1,richtung2:trichtung;
 pos1,pos2:trichtung;
 index1,index2:integer;
```

n1: der Startknoten
n2: der Endknoten
stuetz: Stützpunkte
typ: der Form der Kante (Linie oder definiertes Shape)

richtung1,richtung2: gewünschte Lage am Knoten
rcenter: immer zum Zentrum
rrand: immer an den Rand je nach Lage des anderen Knotens bzw. Stützpunktes
rvertikal immer oben oder unten, ja nach Lager des anderen Knotens
rhorizontal immer links oder rechts ja nach Lage des anderen Knotens
roben, runten, rlinks, rrechts immer an dem Rand unabhängig von der Lage des anderen Knotens
pos1,pos2: tatsächliche aktuelle Lage
index1,index2: Sortierung auf der Seite, bei pos<>rcenter

Von tedg abgeleitet ist:

```
type tmanhattan=class(tedg2)
private waag:boolean;
```



und

```
type tedgortho=class(tedg2)
```

Sollen 2 Knoten horizontal ausgerichtet sein, wäre dies über eine gemeinsame Y Koordinate möglich.Bei der Definition des 2. Knotens muss jedoch wie im RelCAD vom Benutzer auf die Y Koordinate des 1. Knotens Bezug genommen werden.
Hier daher ein anderes Konzept:

```
type tconstr=class(tany)
 gleich:boolean;
 nodes:tmylist;

type twaagrecht=class(tconstr)
 y,xfirst,xlast:double;
```

nodes: Knoten, die zu diesem Constraint gehören
gleich: die Knoten sollen einen gleichmäßigen Abstand haben
y: die Y Koordinate aller Knoten
xfirst: die erste X Koordinate, nur bei gleich
xlast: die letzte X Koordinate, nur bei gleich

Analog:

```
type tsenkrecht=class(tconstr)
 x,yfirst,ylast:double;
```

und

```
type tausrichten=class(tconstr)
{Knoten werden auf gleich Größe ausgerichtet}
 widtheq,heighteq:boolean;
 width,height:double;
```


```
[ 0tnod] 2 nil 81.76 146.2 Box
 [ 3:tmystringlist] 1 123
 [ 4:tmylist] 1 [2]
 nil
[ 1tnod] 9 nil 158.5 146.2 Box
 [ 5:tmystringlist] 1 asd
 [ 6:tmylist] 1 [2]
 nil
[ 2twaagrecht] -1
 [ 7:tmylist] 2 [0] [1]
```

146.2 81.76 158.5 0

Wird nun ein Knoten verschoben oder verändert propagiert er seine Änderung an seine Constraints und diese wiederum propagieren die Änderung an alle zugeordneten Knoten.

```
procedure compute(afrom:tany;amode:byte);
```

Mode kann folgende Werte annehmen:

```
const isx=1;isy=2;isxy=3;isin=4;iswidth=8;isheight=16;
```

Bei jedem compute wird zunächst geprüft ob die entsprechende Eigenschaft noch nicht propagiert ist:

```
procedure tnod.compute(afrom:tany;amode:byte);
begin
mode:=(not fixed) and amode;
if mode<>0 then begin
 fixed:=fixed or mode;


[-1]->[1]3 comp node
[1]->[2]3 comp waagr
[2]->[0]2 comp node
```

Ein Frame ist ein Knoten, der einen Subgraphen enthalten kann. Dehnt sich der Subgraph aus, sollte sich auch der Frame ausdehnen

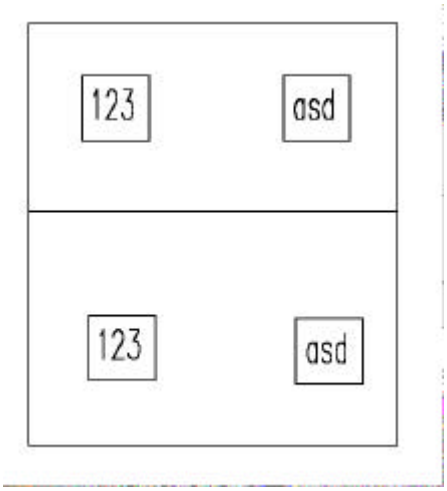
```
tfram=class(tnod)
 x1,y1,x2,y2,randx1,randy1,randx2,randy2:real;
 xdiff,ydiff:real;
 sub:tmylist;
```

x1,y1,x2,y2: die Ausdehnung
Randx1,randy1,randx2,randy2: die Ränder
sub: die enthaltenen Knoten

Hat ein Knoten super<>nil propagiert er seine Änderung auch an den übergeordneten Frame (isin). Dieser prüft, ob die neue Position außerhalb des Frames liegt und dehnt sich evtl. aus. Umgekehrt wird eine Änderung der Position des Frames auf alle enthaltenen Knoten propagiert.

Bisweilen hat man auch mehrere Frames die Horizontal oder vertikal nebeneinander liegen sollen.

```
tgroup=class(tfram)
 zz:tmylist;//Trennlinien vertikal y horizontal x
 horizontal:boolean;
```



Hier werden bei einer Änderung der Ausdehnung eines Teilframes auch alle vorangehenden bzw. nachfolgenden Teilframes und die in ihnen enthaltenen Knoten verschoben.

Propagierung der Kanten

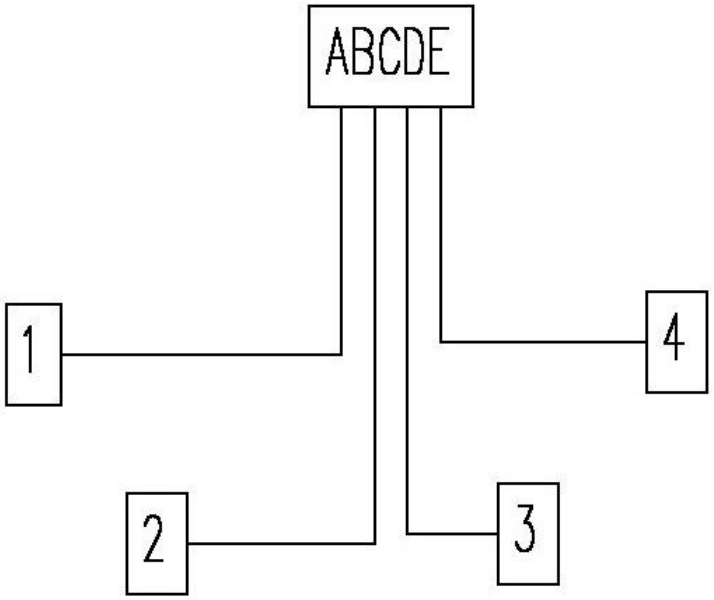
Neuberechnen der Kanten erst sinnvoll, wenn alle adjazenten Knoten neu berechnet sind.
Bei Änderng eines Knotens erhalten alle adjazenten Kanten
`computed:=true;`

Die Ausrichtung einer Kante an einem Knoten wird bestimmt durch:

```
type trichtung=(rcenter,rrand,rvertikal,rhorizontal,
                roben,runten,rrechts,rlinks,
                recke,rul,rur,ror,rol);
```

- **rcenter immer zum Zentrum**
- **rrand immer an den Rand je nach Lage des anderen Knotens bzw. Stützpunktes**
- **rvertikal immer oben oder unten, ja nach Lager des anderen Knotens oder Stützpunktes**
- **rhorizontal immer links oder rechts ja nach Lage des anderen Knotens oder Stützpunktes**
- **recke: immer an eine Ecke je nach Lage des anderen Knotens oder Stützpunktes**

Dies wird bestimmt durch die Komponenten
`richtung1,richtung2`



Bei der Änderung der Knoten werden alle adjazenten Kanten in einer Liste gesammelt:

```
procedure recomputeedges(edgelist:tmylist);
var i,j:integer;
 e:tedg;
 n:tnod;
 nodelist:tmylist;
begin
nodelist:=tmylist.create;
for i:=0 to edgelist.count-1 do begin
 e:=edgelist[i]as tedg;
 e.comppos;
 nodelist.addnew(e.n1);nodelist.addnew(e.n2);
end;
```

Aus Richtung und Lage berechnet comppos die aktuelle Ausrichtung am Knoten

```
rrand-> roben,runten,rrechts,rlinks
recke-> rul,rur,ror,rol

for i:=0 to nodelist.count-1 do begin
 n:=nodelist[i]as tnod;
 n.compindex;
 for j:=0 to n.edges.count-1 do
 edgelist.addnew(n.edges[j]);
end;
```

Bei roben..rlinks wird aus der aktuellen Position die Reihenfolge berechnet.

```
for i:=0 to edgelist.count-1 do begin
 e:=edgelist[i]as tedg;
 e.compstuetz;
 e.updatepdb;
end;
end;
```

Aus der Ausrichtung am Knoten werden für die Manhattan und die orthogonalen Kanten die Stützpunkte neuberechnet und dann die Kanten berechnet

Als Beispiel comppos für Manhattan-Kanten

```
procedure tmanhattan.comppos;
begin
if waag then begin
 if n2.x>n1.x then pos1:=rrechts else pos1:=rlinks;
 if n1.y>n2.y then pos2:=roben else pos2:=runten;
end else begin
 if n1.x>n2.x then pos2:=rrechts else pos2:=rlinks;
 if n2.y>n1.y then pos1:=roben else pos1:=runten;
end;
end;
```

compindex berechnet für die Manhattan- und orthogonalen Kanten mit Position auf dem Rand (roben..rlinks) die Reihenfolge:

```
procedure tnod.compindex;
var ii,i,j,ixy:integer;
 e:tedg; n2:tnod; edgelist:tmylist; ir:trichtung;
 listexy,listee:array[roben..rlinks]of tmylist;
 xy:txy; sortgreater,sortless:tsortlist;
begin
for ir:=roben to rlinks do begin
 listexy[ir]:=tmylist.create;
 listee[ir]:=tmylist.create;
end;
for j:=0 to edges.count-1 do begin
 e:=edges[j]as tedg;
 if e.n1=self then begin
  ir:=e.pos1;ixy:=1;n2:=e.n2;
 end else begin
  ir:=e.pos2;ixy:=0;n2:=e.n1;
 end;
 if (ir>=roben)and(ir<=rlinks)then begin
  listee[ir].add(e);
  if e is tedgortho then begin
   xy:=e.stuetz[ixy]as txy;
   xy:=txy.create(xy.x,xy.y);
   listexy[ir].add(xy);
  end else listexy[ir].add(txy.create(n2.x,n2.y));
 end;
end;
```

wenn mehr als eine Kante auf einer Seite die Kanten sortieren

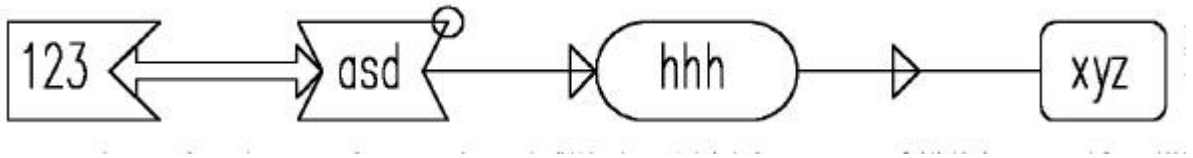
```
for ir:=roben to rlinks do begin
 if listee[ir].count>1 then begin
  sortgreater:=tsortlist.create;
  sortless:=tsortlist.create;
  for j:=0 to listee[ir].count-1 do begin
   xy:=listexy[ir][j]as txy;
   e:=listee[ir][j] as tedg;
   if (ir=roben)or(ir=runten) then begin
```

Bei oben,unten sortieren (1) ><x und (2) nach y

```
    if xy.x>=x then sortgreater.add(e,xy.y)
```

```
    else sortless.add(e,xy.y);
   end else begin
```
Bei links,rechts sortieren (1) ><y und (2) nach x
```
    if xy.y>=y then sortgreater.add(e,xy.x)
    else sortless.add(e,xy.x);
   end;
  end;
```
Die Indices setzen
```
  ii:=(listee[ir].count-1)*100;
  if (ir=rrechts)or(ir=roben)then begin
   for j:=0 to sortless.count-1 do begin
    e:=sortless[j] as tedg;
    if e.n1=self then e.index1:=ii else e.index2:=ii;
    inc(ii);
   end;
   for j:=sortgreater.count-1 downto 0 do begin
    e:=sortgreater[j] as tedg;
    if e.n1=self then e.index1:=ii else e.index2:=ii;
    inc(ii);
   end;
  end else begin
   for j:=sortless.count-1 downto 0 do begin
    e:=sortless[j] as tedg;
    if e.n1=self then e.index1:=ii else e.index2:=ii;
    inc(ii);
   end;
   for j:=0 to sortgreater.count-1 do begin
    e:=sortgreater[j] as tedg;
    if e.n1=self then e.index1:=ii else e.index2:=ii;
    inc(ii);
   end;
  end;
 end else if listee[ir].count=1 then begin
  e:=listee[ir][0] as tedg;
  if e.n1=self then e.index1:=0 else e.index2:=0;
 end;
end; //for ir
end;
```

Definition der Gestalt von Knoten und Kanten



Hierzu werden die X und Y Werte mit -100..+100 skaliert.
Zu einem solchen relativen Wert kann noch ein Distanz angegeben werden, die entweder absolut ist, oder sich auf Breite bzw. Höhe bezieht.

Fahne1:
P -100, -100; 100y100,-100; 100,0; 100y100,100; -100,100

Fahne2:
P -100y-50, -100; 100y50,-100; 100,0; 100y50,100; -100y-50,100;-100,0
C 100y50, 100, 0y30

Oval:
P -100,-100;100,-100,180;100,100;-100,100,180

Bei den Kanten wird der Weg vom Start- zum Zielknoten mit 0..100 skaliert.

Pfeil mit Spitze
L 0,0;100,0
P 100a-3,3;100a-3,-3;100,0

Doppelpfeil:
P 0,0;0a3,-3;0a3,-1;100a-3,-1;100a-3,-3;100,0;100a-3,3;100a-3,1;0a3,1;0a3,3

Pfeil mit Spitze in der Mitte
L 0,0;100,0
P 50a-3,3;50a-3,-3;50,0