2. Ein exemplarischer Ansatz: Realm-basierte räumliche Datentypen [nach Schneider/Güting]

Definition: Ein **Realm** $R = (P, S)$ in einem Gitter $N \times N$ [1]:

$N := \{0, ..., n\}$, n finite and representable		
$P_N := N \times N$	N-points	(grid)
$S_N := P_N \times P_N$	N-segments	
$P \subseteq P_N$	R-points	
$S \subseteq S_N$	R-segments	

Properties

(i) $\forall\, s \in S : s = (p, q) \Rightarrow p \in P \wedge q \in P$

(ii) $\forall\, p \in P\ \forall\, s \in S : \neg\, (p \underline{\text{ in }} s)$

(iii) $\forall\, s, t \in S, s \neq t : \neg\, (s \text{ and } t \underline{\text{ intersect}})$
$\wedge\, \neg\, (s \text{ and } t \underline{\text{ overlap}})$

- Ein Realm entspricht einem planaren Graphen mit Knoten P und Kanten S, die mit räumlichen Lagen behaftet sind.
- Eigenschaften[2]:
  - (i) Endpunkte von Realm-Segmenten gehören zu den Realm-Punkten.
  - (ii) Keine Realm-Punkte liegen mitten auf Realm-Segmenten.
  - (iii) Realm-Segmente schneiden sich nur in Endpunkten.

[1] Die Realm-Bedingungen werden beim Einfügen neuer Realm-Punkt und -Segmente sichergestellt. So müssen Segmente entsprechend neu ermittelter Schnittpunkte u.U. aufgespalten werden! — Die folgende Darstellung hängt jedoch nicht unbedingt von der Vorgabe eines endlichen Gitters ab. Beim Einfügen brauchen Segmente nicht unbedingt im Gitter neu gezeichnet (also mit mehr "Zacken" versehen) werden, sofern man kleine geometrische Ungenauigkeiten in Kauf nimmt, aber den Mehraufwand durch viele Zwischenpunkte einsparen will. Wesentlich ist der konsequente Aufbau komplexer raümlicher Objekte nur aus primitiven Objekten, nämlich Punkten und Segmenten.

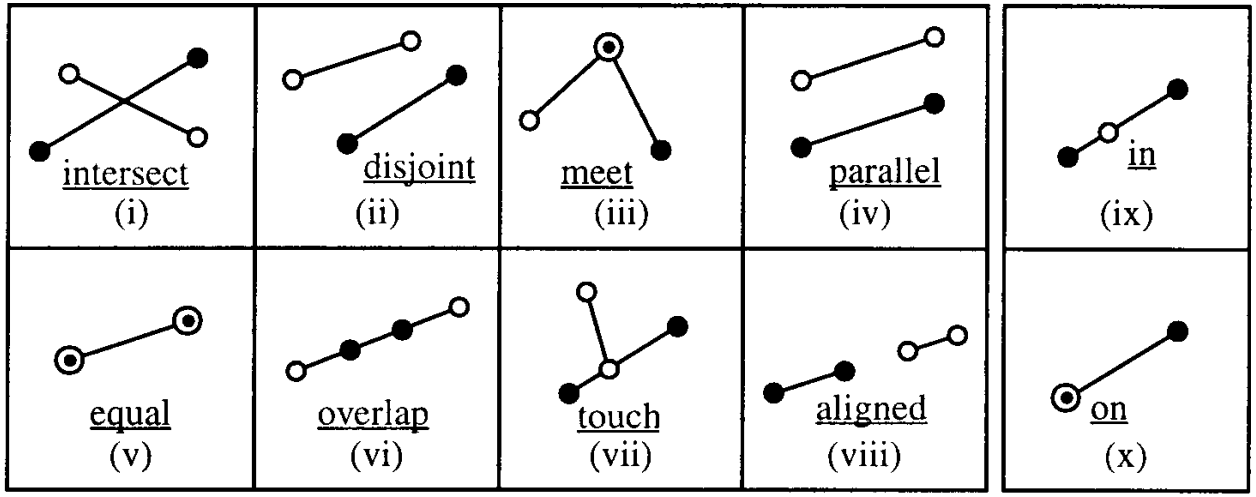
[2] zur Erläuterung der verwendeten Prädikate in, intersect, overlap (in Infix-Notation) s.u.

Realm-basierte räumliche Datentypen (Forts.)

Definitionsstufen:

4	ROSE Algebra Operations	Objects: *points*, *lines*, *regions* Operations: **=**, ≠, **inside**, **edge_inside**, **vertex_inside**, **area_disjoint**, **edge_disjoint**, **disjoint**, **intersects**, **meets**, **adjacent**, **encloses**, **on_border_of**, **border_in_common**, **intersection**, **plus**, **minus**, **common_border**, **vertices**, **contour**, **interior**, **count**, **dist**, **diameter**, **length**, **area**, **perimeter**, **sum**, **closest**, **decompose**, **overlay**, **fusion**
3	Spatial Data Types and Spatial Algebra Primitives	Objects: *points*, *lines*, *regions* Operations: ***union***, ***intersection***, ***difference***, (***area-***)***inside***, ***edge-inside***, ***vertex-inside***, ***area-disjoint***, ***edge-disjoint***, (***vertex-***)***disjoint***, ***meet***, ***adjacent***, ***intersect***, ***encloses***, ***on_border_of***, ***border_in_common***
2	Realms, Realm-Based Structures, and Realm-Based Primitives	Objects: *R*-point, *R*-segment; *R*-cycle, *R*-face, *R*-unit, *R*-block Operations: *on*, *in*, *out*, (*area-*)*inside*, *edge-inside*, *vertex-inside*, *area-disjoint*, *edge-disjoint*, (*vertex-*)*disjoint*, *adjacent*, *meet*, *encloses*, *intersect*, *dist*, *area*
1	Robust Geometric Primitives	Objects: *N*-point, *N*-segment Operations: =, ≠, meet, overlap, intersect, disjoint, on, in, touches, intersection, parallel, aligned
0	Integer Arithmetic	Objects: integers in a range depending on *n* = integer grid size Operations: +, -, ∗, **div**, **mod**, =, ≠, <, ≤, ≥, >

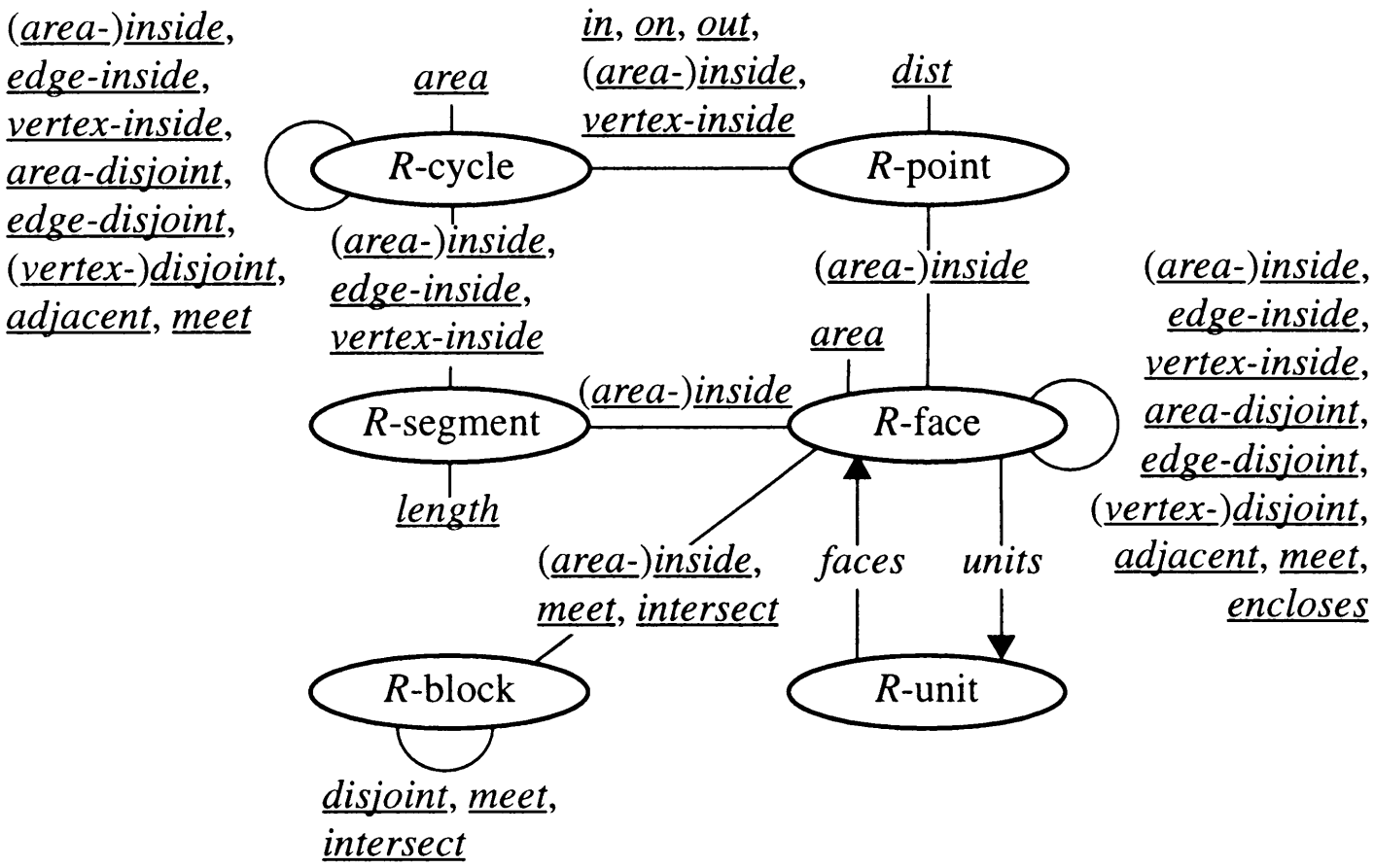
Stufe 1: Geometrische Primitive (für beliebige Segmente)



We explain the primitives informally here: Two N-segments are equal (v) if their end points coincide. Two N-segments meet (iii) if they have exactly one end point in common. They overlap (vi) if they are collinear and share a (partial) N-segment. If they are collinear and do not share a (partial) N-segment, we call them aligned (viii). If they have the same slope, they are parallel (iv). The on (x) primitive tests if an N-point lies on an N-segment; the in (ix) primitive does nearly the same but the N-point must not coincide with one of the end points of the N-segment. An N-segment touches (vii) another N-segment if both N-segments do not overlap and exactly one of the end points of the first N-segment lies in the second N-segment. Two N-segments touch each other if the first N-segment touches the second one, or if the reverse relation holds. If two N-segments have exactly one common point but do neither meet nor touch, they intersect (i). They are disjoint (ii) if they are neither equal nor meet nor touch nor intersect nor overlap. The intersection primitive as the only non-predicate calculates the intersection point of two N-segments and rounds it to the nearest N-point.

Stufe 2: Realm-basierte Strukturen

Neben den Primitiven *R*-Segment und *R*-Punkt werden folgende realm-basierte Strukturen eingeführt und folgende Operationen darauf angeboten (binäre Prädikate für topologische Beziehungen sowie unäre numerische/Konvertierungs-Funktionen):



R-Zyklen

An *R-cycle* c is a set of R-segments $S(c) = \{s_0, ..., s_{m-1}\}$, such that

(i) $\forall\, i \in \{0, ..., m-1\} : s_i$ meets $s_{(i+1) \bmod m}$

(ii) no more than 2 segments from S(c) meet in any point

Beziehungen zwischen beliebigen (N-)Punkten p und einem R-Zyklus c:

- p *on* $c \quad :\Leftrightarrow \exists s \in S(c) : p$ *on* s
- p *in* $c \quad :\Leftrightarrow \neg p$ *on* $c \wedge [*]$ "p liegt im Innern von c"

  Betrachte ausgehend von $p = (p_x, p_y)$ ein vertikal nach oben verlaufendes Sondiersegment $s_p = (p, (p_x, n))$. Als Kriterium für $[*]$ reicht es zu zählen, für wieviele Segmente s von c der rechte, aber nicht der linke Endpunkt *on* s_p liegt[3] oder *intersect* s_p gilt; diese Anzahl muss ungerade sein.

- p *out* $c :\Leftrightarrow \neg\, (p$ *on* $c \vee p$ *in* $c)$

$P_{on}(c), P_{in}(c), P_{out}(c)$ bezeichnen die Mengen aller Punkte, die das jeweilige Prädikat bzgl. c erfüllen. Sei $P(c) := P_{on}(c) \cup P_{in}(c)$.

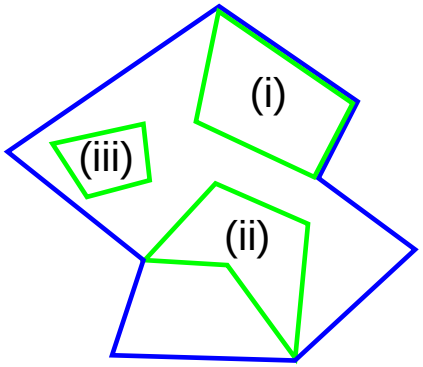
[3] Der rechte Endpunkt eines Segments ist der größere der beiden Endpunkte in (x, y)-lexikographischer Ordnung.

R-Zyklen (Forts.)

c_2 is

- (*area-*)*inside* (i, ii, iii)
- *edge-inside* (ii, iii)
- *vertex-inside* (iii)

c_1.

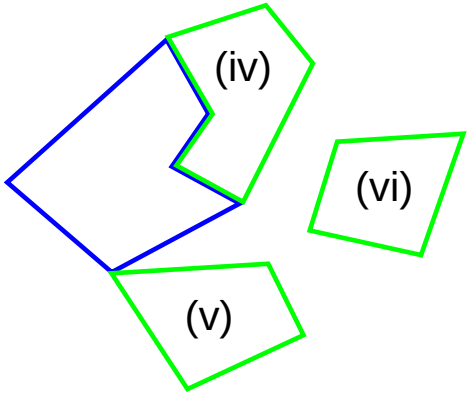


c_1

c_2

c_1 and c_2 are

- *area-disjoint* (iv, v, vi)
- *edge-disjoint* (v, vi)
- (*vertex-*)*disjoint* (vi)



c_1 (*area-*)*inside* c_2 $:\Leftrightarrow$ $P(c_1) \subseteq P(c_2)$

c_1 *edge-inside* c_2 $:\Leftrightarrow$ c_1 *area-inside* c_2
$\wedge\ S(c_1) \cap S(c_2) = \varnothing$

c_1*vertex-inside* c_2 $:\Leftrightarrow$ c_1 *edge-inside* c_2
$\wedge\ P_{on}(c_1) \cap P_{on}(c_2) = \varnothing$

c_1 and c_2 are *area-disjoint* $:\Leftrightarrow$
$P_{in}(c_1) \cap P(c_2) = \varnothing \wedge P_{in}(c_2) \cap P(c_1) = \varnothing$

c_1 and c_2 are *edge-disjoint* $:\Leftrightarrow$
c_1 and c_2 are *area-disjoint* $\wedge S(c_1) \cap S(c_2) = \varnothing$

c_1 and c_2 are (*vertex-*)*disjoint* $:\Leftrightarrow$
c_1 and c_2 are *edge-disjoint* $\wedge P_{on}(c_1) \cap P_{on}(c_2) = \varnothing$

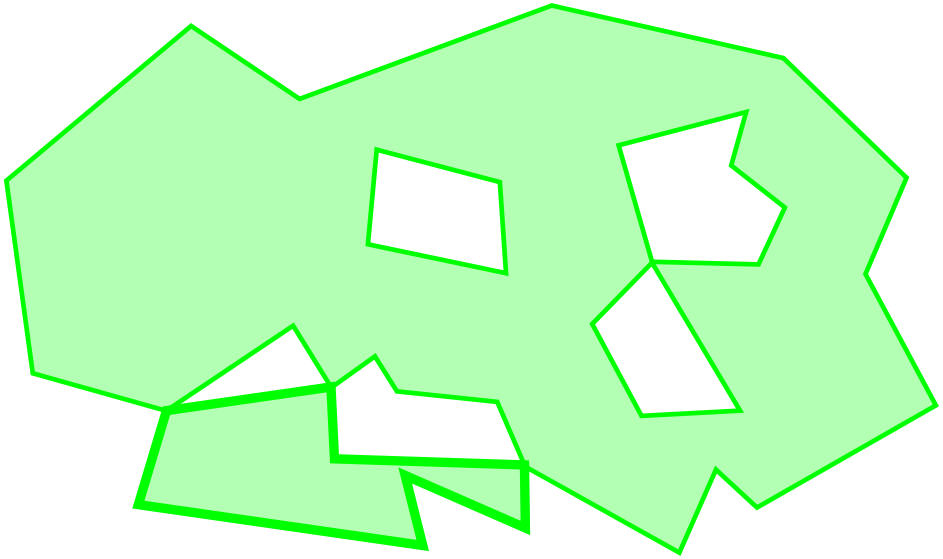
R-"faces"

An *R-face* f is a pair (c, H) where c is an R-cycle and $H = \{h_1, ..., h_m\}$ is a (possibly empty) set of R-cycles such that:

(i) $\forall\, i \in \{1, ..., m\} : h_i$ *edge-inside* c

(ii) $\forall\, i, j \in \{1, ..., m\}, i \neq j : h_i$ and h_j are *edge-disjoint*

(iii) "no other cycle can be formed from the segments of f"



← Region mit Löchern

(nicht zerlegbar)

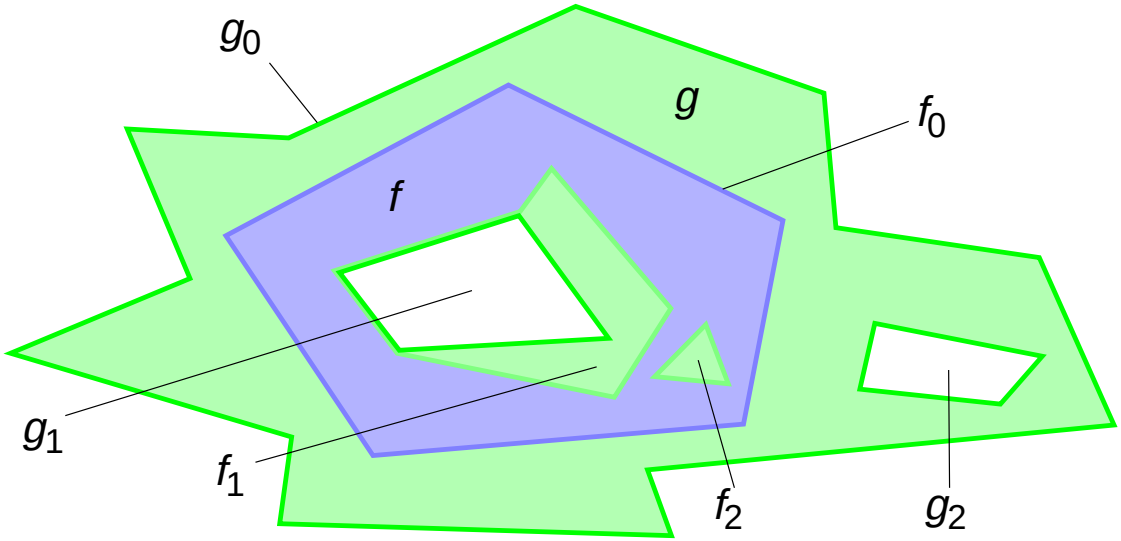
← not 1 face, but 2 faces!

R-"faces" (Forts.)

Let $f = (f_0, F)$ and $g = (g_0, G)$ be two *R*-faces. Then

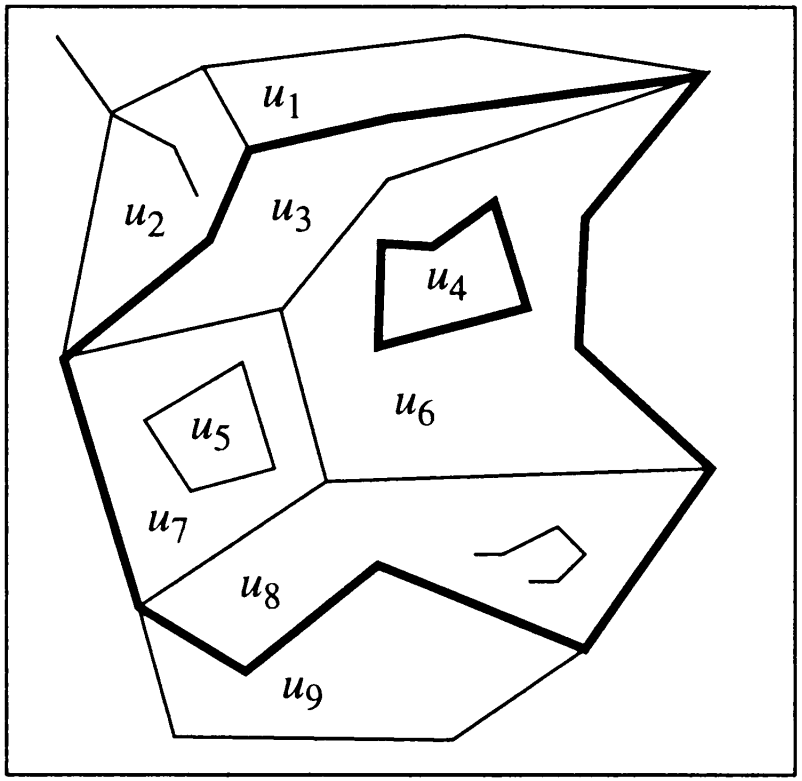
f (*area-*)*inside* $g \quad :\Leftrightarrow \quad f_0$ *area-inside* g_0

$\wedge \forall\, g' \in G : (g'$ *area-disjoint* $f_0 \vee \exists\, f' \in F : g'$ *area-inside* $f')$



Sei $F(R)$ die Menge aller möglichen *R*-faces. Eine *R*-Masche (*R*-unit) ist ein minimales *R*-face, d.h.: $\forall g \in F(R) : g$ *area-inside* $f \Rightarrow g = f$

R-faces vs. R-units



$\leftarrow \{u_3, u_5, u_7, u_6, u_8\}$

Figure 3-17

Figure 3-17 shows an example of a realm with all its R-units u_i and an emphasized R-face which is not an R-unit.

R-faces vs. *R*-units (Forts.)

In der nächsten Stufe werden **Regionen** (ein *regions*-Objekt)

- als Menge F von paarweise kantendisjunkten R-faces
  (minimale Begrenzung, Komponentensicht, strukturierte Sicht)
- *oder* als Menge U von paarweise flächendisjunkten R-units
  (minimale Bausteine, Mengensicht, flache Sicht)

definiert. Diese Auffassungen sind bijektiv ineinander konvertierbar:

$$units(F) := \{\, u \ R\text{-unit} \mid \exists f \in F : \ u \ \underline{area\text{-}inside} \ f \,\}$$

$$faces(U) := \{\, f \ R\text{-face} \mid f \text{ ergibt sich durch “Verschmelzen”}^4 \text{ von } R\text{-units in } U \,\}$$

Es gilt: $faces(units(F)) = F$ und $units(faces(U)) = U$.

[4] Verschmelzen: i.w. durch Weglassen gemeinsamer Segmente, d.h. für je zwei Faces f, g wird das Face $f \triangle g$ mit der Segmentmenge $(S(f) - S(g)) \cup (S(g) - S(f))$ gebildet, usw. iteriert

R-Blöcke vs. *R*-Segmente

Analog lassen sich **Linien** (ein *lines*-Objekt) als Mengen von *R*-Segmenten oder als Menge von disjunkten *R*-**Blöcken** (maximale zusammenhängende Teilgraphen bei Auffassung des Realm als Graph) darstellen.

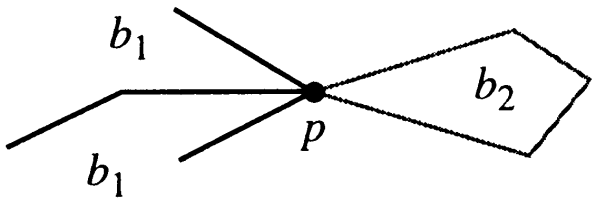
Für zwei R-Blöcke b_1, b_2 sei dabei definiert:

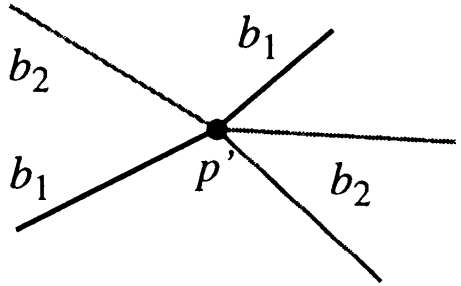
b_1 *disjoint* b_2 $:\Leftrightarrow$ $\forall s_1 \in S(b_1)\ \forall s_2 \in S(b_2) :$ s_1 *disjoint* s_2

b_1 and b_2 *meet* $:\Leftrightarrow$ $\exists\, s \in S(b_1)\ \exists\, t \in S(b_2)$: s and t meet in a meeting point $\wedge$
$\forall\ s \in S(b_1)\ \forall\ t \in S(b_2) : s \neq t \wedge$
(s and t meet in $p \Rightarrow p$ is a meeting point)

b_1 and b_2 *intersect* $:\Leftrightarrow$ $\forall\ s \in S(b_1)\ \forall\ t \in S(b_2) : s \neq t \wedge \exists\, s \in S(b_1)\ \exists\, t \in S(b_2) :$
s and t meet in $p \wedge p$ is not a meeting point

For an *R*-point p we consider the angularly sorted cyclic list L_p of *R*-segments $s \in S(b_1) \cup S(b_2)$ that meet in p. p is called a *meeting point* if L_p is the concatenation of two sublists $L_{p,1}$ and $L_{p,2}$ so that all *R*-segments of $L_{p,1}$ are elements of $S(b_1)$ and all *R*-segments of $L_{p,2}$ are elements of $S(b_2)$, or vice versa. In the Figure p represents and p' does not represent a meeting point.





Operationen mit numerischen Ergebniswerten

$\underline{dist}(p,q)$, $\underline{length}(s)$: wie üblich

$\underline{area}(c)$ für R-Zyklus c mit Segmentmenge
$S(c) = \{s_0, ..., s_{m-1}\}$, $s_i = ((x_i, y_i), (x_{(i+1) \bmod m}, y_{(i+1) \bmod m}))$:

$$\underline{area}(c) = \frac{1}{2} \cdot \left| \sum_{i=0}^{m-1} (x_{(i+1) \bmod m} - x_i) \cdot (y_{(i+1) \bmod m} + y_i) \right|$$

Es werden die positiven bzw. negativen Trapezflächen zwischen x-Achse und den Segmenten "aufaddiert".

$\underline{area}(f)$ für R-face $f = (c, H)$:

$$\underline{area}(f) = \underline{area}(c) - \sum_{h \in H} \underline{area}(h)$$

Stufe 3: Räumliche Datentypen (nur) mit Grundoperationen

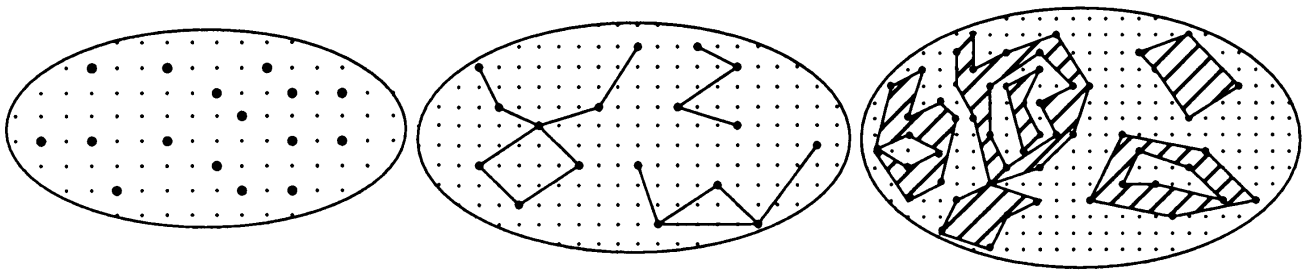
The "flat" view is the following:

> *For a given realm R, an object of type points is a set of R-points, an object of type lines is a set of R-segments, and an object of type regions is a set of R-units.*

The "structured" view, that we shall assume as the formal definition, is as follows:

> *For a given realm R, an object of type points is a set of R-points, an object of type lines is a set of pairwise disjoint R-blocks, and an object of type regions is a set of pairwise edge-disjoint R-faces.*

The first view is conceptually very simple and supports a direct understanding of set operations. The second view is "semantically richer" and shows *lines* and *regions* objects as consisting of a number of *components* (blocks or faces). Moreover, it allows one to express relationships between these components and also emphasizes the representation of the boundary in case of regions. Note that a *regions* object may have holes. Holes are important because (i) they allow for an adequate modelling of area features, and (ii) they make it possible to obtain closure under point set operations. Figure 4-1 illustrates the data types.



a *points* object a *lines* object a *regions* object

The geometric operations can be reduced to the corresponding set-theoretic ones and are defined as follows. Let P_1, P_2 be two *points* objects, L_1, L_2 two *lines* objects, and R_1, R_2 two *regions* objects. Then

$$\textbf{\textit{union}}\ (P_1, P_2) := P_1 \cup P_2$$
$$\textbf{\textit{union}}\ (L_1, L_2) := \mathit{blocks}(S(L_1) \cup S(L_2))$$
$$\textbf{\textit{union}}\ (R_1, R_2) := \mathit{faces}(\mathit{units}(R_1) \cup \mathit{units}(R_2))$$

For ***intersection*** and ***difference*** the definitions are analogous.

The following further primitives are needed. Let F and G be two *regions* objects.

F and G are ***area-disjoint*** $:\Leftrightarrow$ $\forall f \in F\ \forall g \in G$: f and g are *area-disjoint*

F and G are ***adjacent*** $:\Leftrightarrow$ F and G are ***area-disjoint*** $\wedge$
$\exists f \in F\ \exists g \in G$: f and g are *adjacent*

The meaning of the remaining predicates (***area-***)***inside***, ***edge-inside***, ***vertex-inside***, ***edge-disjoint***, (***vertex-***)***disjoint***, ***meet*** should be clear; their formal definitions are left to the reader. We define two further predicates ***intersect*** and ***encloses***:

F and G ***intersect*** $:\Leftrightarrow$ $\mathit{units}(F) \cap \mathit{units}(G) \neq \emptyset$

F ***encloses*** G $:\Leftrightarrow$ $\forall g \in G\ \exists f \in F$: f *encloses* g

Let P and Q be two *points* objects.

P and Q are ***disjoint*** $:\Leftrightarrow$ $P \cap Q = \emptyset$

Let K and L be two *lines* objects.

K and L ***disjoint***	$:\Leftrightarrow$	$\forall\, k \in K\, \forall\, l \in L : k$ and l are *disjoint*
K and L ***meet***	$:\Leftrightarrow$	$(\forall\, k \in K\, \forall\, l \in L : k$ and l are *disjoint* $\vee\ k$ and l *meet*$) \wedge$ $(\exists\, k \in K\, \exists\, l \in L : k$ and l *meet*$)$
K and L ***intersect***	$:\Leftrightarrow$	$(\forall\, k \in K\, \forall\, l \in L : k$ and l are *disjoint* $\vee$ k and l *intersect*$) \wedge$ $(\exists\, k \in K\, \exists\, l \in L : k$ and l *intersect*$)$

Let P be a *points* object, L a *lines* object, F a *regions* object, and v, w *lines* or *regions* objects.

P (***area-***)***inside*** F	$:\Leftrightarrow$	$\forall\, p \in P\, \exists\, f \in F : p$ *area-inside* f
L (***area-***)***inside*** F	$:\Leftrightarrow$	$\forall\, l \in L\, \exists\, f \in F : l$ *area-inside* f
L and F ***meet***	$:\Leftrightarrow$	$\forall\, l \in L\, \forall\, f \in F : \neg\, l$ *area-inside* $f \wedge$ $\exists\, l \in L\, \exists\, f \in F : l$ and f *meet*
L and F ***intersect***	$:\Leftrightarrow$	$\exists\, l \in L\, \exists\, f \in F : l$ and f *intersect*
P ***on_border_of*** v	$:\Leftrightarrow$	$\forall\, p \in P\, \exists\, s = (q_1, q_2) \in S(v) :$ $p = q_1 \vee p = q_2$
v ***border_in_common*** w	$:\Leftrightarrow$	$\exists\, s \in S(v)\, \exists\, t \in S(w) : s = t$

!

Stufe 4: Vollständige räumliche Datentypen

0. Typmengen (als Wertevorräte für Typvariablen)

DATA = {*int*, *real*, *bool*, ...}

EXT = {*lines*, *regions*}

GEO = {*points*, *lines*, *regions*}

$regions^{area-disjoint}$ steht für alle Auswahlen von *area-disjoint* *regions*-Objekten.

1. Räumliche Beziehungen

$\forall$ *geo* in GEO. $\forall$ *ext*, ext_1, ext_2 in EXT. $\forall$ *area* in $regions^{area\text{-}disjoint}$

$geo \times geo$	$\rightarrow$	*bool*	**=, ≠, disjoint**
$geo \times regions$	$\rightarrow$	*bool*	**inside**
$regions \times regions$	$\rightarrow$	*bool*	**area_disjoint, edge_disjoint, edge_inside, vertex_inside**
$ext_1 \times ext_2$	$\rightarrow$	*bool*	**intersects, meets**
$area \times area$	$\rightarrow$	*bool*	**adjacent, encloses**
$points \times ext$	$\rightarrow$	*bool*	**on_border_of**
$ext_1 \times ext_2$	$\rightarrow$	*bool*	**border_in_common**

(i.w. Übernahme aus Stufe 3)

2. Operationen, die SDT-Werte liefern

$\forall$ *geo* in GEO. $\forall$ *ext*, ext_1, ext_2 in EXT.

points × *points*	→	*points*	**intersection**
lines × *lines*	→	*points*	**intersection**
regions × *regions*	→	*regions*	**intersection**
regions × *lines*	→	*lines*	**intersection**
geo × *geo*	→	*geo*	**plus, minus**
$ext_1 \times ext_2$	→	*lines*	**common_border**
ext	→	*points*	**vertices**
regions	→	*lines*	**contour**
lines	→	*regions*	**interior**

übernehmen i.w. Operationen; neu:

$$\text{intersection}(R, L) := \mathit{blocks}(\{\, s \in S(L) \mid \exists f \in R : s \ \underline{\mathit{inside}}\ f \,\})$$

(für R:*regions*, L:*lines*)

2. Operationen, die SDT-Werte liefern (Forts.)

für L, L_1, L_2 : *lines*, R, R_1, R_2 : *regions*, v : *lines* oder *regions*

common_border(L_1, L_2) := intersection(L_1, L_2) = $blocks(S(L_1) \cap S(L_2))$

common_border(R, L) := $blocks(S(R) \cap S(L))$

common_border(R_1, R_2) := $blocks(S(R_1) \cap S(R_2))$

vertices(v) := $\{ p \mid \exists s \in S(v) : s = (p, q) \}$

contour(R) := $blocks(\bigcup_{i=1...n} S(c_i))$

falls $R = \{(c_1, H_1), ..., (c_n, H_n)\}$ (Menge von faces)

interior(L) := $faces(\bigcup_{c \in cycles(S(L))} S(c) - \{s \in S(L) \mid \exists c \in cycles(S(L)) : s$ *edge-inside* $c\})$

3. Operationen, die numerische Werte liefern

$\forall\ geo, geo_1, geo_2$ in GEO.

geo	$\rightarrow$	*int*	**no_of_components**
$geo_1 \times geo_2$	$\rightarrow$	*real*	**dist**
geo	$\rightarrow$	*real*	**diameter**
lines	$\rightarrow$	*real*	**length**
regions	$\rightarrow$	*real*	**area, perimeter**

$\text{no\_of\_components}(g) := |g|$

(g als Menge von Punkten, Blöcken oder faces)

z.B. $\text{dist}(L_1, L_2) := \min\{\, \text{dist}(s_1, s_2) \mid s_i \in S(L_i),\ i = 1, 2 \,\}$

$\text{diameter}(g) := \max\{\, \text{dist}(p, q) \mid p, q \in \text{vertices}(g) \,\}$

$\text{area}(R) := \sum_{f \in R} \text{area}(f)$ (length, perimeter analog)

4. Operationen auf Mengen von (Datenbank-)Objekten

$\forall\ obj, obj_1, obj_2$ in OBJ. $\forall\ geo, geo_1, geo_2$ in GEO.
$\forall\ area_1, area_2$ in $\underline{regions}^{\textbf{area-disjoint}}$. $\forall\ data_i$ in DATA. $\forall\ geo_j$ in GEO.

$\underline{set}(obj) \times (obj \to geo)$	$\to$	geo	**sum** (union)
$\underline{set}(obj) \times (obj \to geo_1) \times geo_2$	$\to$	$\underline{set}(obj)$	**closest**
$\underline{set}(obj) \times (obj \to geo) \times \underline{ident}$	$\to$	$\underline{set}(o\text{: OBJ})$	**decompose**
$\underline{set}(obj_1) \times (obj_1 \to area_1) \times \underline{set}(obj_2) \times (obj_2 \to area_2) \times \underline{ident}$	$\to$	$\underline{set}(o\text{: OBJ})$	**overlay**
$\underline{set}(obj) \times (obj \to data_i)^+ \times (obj \to geo_j)^+$	$\to$	$\underline{set}(o\text{: OBJ})$	**fusion**

Diese Operationen erfordern eine geeignete DBMS-Schnittstelle.

z.B. closest(Objektmenge O, räumliches Attribut $attr, g$)
$:= \{\, o \in O \mid \text{dist}(g, attr(o)) \text{ minimal} \,\}$

Zur Implementierung realm-basierter räumlicher Datentypen

Jeder *deskriptiv definierte* Operator wird in evtl. mehrere *ausführbare* Operatoren übersetzt. Diese können sich in den behandelten Datentypen oder verwendeten Algorithmen unterscheiden.

I.w. lassen sich hier folgende Paradigmen für geometrische Algorithmen anwenden:

- "parallel traversal" (PT): merge-ähnlicher Durchlauf durch sortierte Sequenzen
- "plane sweep" (PS)
- Graphalgorithmen (G)

Dabei ist die Realm-Basierung auszunutzen, die insbesondere die Anwendung des PS-Paradigmas vereinfacht, weil keine Schnitte mehr dynamisch aufgefunden werden brauchen. Passende sortierte Repräsentationen ermöglichen zudem oft die Anwendung des noch einfacheren PT-Paradigmas. Schließlich erlaubt die Auffassung eines Realms als Graph auch die Verwendung von (G)-Algorithmen.

Die folgende Tabelle gibt einen Überblick über die zugeordneten Operatoren und listet die Zeitkomplexitäten (TC) in Abhängigkeit von n (Gesamtgröße der Operanden), m (Größe des *regions*-Operanden, falls vorhanden) und k (Ausgabegröße). Die Präfixe 'pp_', 'rl_' der Operatoren nennen die behandelten Datentypen (points, lines, regions).

Descriptive Operator	Executable Operator	PT	PS	G	TC
$geo \times geo \rightarrow \underline{bool}$ =	**pp_equal, ll_equal, rr_equal**	x			$O(n)$
≠	**pp_unequal, ll_unequal, rr_unequal**	x			$O(n)$
disjoint	**pp_disjoint, ll_disjoint**	x			$O(n)$
	rr_disjoint		x		$O(n \log n)$
$geo \times \underline{regions} \rightarrow \underline{bool}$ **inside**	**pr_inside, lr_inside**		x		$O(n \log m)$
	rr_inside		x		$O(n \log n)$
$\underline{regions} \times \underline{regions} \rightarrow \underline{bool}$ **area_disjoint**	**rr_area_disjoint**		x		$O(n \log n)$
edge_disjoint	**rr_edge_disjoint**		x		$O(n \log n)$
edge_inside	**rr_edge_inside**		x		$O(n \log n)$
vertex_inside	**rr_vertex_inside**		x		$O(n \log n)$
$ext_1 \times ext_2 \rightarrow \underline{bool}$ **intersects**	**ll_intersects**	x			$O(n)$
	lr_intersects, rl_intersects		x		$O(n \log m)$
	rr_intersects		x		$O(n \log n)$
meets	**ll_meets**	x			$O(n)$
	lr_meets, rl_meets		x		$O(n \log m)$
	rr_meets		x		$O(n \log n)$
border_in_common	**ll_border_in_common, lr_border_in_common, rl_border_in_common, rr_border_in_common**	x			$O(n)$

Descriptive Operator	Executable Operator	PT	PS	G	TC
area × *area* → *bool* **adjacent**	**rr_adjacent**		x		$O(n \log n)$
area × *area* → *bool* **encloses**	**rr_encloses**		x		$O(n \log n)$
points × *ext* → *bool* **on_border_of**	**pl_on_border_of, pr_on_border_of**	x			$O(n)$
points × *points* → *points* **intersection**	**pp_intersection**	x			$O(n + k \log k)$
lines × *lines* → *points* **intersection**	**ll_intersection**	x			$O(n + k \log k)$
regions × *regions* → *regions* **intersection**	**rr_intersection**		x		$O(n \log n)$
regions × *lines* → *lines* **intersection**	**rl_intersection**		x		$O(n \log m + k \log k)$
geo × *geo* → *geo* **plus**	**pp_plus, ll_plus**	x			$O(n + k \log k)$
	rr_plus		x		$O(n \log n)$
geo × *geo* → *geo* **minus**	**pp_minus, ll_minus**	x			$O(n + k \log k)$
	rr_minus		x		$O(n \log n)$
$ext_1 \times ext_2$ → *lines* **common_border**	**ll_common_border, lr_common_border, rl_common_border, rr_common_border**	x			$O(n + k \log k)$

Descriptive Operator	Executable Operator	PT	PS	G	TC
ext → *points* **vertices**	**l_vertices, r_vertices**	x			$O(n + k \log k)$
regions → *lines* **contour**	**r_contour**			x	$O(n \log n)$ / $O(k \log k)$
lines → *regions* **interior**	**l_interior**			x	$O(n \log n)$
geo → *int* **no_of_components**	**p_no_of_components**	x			$O(n)$ / $O(1)$
	l_no_of_components, r_no_of_components			x	$O(n \log n)$ / $O(1)$
$geo_1 \times geo_2$ → *real* **dist**	**pp_dist, pl_dist, pr_dist, lp_dist, ll_dist, lr_dist, rp_dist, rl_dist, rr_dist**				
geo → *real* **diameter**	**p_diameter, l_diameter, r_diameter**	*special algorithm*			$O(n)$ / $O(1)$
lines → *real* **length**	**l_length**	x			$O(n)$ / $O(1)$
regions → *real* **area**	**r_area**	x			$O(n)$ / $O(1)$
perimeter	**r_perimeter**	x			$O(n)$ / $O(1)$

Realm-DB-Interface

Für alle realm-basierten Operationen wird folgendes Realm-Teilschema im Rahmen einer räumlichen Datenbank angenommen:

- Die DB- bzw. Anwendungsobjekte haben räumliche Komponenten ($\rightsquigarrow$ Relationship).
- Jede räumliche Komponente *sc* (spatial component) ist von einem räumlichen Datentyp der obersten Stufe, hat eine ID *scid* (vom Datentyp *SCID*) und trägt – redundant – ihren Gesamtgeometriewert und weitere vorausberechnete Größen, z.B. Länge/Fläche.
- Jede räumliche Komponente ist aus Realm-Objekten, d.h. Punkten und Segmenten, zusammengesetzt ($\rightsquigarrow$ Relationship).
- Jedes Realm-Objekt r hat eine ID *roid* (vom Datentyp *ROID*) und trägt seine Geometrie, dargestellt als Punkt oder Punktpaar.
- Über das Zusammensetzungs-Relationship seien zu einer räumlichen Komponente *sc* alle zugehörigen Realm-Objekte $roids(sc)$ und zu einem Realm-Objekt r alle dieses enthaltenden räumlichen Komponenten $scids(r)$ zugreifbar.

Realm-DB-Interface (Forts.)

Damit kann ein Realm R mit seinem Kontext dargestellt werden als:

$$\{ (r, roid(r), scids(r)) \mid r \in R \} : Realm$$

Diese Realm-Repräsentation sei u.a. über einen räumlichen Index zugreifbar, der z.B. einen schnellen Zugriff auf alle R-Segmente und R-Punkte innerhalb eines gegebenen Rechteckfensters erlaubt (siehe Operation *Window* unten, zu Indexen siehe Kapitel 3).

Indirekt kann so – per Index und *scids*-Komponente – auch auf die DB-Objekte zugegriffen werden.

Gewartet wird diese Realm-Repräsentation durch nachfolgende Update- und Zugriffs-Operationen:

Realm-DB-Interface (Forts.)

sorts *Realm, Point, Segment, RealmObject, ROID, SCID, Rectangle, Bool, Integer*

ops

InsertNPoint:	$Realm \times Point$	$\rightarrow$	$Realm \times ROID \times (SCID \times (Segment \times ROID)^*)^*$
InsertNSegment:	$Realm \times Segment$	$\rightarrow$	$Realm \times (Segment \times ROID)^* \times (SCID \times (Segment \times ROID)^*)^* \times Bool$
SplitRSegment:	$Realm \times ROID \times Point$	$\rightarrow$	$Realm \times ROID \times (SCID \times (Segment \times ROID)^*)^* \times Bool$
Delete:	$Realm \times ROID$	$\rightarrow$	$Realm \times Bool$

The operation *InsertNSegment* takes a realm and an N-segment. It returns (i) the modified realm, (ii) a list of segments with their *roids* which may contain either the original segment as the only element or a redrawing of this segment, and (iii) a possibly empty set of database segments that must be redrawn. Here the inserted segment may need redrawing because it traverses R-points, overlaps R-segments, or intersects R-segments. Only those R-segments need redrawing that are intersected by this segment. For each such R-segment its *scid* together with a list of pairs $(s, roid(s))$ (where s is a segment of the redrawing) is returned. The last parameter (iv) indicates whether insertion was performed. It was rejected, if not both end points of the segment were present in the realm.

Realm-DB-Interface (Forts.)

The second group of operations supports the management of the two-way linking between realm objects and components of spatial attribute values in the database:

Register:	$Realm \times ROID \times SCID$	$\rightarrow$	$Realm$
Unregister:	$Realm \times ROID \times SCID$	$\rightarrow$	$Realm$
GetSCIDs:	$Realm \times ROID$	$\rightarrow$	$SCID^*$
GetRealmObject:	$Realm \times ROID$	$\rightarrow$	$RealmObject$

Here *Register* informs a realm object *roid* about a spatial component *scid* depending on it. *Unregister* removes such an information. *GetSCIDs* returns the *scids* of spatial components depending on a given *roid*, *GetRealmObject* returns the geometry.

The last group of operations supports the selection of realm objects for the construction of spatial objects:

Window:	$Realm \times Rectangle$	$\rightarrow$	$(RealmObject \times ROID)^*$
Identify:	$Realm \times Point \times Integer$	$\rightarrow$	$ROID \times Bool$

Window returns all realm objects together with their *roid* that are inside or intersect a given rectangular window. *Identify* tries to identify a realm object close to the *N*-point given as a parameter. The number given as a third parameter controls the "pick distance". A *roid* (possibly undefined) is returned together with an indication whether identification was successful.

Sortierungen räumlicher Objekte

Für zwei Gitter-Punkte $p_1 = (x_1, y_1), p_2 = (x_2, y_2) \in P_N$ lässt sich eine ((x, y)-lexikographische) Ordnung wie folgt definieren:

$$p_1 < p_2 \;:\Leftrightarrow\; x_1 < x_2 \vee (x_1 = x_2 \wedge y_1 < y_2)$$

Zu jedem normalisierten Segment $s = (p, q), p < q$ (d.h. p "linker" und q "rechter Endpunkt) lassen sich zwei **Halbsegmente**

$$h = (s, d),\; d \in \{left, right\}$$

bilden. Dadurch wird ein Zielpunkt (*destination point*) ausgezeichnet:

$$\text{für } h = (s, d),\ s = (p, q): \quad dp(h) := p, \text{ falls } d = left, \quad q \text{ sonst}$$

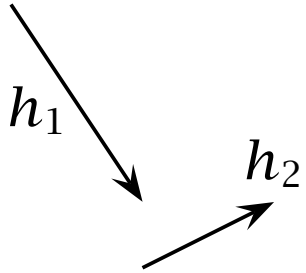
Auf Halbsegmenten lässt sich wie folgt eine Ordnung definieren:

$$\begin{aligned} h_1 < h_2 \;:\Leftrightarrow\; & dp(h_1) < dp(h_2) \vee (dp(h_1) = dp(h_2) \wedge \\ & ((d_1 = right \wedge d_2 = left) \vee (d_1 = d_2 \wedge rot(h_1, h_2)))) \end{aligned}$$

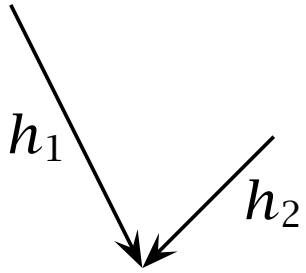
Dabei sei $rot(h_1, h_2)$ wahr gdw. sich h_1 gegen den Uhrzeigersinn mit einem Winkel $0 < \alpha \leq 180°$ so drehen lässt, dass h_2 überlappt wird.

Sortierungen räumlicher Objekte (Forts.)

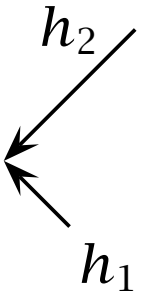
Z. B. gilt in den folgenden Fällen $h_1 < h_2$:



$dp(h_1) < dp(h_2)$

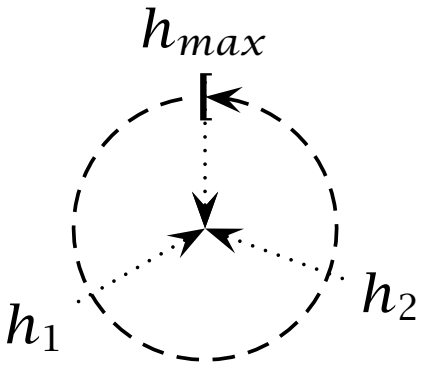


$d_1 = right \wedge d_2 = left$



$rot(h_1, h_2)$

Segmente mit gleichem Zielpunkt sind wie folgt angeordnet:



$h_1 < h_2 < h_{max}$

Repräsentation räumlicher Objekte

Jedes *points*- Objekt v wird durch die sortierte Sequenz aller enthaltenen Punkte (Menge $P(v)$) dargestellt, jedes *lines*- bzw. *regions*-Objekt v durch die sortierte Sequenz aller enthaltenen Segmente, jeweils in beiden Richtungen (Menge $H(v)$ der Halbsegmente). Jede Sequenz trägt eine aktuelle Position (Cursor).

Hilfsmethoden für den parallelen Durchlauf von zwei sortierten Sequenzen v, w:

..._select_first$(v, w, object, status)$,
..._select_next$(v, w, object, status)$

melden, welche der beiden Sequenzen das erste minimale bzw. das nächste minimale Element liefert, und welche Sequenz bereits vollständig durchlaufen worden ist (nachdem sie vorher die Cursor initialisiert bzw. abhängig vom vorigen Ergebnis einen oder beide Cursor weitergesetzt haben):

object = *none* | *first* | *second* | *both*
status = *end_of_none* | *end_of_first* | *end_of_second* | *end_of_both*

get_pt bzw. *get_hs* holen den aktuellen Punkt bzw. das aktuelle Halbsegment.

Bei einer *points*-Sequenz und einer *lines*- oder *regions*-Sequenz ("*pr_select_...*") werden nur Punkte und (verschiedene) Zielpunkte verglichen.

Parallel-Traversal (PT) – basierte Algorithmen

algorithm pp_plus
input: Two *points* objects P_1 and P_2
output: A *points* object P_{new} containing all points of P_1 and P_2
begin
  P_{new} := *new*();
  pp_select_first(P_1, P_2, *object*, *status*);
  while *status* ≠ *end_of_both* **do**
    if *object* = *first* **then** p := *get_pt*(P_1)
    else if *object* = *second* **then** p := *get_pt*(P_2)
    else if *object* = *both* **then** p := *get_pt*(P_1)
    end-if;
    P_{new} := *insert*(P_{new}, p);
    pp_select_next(P_1, P_2, *object*, *status*);
  end-while;
  return P_{new}
end pp_plus.

Parallel-Traversal (PT) – basierte Algorithmen (Forts.)

algorithm pr_on_border_of
input: A *points* object P and a *regions* object R
output: *true*, if $\forall\ p \in P(P)\ \exists\ h \in \tilde{H}(R) : p = dp(h)$
false, otherwise
begin
pr_select_first(P, R, *object*, *status*);
while (*object* ≠ *first*) **and** (*status* = *end_of_none*) **do**
pr_select_next(P, R, *object*, *status*);
end-while;
return (*object* ≠ *first*) **and** (*status* ≠ *end_of_second*)
end pr_on_border_of.

Parallel-Traversal (PT) – basierte Algorithmen (Forts.)

```
algorithm ll_intersects
input: Two lines objects L1 and L2
output: true, if no common segment exists, but a common point
                which is not a meeting point
        false, otherwise
begin
  ll_select_first(L1, L2, object, status);
  if object = first then act_dp := dp(get_hs(L1))
  else if object = second then act_dp := dp(get_hs(L2))
  end-if;
  old_obj := object; found := false; count := 0;
  while (status = end_of_none) and (object ≠ both) do
    ll_select_next(L1, L2, object, status);
    if (status ≠ end_of_both) and (object ≠ both) and not found then
      if object = first then
        new_dp := dp(get_hs(L1))
      else if object = second then
        new_dp := dp(get_hs(L2))
      end-if;
      if new_dp ≠ act_dp then (* new point *)
        act_dp := new_dp;
        count := 0;
        old_obj := object;
      else if object ≠ old_obj then (* object switch *)
        count := count + 1;
        old_obj := object;
        found := found or (count > 2);
      end-if;
    end-if;
  end-while;
  return found and (object ≠ both);
end ll_intersects.
```

Plane-Sweep (PS) – basierte Algorithmen

Sweep-Status S = sortierte Sequenz von Segmenten mit aktueller Position und Attributen, im folgenden formal dargestellt als

$$S = \{\, (pos, \langle s_1, ..., s_n \rangle, \langle a_1, ..., a_n \rangle \mid s_i \text{ Realm-Segment},$$
$$a_i \text{ Menge von Attributwerten}, \; s_i \; \textit{über} \; s_{i-1} \;\; \text{für } i = 1, ..., n \,\}$$

Dabei gilt s_i *über* s_{i-1} gdw. der Durchschnitt der x-Intervalle von s_i und s_{i-1} nichtleer ist und an den Enden des gemeinsamen x-Intervalls

$$y\text{-Koordinate}(s_i) \geq y\text{-Koordinate}(s_{i-1})\,.$$

Analog *unter*. — Wegen der Realm-Eigenschaften kommen keine Schnitte und Überlappungen der Segmente vor.

Auch *points*-, *lines*- und *regions*-Objekte können zu jedem ihrer Punkte bzw. Halbsegmente Attributwerte tragen.

Aktionen für Sweep-Status S und Segmente s:

add_left(S, s): fügt Segmentkomponente eines linken Halbsegments in S ein
del_right(S, s): löscht Segmentkomponente eines rechten Halbsegments aus S

Plane-Sweep (PS) – basierte Algorithmen (Forts.)

Weitere Hilfsoperationen:

new_sweep(): Generator

pred_of_s(S, s): setzt aktuelle Position von S auf das nächste Segment in S unter dem Segment s

pred_of_p(S, p): setzt aktuelle Position von S auf das nächste Segment in S unter dem Punkt p

current_exists(S), *pred_exists*(S): aktuelle Position bzw. Vorgängerposition besetzt ?

set_attr($S, ...$), *get_attr*(S), *get_pred_attr*(S):
setzt/liest Attributwerte an aktueller/Vorgänger-Position

Ereignisliste: hier statisch durch sortierte Argumentobjekt(punkt)e vorgegeben, insbesondere von links nach rechts

Realisierung des Sweep-Status: z.B. mit AVL-Baum

Plane-Sweep (PS) – basierte Algorithmen: *InsideAttribute*

algorithm *InsideAttribute*
input: A *regions* object $R = (i, < h_1, ..., h_n >, < a_1, ..., a_n >)$
output: $R' = (i', < h_1, ..., h_n >, < a_1', ..., a_n' >)$ with $a_i' = a_i$ if h_i is a right halfsegment, $a_i' = a_i \cup \{InsideAbove\}$ if h_i is a left halfsegment and the area of R lies above or left of the segment component of h_i, and $a_i' = a_i \setminus \{InsideAbove\}$ otherwise (for $1 \leq i \leq n$)
begin
  S := *new_sweep*();
  R := *select_first*(R);

Idee: Plane-Sweep über (die Segmente des) *regions*-Objekts R (z.B. am Ende der Konstruktion von R, so dass *InsideAbove* vorausberechnet wird)

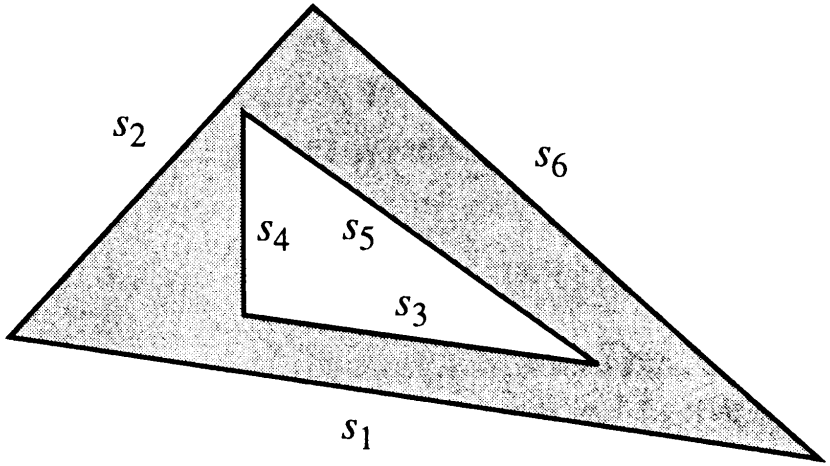
Jedes ungerade Segment im Sweep-Status erhält das *InsideAbove*-Attribut; genauer jedes Segment erhält beim Einfügen in den Sweep-Status das Attribut, falls es das erste Segment ist oder falls sein Vorgänger nicht das Attribut hat.

```
    while not end_of_hs(R) do
        h := get_hs(R); (* Let h = (s, d) *)
        attr := get_attr(R);
        if d = left then
            S := add_left(S, s);
            if not pred_exists(S) then
                S := set_attr(S, {InsideAbove});
                R := update_attr(R, attr ∪ {InsideAbove});
            else
                pred_attr := get_pred_attr(S);
                if InsideAbove ∈ pred_attr then
                    S := set_attr(S, ∅);
                    R := update_attr(R, attr \ {InsideAbove});
                else
                    S := set_attr(S, {InsideAbove});
                    R := update_attr(R, attr ∪ {InsideAbove});
                end-if;
            end-if;
        else
            S := del_right(S, s);
        end-if;
    R := select_next(R);
    end-while;
end InsideAttribute.
```

Plane-Sweep (PS) – basierte Algorithmen: *InsideAttribute* (Forts.)

Beispiel:

We consider the following *regions* object R:



Let $h_i^l = (s_i, \mathit{left})$ and $h_i^r = (s_i, \mathit{right})$ denote the left and right halfsegments belonging to the segments s_i for $1 \le i \le 6$. We assume that no attributes have been assigned to the halfsegments of R. Then R has the following structure after the call of *select_first*:

$$R = (1, < h_1^l, h_2^l, h_3^l, h_4^l, h_4^r, h_5^l, h_2^r, h_6^l, h_5^r, h_3^r, h_6^r, h_1^r >, < \emptyset, ..., \emptyset >)$$

Figure 5-2 shows in the first column the halfsegment currently processed within the while-loop and in the second column the effect of this process on the sweep line status.

After the execution of algorithm *InsideAttribute*, the *regions* object R has the following structure:

$$R = (0, \; < h_1^l, h_2^l, h_3^l, h_4^l, h_4^r, h_5^l, h_2^r, h_6^l, h_5^r, h_3^r, h_6^r, h_1^r >,$$
$$< \{I\}, \emptyset, \emptyset, \{I\}, \emptyset, \{I\}, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset >)$$

Plane-Sweep (PS) – basierte Algorithmen: *InsideAttribute* (Forts.)

Halfsegment	Sweep line status S
-	$(0, \quad \Diamond, \quad \Diamond)$
h_1^l	$(1, \quad < s_1 >, \quad < \{I\} >)$
h_2^l	$(2, \quad < s_1, s_2 >, \quad < \{I\}, \varnothing >)$
h_3^l	$(2, \quad < s_1, s_3, s_2 >, \quad < \{I\}, \varnothing, \varnothing >)$
h_4^l	$(3, \quad < s_1, s_3, s_4, s_2 >, \quad < \{I\}, \varnothing, \{I\}, \varnothing >)$
h_4^r	$(0, \quad < s_1, s_3, s_2 >, \quad < \{I\}, \varnothing, \varnothing >)$
h_5^l	$(3, \quad < s_1, s_3, s_5, s_2 >, \quad < \{I\}, \varnothing, \{I\}, \varnothing >)$
h_2^r	$(0, \quad < s_1, s_3, s_5 >, \quad < \{I\}, \varnothing, \{I\} >)$
h_6^l	$(4, \quad < s_1, s_3, s_5, s_6 >, \quad < \{I\}, \varnothing, \{I\}, \varnothing >)$
h_5^r	$(0, \quad < s_1, s_3, s_6 >, \quad < \{I\}, \varnothing, \varnothing >)$
h_3^r	$(0, \quad < s_1, s_6 >, \quad < \{I\}, \varnothing >)$
h_6^r	$(0, \quad < s_1 >, \quad < \{I\} >)$
h_1^r	$(0, \quad \Diamond, \quad \Diamond)$

Figure 5-2

Plane-Sweep (PS) – basierte Algorithmen: *pr_inside*
[Anwendung des *InsideAbove*-Attributs]

```
algorithm pr_inside
input: A points object P and a regions object R
output: true, if all points of P lie in the area of R
        false, otherwise
begin
  S := new_sweep();
  inside := true;
  pr_select_first(P, R, object, status);
  while (status ≠ end_of_first) and inside do
    if (object = both) or (object = second) then
      h := get_hs(R); (* Let h = (s, d). *)
      attr := get_attr(R);
      if d = left then
        S := add_left(S, s);
        if InsideAbove ∈ attr then
          S := set_attr(S, {InsideAbove});
        end-if
      else
        S := del_right(S, s);
      end-if
    else
      S := pred_of_p(S, get_pt(P));
      if current_exists(S)
        then inside := (InsideAbove ∈ get_attr(S))
        else inside := false
      end-if
    end-if;
    pr_select_next(P, R, object, status);
  end-while;
  return inside;
end pr_inside.
```

Plane-Sweep (PS) – basierte Algorithmen: *rl_intersection*
[Weitere Anwendung des *InsideAbove*-Attributs]

```
algorithm rl_intersection
input: A lines object L and a regions object R
output: A new lines object L_new containing all halfsegments of L
        whose segment components lie in R
begin
  L_new := new();
  S := new_sweep();
  lr_select_first(L, R, object, status);
  while status = end_of_none do
    if (object = both) or (object = second) then
      h := get_hs(R); (* Let h = (s, d). *)
      attr := get_attr(R);
      if d = left then
        S := add_left(S, s);
        if InsideAbove ∈ attr then
          S := set_attr(S, {InsideAbove});
        end-if
      else
        S := del_right(S, s);
      end-if
    end-if;
```

Plane-Sweep (PS) – basierte Algorithmen: *rl_intersection* (Forts.)

```
        end-if;
        if object = both then
            h := get_hs(L);
            L_new := insert(L_new, h);
        else if object = first then
            h := get_hs(L); (* Let h = (s, d). *)
            S := pred_of_s(S, s);
            if current_exists(S) and (InsideAbove ∈ get_attr(S)) then
                L_new := insert(L_new, h);
            end-if;
        end-if;
        lr_select_next(L, R, object, status);
    end-while;
    return L_new;
end rl_intersection.
```

Weitere Plane-Sweep-Algorithmen und Attributierungen

Für Operationen über zwei oder mehreren *regions*-Objekten:

Jedes beteiligte Segment s erhält als Attribut eine Klassifikation

$$ovl(s) = (m/n)$$

wobei n = #überlappende (*overlapping*) Regionen über (oder links von) s und m = #überlappende Regionen unter (oder rechts von) s.

Bei *rr*-Operationen kann dieses Attribut per Plane-Sweep während der Operation berechnet werden. Zu überprüfen bleibt z.B.

R_1 *rr_inside* R_2 gdw.

$$\begin{aligned} &\forall s \in S(R_1) - S(R_2): && ovl(s) \in \{(1/2),(2/1)\} \\ \wedge\ &\forall s \in S(R_2) - S(R_1): && ovl(s) \in \{(0/1),(1/0)\} \\ \wedge\ &\forall s \in S(R_1) \cap S(R_2): && ovl(s) \in \{(0/2),(2/0)\} \end{aligned}$$

R_1 *rr_intersects* R_2 gdw.

$$\exists s \in S(R_1) \cup S(R_2): \ ovl(s) \in \{(0/2),(1/2),(2/1),(2/0)\}$$

Graphen-Algorithmen

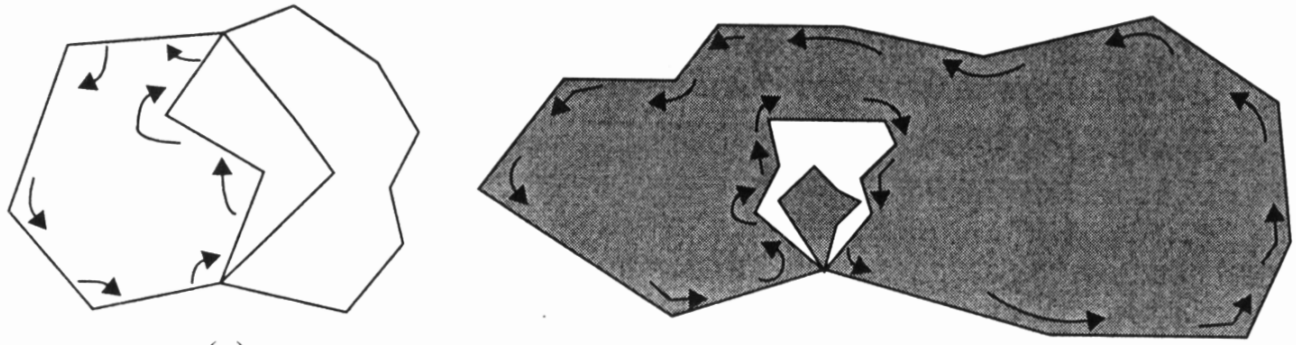
Die betroffenen Objekte werden als ein planarer Graph aufgefasst.

Verwendet werden Adjazenzlisten zu jedem Graphknoten, genauer an jedem Punkt die Liste der eingehenden Halbsegmente in der üblichen Ordnung.

Zudem wird von jedem Halbsegment auf das zugehörige Partnerhalbsegment verwiesen, wodurch sich Graphkanten verfolgen lassen.

Einblick in einen Graphen-Algorithmus: *r_contour* → → → → →

The main problem of the algorithm for **r_contour** is the assignment of the segments to the correct *outer cycles* and *hole cycles* which according to the face definition is unique. According to that definition, the *regions* object in Figure 5-6(a) consists of two faces rather than of a single face with a hole.



(a) (b)

Figure 5-6

An important observation is that for the *first* halfsegment of any cycle (with respect to the order of halfsegments) we can decide whether it belongs to an outer cycle or a hole. It is a left halfsegment and belongs to an outer cycle if and only if the attribute *InsideAbove* has been set, otherwise it belongs to a hole.

We adopt the following strategy: If for a given left halfsegment it is known that it belongs to an outer cycle, then we traverse the graph forming a *minimal* cycle containing that segment. This works as follows: For the given halfsegment, get the partner halfsegment (i.e. follow the edge). From the partner, go around that node to the *predecessor* in the counterclockwise order. Follow that edge, etc. As soon as the node of the initial halfsegment is reached again, a complete cycle has been found and its halfsegments can be marked as outer halfsegments.

This strategy works fine for the *regions* object in Figure 5-6(a) where it correctly determines the left face. However, in Figure 5-6(b) the cycle would also include the hole segments.

Vorausberechnungen für räumliche Objekte

Die Konstruktion eines *regions*-Objekts beinhaltet folgende Schritte:

- Füge Halbsegmente in einen AVL-Baum ein.
- Verknüpfe Halbsegmente in Sortierfolge (per Inorder-Durchlauf) und berechne die *bounding box* (kleinstes umgebendes Rechteck, für Vorfilterungen von Operationen).
- Benutze *InsideAttribute*-Algorithmus, um *InsideAbove*-Attribut zu jedem Segment zu berechnen (Plane Sweep).
- Klassifiziere Zyklen: Zyklusnummer zu jedem Segment, Zyklusnummer des umgebenden Zyklus zu jedem Lochsegment (Plane Sweeps).
- Durchlaufe Liste aller Segmente und verknüpfe Segmente zu *cycles* und *faces*; berechne auch dafür *bounding boxes*.
- Berechne ggf. (oder speichere beim ersten Aufruf) weitere Größen wie Fläche, Umfang, Durchmesser.

analog für *points*- und *lines*-Objekte