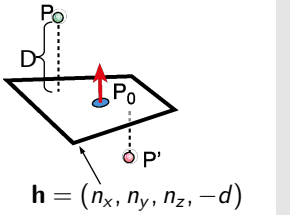
Clipping in Bezug auf das View Frustum

- Erinnerung: homogene Ebenengleichung

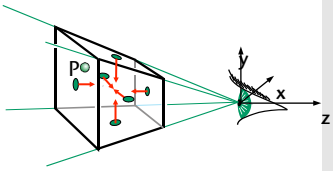
$$\mathbf{h} \cdot \mathbf{p} = 0$$

- Clipping eines Punktes:
  - Teste gegen alle 6 Ebenen
  - Annahme: Normalenvektoren sind nach innen gerichtet
  - Verwerfe Punkt P wenn für eine Ebene

$$\mathbf{h} \cdot \mathbf{p} < 0$$



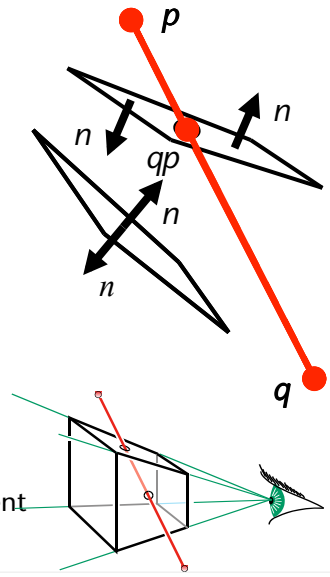
$\mathbf{h} = (n_x, n_y, n_z, -d)$



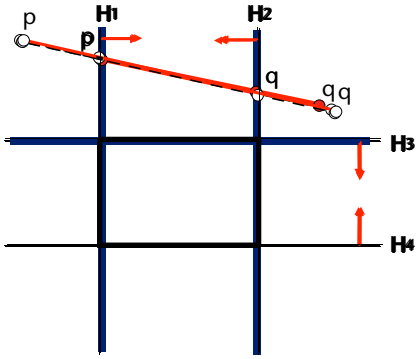
- Klappt für beliebige konvexe Polyeder

Clipping eines Linien-Segmentes

- 4 Fälle:
  1. Wenn $\mathbf{h} \cdot p > 0 \ \wedge \ \mathbf{h} \cdot q < 0$
     - Ersetze q
  2. Wenn $\mathbf{h} \cdot p < 0 \ \wedge \ \mathbf{h} \cdot q > 0$
     - Ersetze p
  3. Wenn $\mathbf{h} \cdot p > 0 \ \wedge \ \mathbf{h} \cdot q > 0$
     - "pass through"
  4. Wenn $\mathbf{h} \cdot p < 0 \ \wedge \ \mathbf{h} \cdot q < 0$
     - Komplett verwerfen ("reject")
- Für das ganze Frustum: alle Ebenen auf diese Weise durchlaufen
- Das Ergebnis ist ein einzelnes Segment (warum?)



Ist dieses Clipping effizient?



- Was ist das Problem?
  - Die Berechnung der Schnittpunkte und aller dazugehörigen interpolierten Werte sind — in diesem Fall — unnötig!
- Kann man dies früher erkennen?

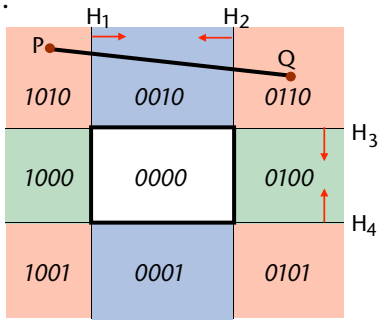
Erhöhung der Effizienz: Outcodes

- Berechne die "Sidedness" jedes Vertex bzgl. jeder Ebene
  - 0 = "richtige" Seite (Vorderseite); 1 = "falsche" Seite (Rückseite)
  - Ergibt pro Vertex einen 6-Bit langen *Outcode* (4 Bit im 2D)
- Bedingung:

$$\text{out}(P) \wedge \text{out}(Q) \neq 0 \rightarrow \text{"trivial reject"}$$

Bitweises UND!!

- Bsp.:

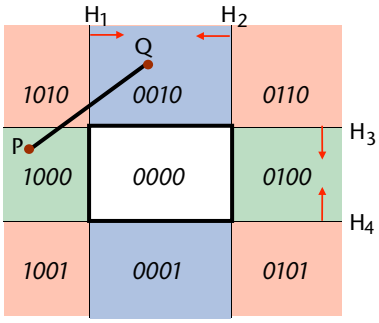


Outcode von P : 1010
Outcode von Q : 0110

AND : 0010

→ "trivial reject", da ≠ 0

- Weiteres Beispiel:



Outcode von P : 1000
Outcode von Q : 0010

AND : 0000

→ "potentially visible"

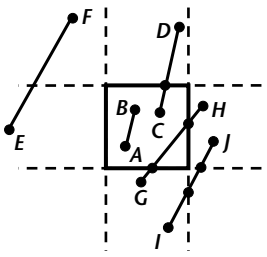
- In diesen Fällen macht der Test also keine Aussage!
- Dies ist der sog. Cohen-Sutherland-Algorithmus

- Weitere Aussage auf Basis der Outcodes:

$$\text{out}(P) \vee \text{out}(Q) = 0 \rightarrow \text{"trivial accept"}$$

Bitweises ODER!!

- Beispiele:



Linie	out(A)	out(B)	AND	OR
AB	0000	0000	0000	0000
CD	0000	1000	0000	1000
EF	0001	1001	0001	1001
GH	0100	0010	0000	0110
IJ	0100	0010	0000	0110

Der Cohen-Sutherland-Algorithmus

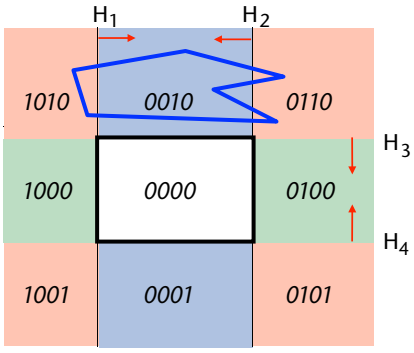
- Genereller Trick, der hier angewandt wird:
  - Erst einfache Tests durchführen, ob Clipping nötig ist
    - Die Tests liefern evtl. keine definitive Antwort, aber dafür sind sie sehr schnell
  - Dann erst im "nötigen" Fall die (teuren) mathematischen Operationen durchführen
- Der Code für die Outcodes:

```
unsigned int outcode( int x, int y )
{
  unsigned int c = 0;

  if ( y > ymax ) c = c | 8; // 1000
  if ( y < ymin ) c = c | 4; // 0100
  if ( x > xmax ) c = c | 2; // 0010
  if ( x < xmin ) c = c | 1; // 0001

  return c;
}
```


- Das Gute an dem Test: er funktioniert für beliebige Primitive in beliebigen Dimensionen mit beliebigen, konvexen Clip-Windows!
- Beispiel:



Outcode of p	: 1010
Outcode of q	: 1010
Outcode of r	: 0110
Outcode of s	: 0010
Outcode of t	: 0110
Outcode of u	: 0010
AND	: 0010

→ Clipped

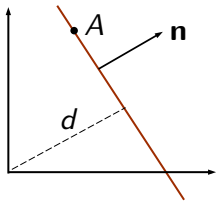
Clipping in homogenen Koordinaten

- Erinnerung:
  - Ein Punkt $P = (x, y, z) \in \mathbb{R}^3 \equiv$ Vektor $\hat{\mathbf{p}} = \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \in \mathbb{R}^4$ in homogenen Koord.
  - Alle Vielfache des homogenen Vektors entsprechen demselben Punkt:

$$s \cdot \hat{\mathbf{p}} \equiv P$$

- Sei eine Ebene gegeben durch $\mathbf{n}$, A
- Dann liegt P in der Ebene $\Leftrightarrow$

$$(P - A) \cdot \mathbf{n} = P \cdot \mathbf{n} - \underbrace{A \cdot \mathbf{n}}_{d} \equiv \hat{\mathbf{p}} \cdot \begin{pmatrix} \mathbf{n}_x \\ \mathbf{n}_y \\ \mathbf{n}_z \\ -d \end{pmatrix} = 0$$



- Der Vektor $\hat{\mathbf{n}} = (\mathbf{n}_x, \mathbf{n}_y, \mathbf{n}_z, -d)$ heißt auch homogene Darstellung der Ebene, oder homogene Normale

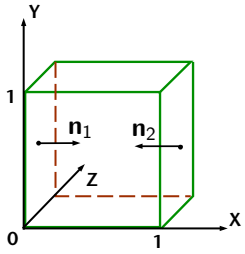
- Fazit: Punkt/Vektor im 3D ≡ Vektor im 4D
  Ebene im 3D ≡ Vektor im 4D
- Bemerkung: alle Vektoren $t \cdot \hat{\mathbf{n}}$ beschreiben die gleiche Ebene

$$\hat{\mathbf{p}} \cdot \hat{\mathbf{n}} = 0 = s\hat{\mathbf{p}} \cdot t\hat{\mathbf{n}}$$

Outcodes in homogenen Koordinaten

- Betrachte das Clip-Window (0,0,0) — (1,1,1)
  - Die homogene Darstellung der beiden Ebenen x=0 und x =1 ist

$$\mathbf{n}_1 = (1\ 0\ 0\ 0) \quad \mathbf{n}_2 = (-1\ 0\ 0\ 1)$$



- Definiere die *boundary distance (BD)*

$$d_i = \mathbf{p} \cdot \mathbf{n}_i\ , \quad \mathbf{p} = (x, y, z, w)$$

  - Berechnung ist trivial; z.B. $d_1 = x \qquad d_2 = w - x$
  - Erstelle Tabelle aller BDs →

BD	homog. Wert	Ebene
d_1	x	x=0
d_2	$w - x$	x=1
d_3	y	y=0
d_4	$w - y$	y=1
d_5	z	z=0
d_6	$w - z$	z=1

  - Outcodes sind auch trivial zu bestimmen:

$$_i(\mathbf{p}) = (d_i)$$

$$= \begin{cases} 1 & , P \text{ außerhalb Ebene } i \\ 0 & , P \text{ innerhalb Ebene } i \end{cases}$$

Trivial accept / reject

- Sei Linie PQ gegeben; in homogenen Koordinaten: **p**, **q**
- Wie bisher auch:
  - $\text{out}(\mathbf{p}) \wedge \text{out}(\mathbf{q}) \neq 000000 \Rightarrow$ trivial reject
  - $\text{out}(\mathbf{p}) \vee \text{out}(\mathbf{q}) = 000000 \Rightarrow$ trivial accept
- Ansonsten: es muß mindestens eine Bitposition *i* geben, wo $\text{out}_i(\mathbf{p}) = 0$ und $\text{out}_i(\mathbf{q})$ oder umgekehrt.
- Schneide X(t) mit dieser Clip-Window-Ebene:

$$X(t) = P_0 + t(Q - P) \ \leftrightarrow\ \hat{x}(t) = \mathbf{p} + t(\mathbf{q} - \mathbf{p})$$

$$\hat{x}(t) \cdot \mathbf{n}_i = [\mathbf{p} + t(\mathbf{q} - \mathbf{p})] \cdot \mathbf{n}_i = \mathbf{p}\mathbf{n}_i + t(\mathbf{q}\mathbf{n}_i - \mathbf{p}\mathbf{n}_i)$$

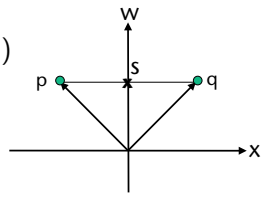
$$= d_i^p + t(d_i^q - d_i^p) \stackrel{!}{=} 0$$

$$t = \frac{d_i^p}{d_i^p - d_i^q}$$

- Bemerkung: $t \in \{0, 1\} \Leftrightarrow \text{sign}(d_i^p) \neq \text{sign}(d_i^q)$

Beispiel

- $P = (-1\ 0\ 0)$ $Q = (1\ 0\ 0)$ Ebene 1: $x = 0$
- $\mathbf{p} = (-1\ 0\ 0\ 1)$ $\mathbf{q} = (1\ 0\ 0\ 1)$ $\mathbf{n}_1 = (1\ 0\ 0\ 0)$
- Parameter des Schnittpunktes **s**:

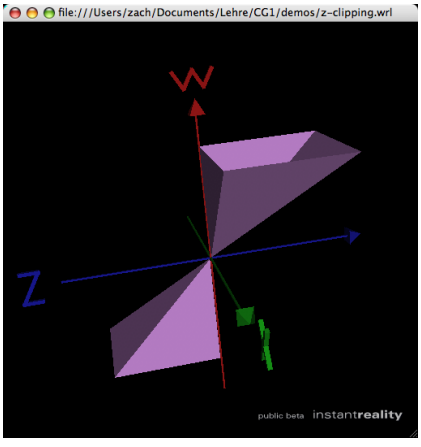


$$t = \frac{d_1^p}{d_1^p - d_1^q} = \frac{-1}{-1-(+1)} = \frac{1}{2}$$

- Schnittpunkt: $\mathbf{s} = \mathbf{p} + \frac{1}{2}(\mathbf{q} - \mathbf{p}) = (0\ 0\ 0\ 1)$
- Frage: was ist mit $\mathbf{p}' = (-2\ 0\ 0\ 2)$? (Ist derselbe Punkt P in 3D!)
  - Parameter $t' = \frac{2}{3}$
  - Ist das ein anderer Punkt?
  - Nein, denn Schnittpunkt

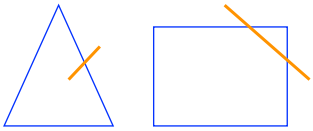
$$\mathbf{s}' = \mathbf{p}' + \frac{2}{3}(\mathbf{q} - \mathbf{p}') = \begin{pmatrix} -2 \\ 0 \\ 0 \\ 2 \end{pmatrix} + \frac{2}{3}\begin{pmatrix} 3 \\ 0 \\ 0 \\ -1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1\frac{1}{3} \end{pmatrix} \equiv \mathbf{s}$$

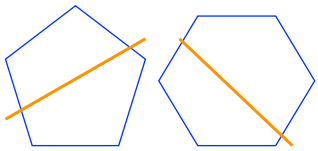
- Visualisierung der Clipping-Region im 4D:

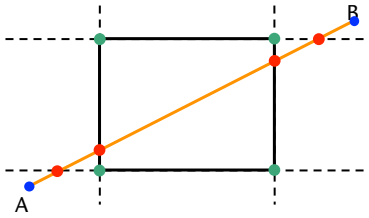


Der Cyrus-Beck-Algorithmus [1978]

- Clippen von Linien an beliebigen, konvexen Clip-Windows
- Verwendet die Parameterdarstellung der Linien
- Im Folgenden wird wieder das Rechteck als Beispiel verwendet; der Algorithmus ist aber für beliebige (konvexe) Clip-Windows anwendbar
- Bei einem n-Eck kann es bis zu n Schnittpunkte geben
- Nur 2 davon sind echte Schnittpunkte mit dem Clip-Window

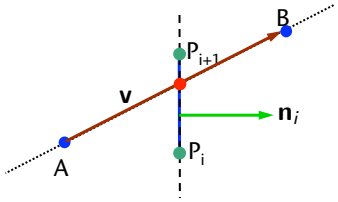






- Berechne für jeden Clipping-Rand *i* das zugehörige *t*:

$$t = -\frac{(A - P_i)\cdot \mathbf{n}_i}{\mathbf{v}\cdot \mathbf{n}_i}$$



- Die Werte $t<0$ und $t>1$ werden ignoriert
- Jeweils genau ein *t* markiert den Eintrittspunkt und den Austrittspunkt der Linie

Entering and Leaving t's

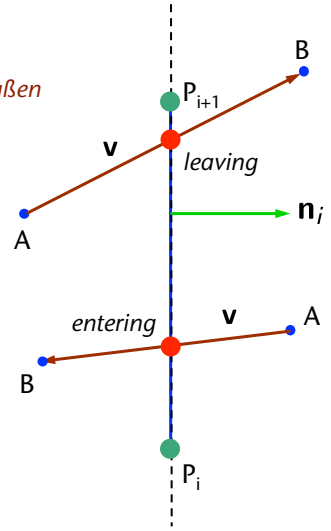
- Idee: klassifiziere alle t's
  - Konvention: Normalen zeigen nach *außen*
  - "Leaving", falls:

$$\mathbf{n}_i \cdot \mathbf{v} > 0 \Rightarrow t_i^l$$

  - "Entering", falls:

$$\mathbf{n}_i \cdot \mathbf{v} < 0 \Rightarrow t_i^e$$

  - Sonst: Sonderfall, der anderweitig abgefangen wird



Der Algorithmus

- Berechne das Maximum und Minimum

$$t_e = \max\{t_i^e, 0\} \qquad t_l = \min\{t_i^l, 1\}$$

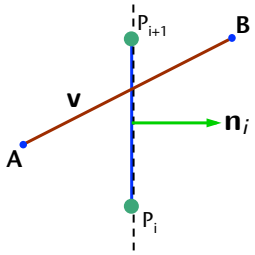
- Falls $t_e > t_l$ → Linie ist komplett außerhalb des Clip-Windows
- Sonst: t_e und t_l definieren die Enden der geclippten Linie

- Alternative Betrachtungsweise:
  - Starte mit Intervall [0,1]
  - Schneide Linie der Reihe nach gegen jeden Clip-Rand
  - Falls "entering" → schneide aktuelles Intervall unten ab (falls überhaupt)
  - Falls "leaving" → schneide aktuelles Intervall oben ab
  - Stop, falls Intervall leer wird

Sonderfälle

- Wann kann der Nenner Null werden?

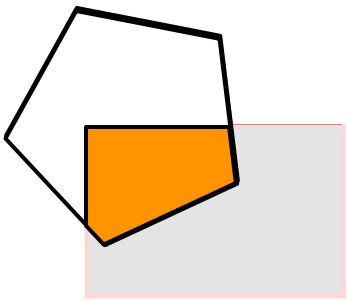
$$t = -\frac{(A - P_i)\cdot \mathbf{n}_i}{\mathbf{v}\cdot \mathbf{n}_i}$$

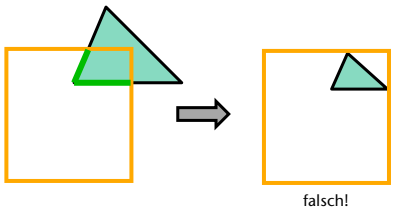


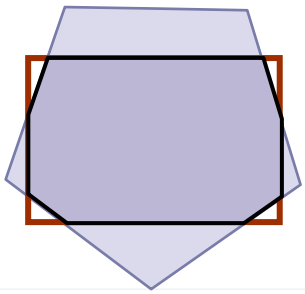
- $\mathbf{v} = 0$:
  - Start- und Endpunkt der Linie sind identisch (muß vorher abgefangen werden)
- $\mathbf{n}_i = 0$:
  - Nur, falls 2 Punkte des Clip-Windows identisch sind, vorher abfangen
- $\mathbf{n}_i \cdot \mathbf{v} = 0$:
  - Zu zeichnende Linie ist parallel zu einer Kante des Clip-Objekts → kein t ausrechnen, nächsten Clip-Rand betrachten

Polygon Clipping in 2D

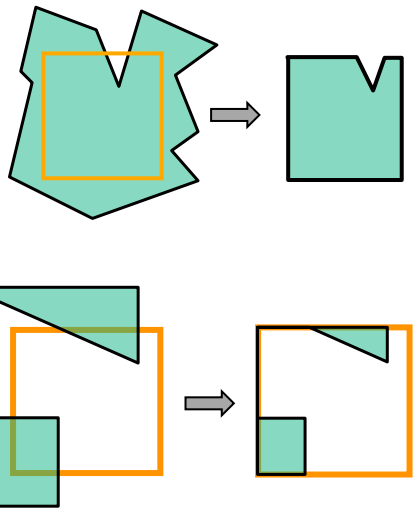
- Clipping ist symmetrisch
- Polygon Clipping ist komplex
  - Sogar bei konvex Polygonen
- Polygon Clipping ≠ Clipping der Liniensegmente!



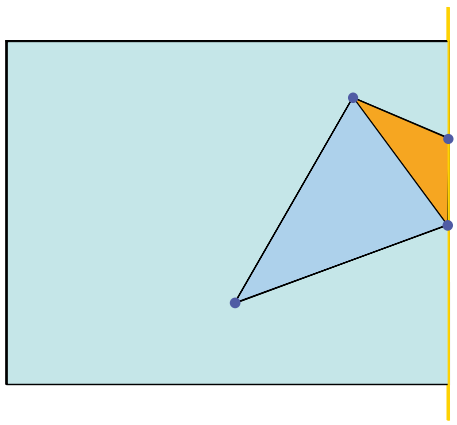


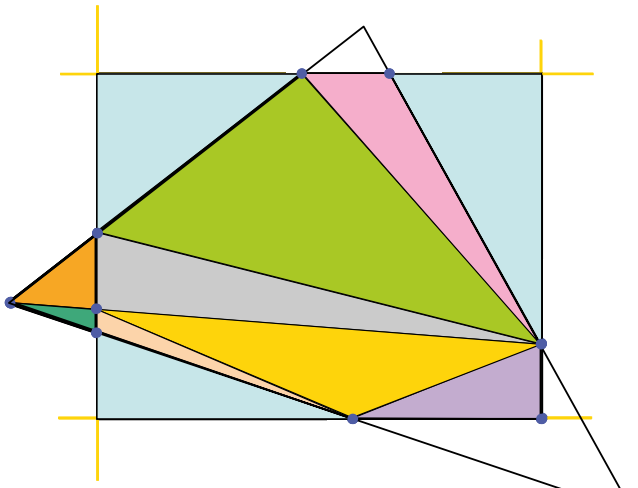


- Polygon-Clipping kann, insbesondere bei nicht-konvexen Polygonen, unangenehm werden
- Es können sogar mehrere Polygone entstehen



Naiver Clipping-Algorithmus

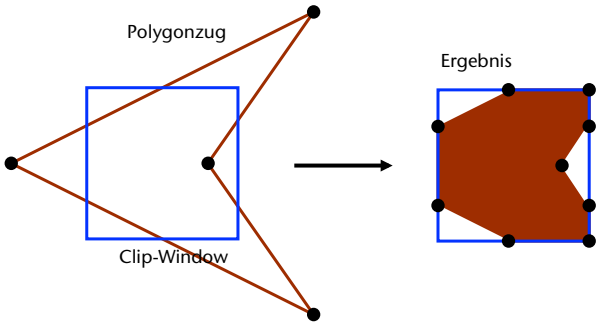




Fazit: das naïve Polygon-Clipping ist schon
für Dreiecke zu ineffizient (erzeugt viel zu viel Output)

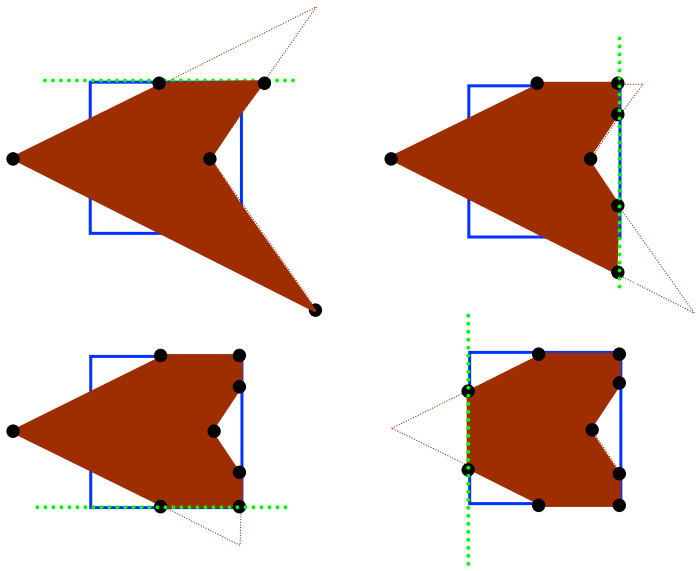
Sutherland-Hodgman

- Clipping eines Polygonzugs gegen ein konvexes Clip-Polygon (z.B. Viewport); der Polygonzug darf konkav sein



Vorgehen

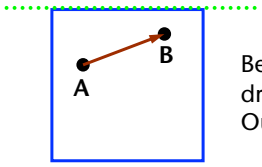
- Eingabe =
  - Liste der Eckpunkte in der richtigen Reihenfolge (gegen den Uhrzeigersinn)
  - Menge von Clip-Kanten, die ein konvexes Clip-Window definieren
- Betrachte eine (beliebige) Clip-Kante:
  - Nach dem Schnitt mit dieser Clip-Kante wird eine neue Liste von Eckpunkten erzeugt
  - Dieses Ergebnis ist wieder ein geschlossener Polygonzug
  - Alle Punkte des neuen Polygons befinden sich auf der "Innenseite" (der "richtigen" Seite) dieser Clip-Kante (Schleifeninvariante)
- Das wird mit allen Clip-Kanten wiederholt
  - (Im Prinzip ist die Reihenfolge der Clip-Kanten egal)



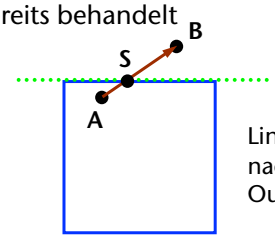
Der Polygonzug wird der Reihe nach an den Clip-Kanten geschnitten

4 Fälle

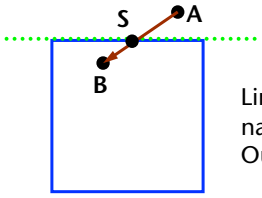
- Annahme: der Punkt A wurde bereits behandelt



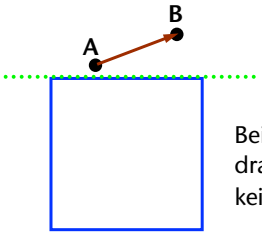
Beide Punkte drinnen: → Output B



Linie „zeigt“ nach außen: → Output S

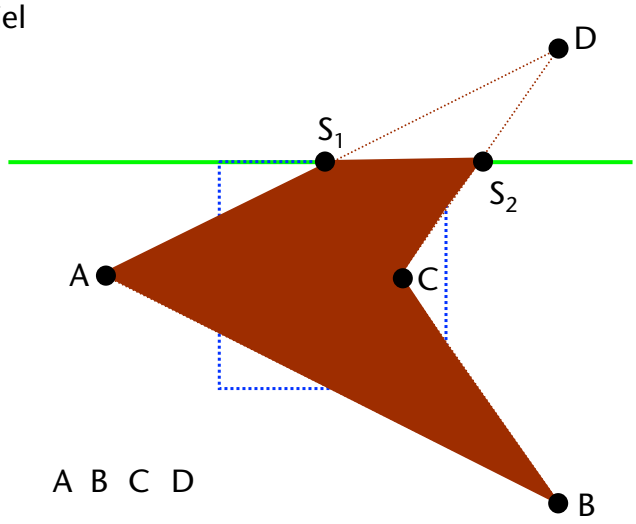


Linie „zeigt“ nach innen: → Output S, B



Beide Punkte draußen: → kein Output

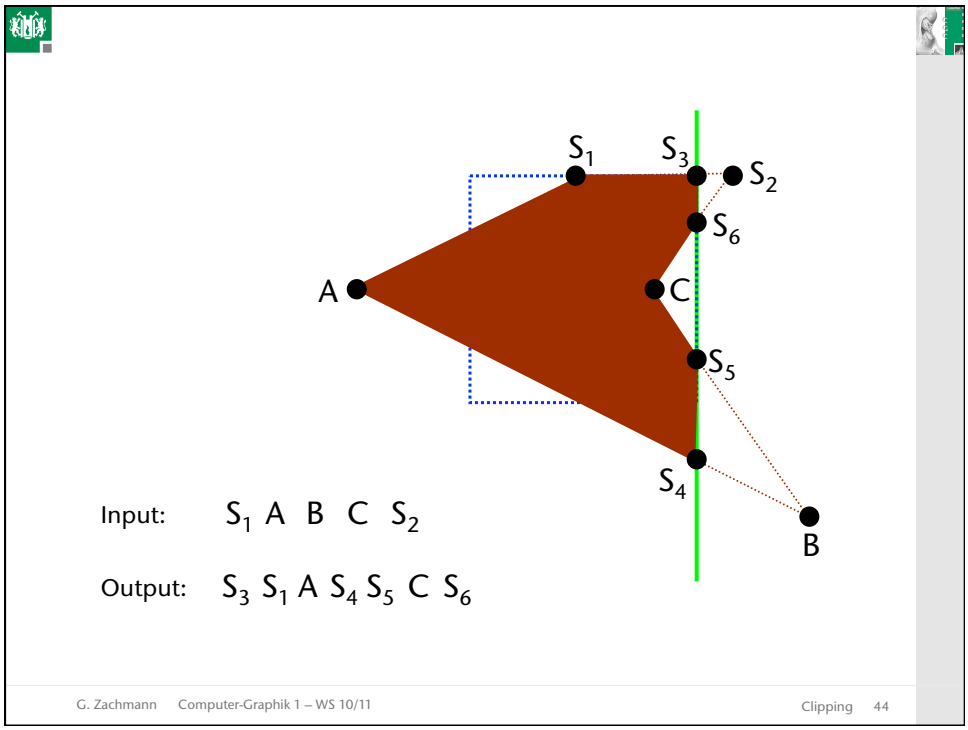
Beispiel

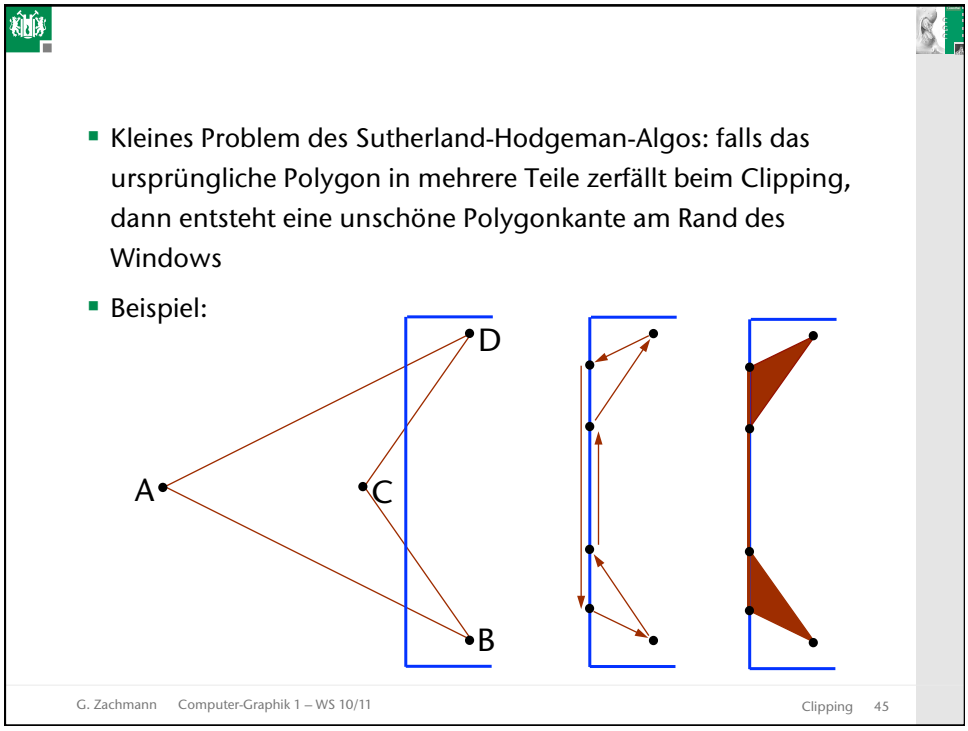


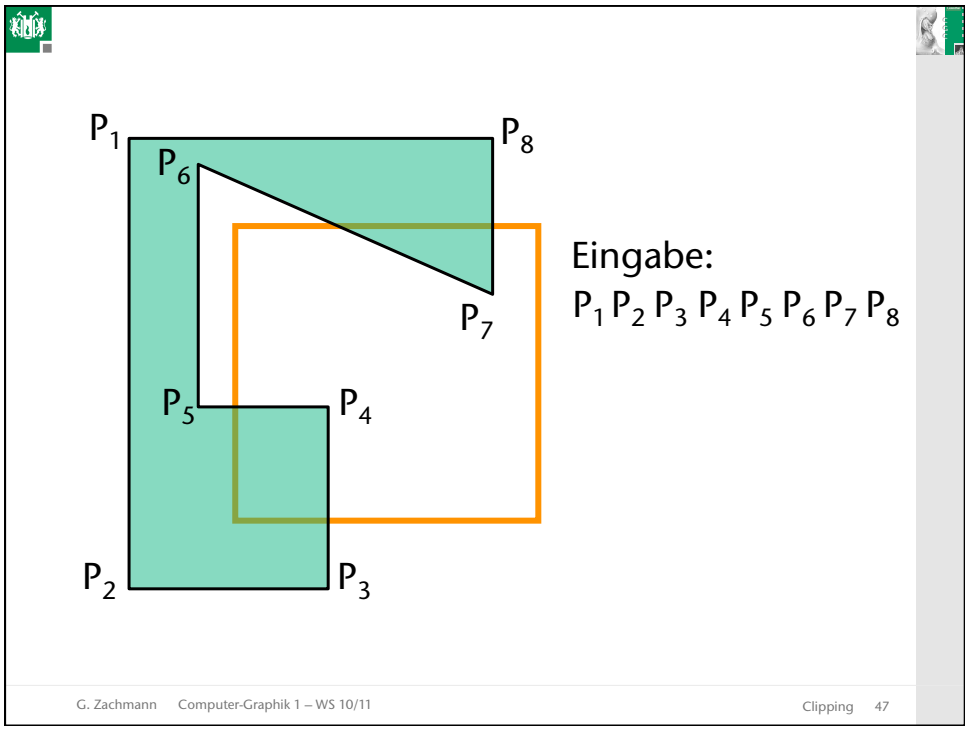
Input: A B C D

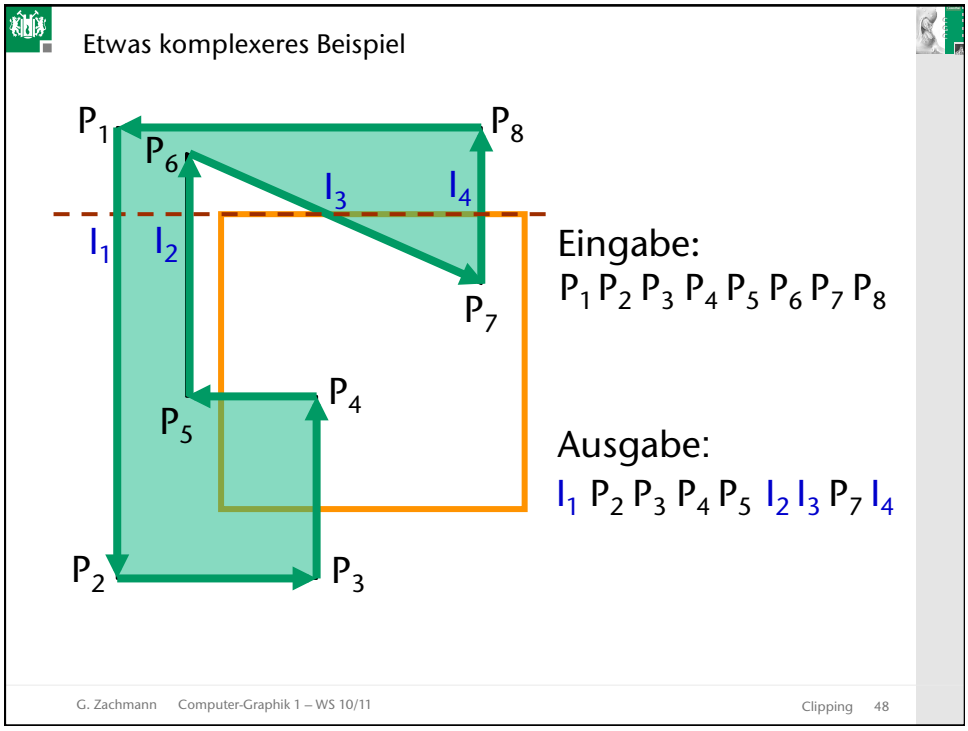
Output: S_1 A B C S_2

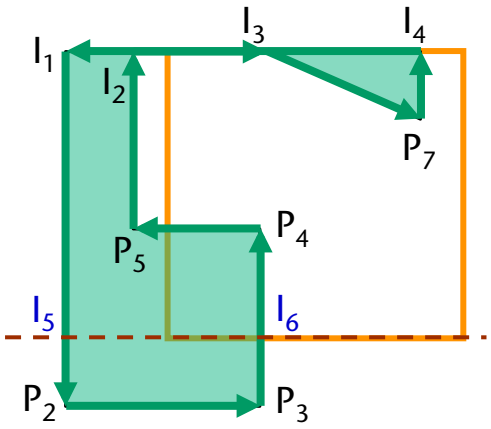
Man beginnt mit der Kante [letzter Punkt – erster Punkt], hier also D-A.
(D wird am Ende der Schleife betrachtet)


S1
S3
S2
S6
A
C
S5
S4
B
Input: S1 A B C S2
Output: S3 S1 A S4 S5 C S6
G. Zachmann Computer-Graphik 1 – WS 10/11
Clipping 44


Kleines Problem des Sutherland-Hodgeman-Algos: falls das ursprüngliche Polygon in mehrere Teile zerfällt beim Clipping, dann entsteht eine unschöne Polygonkante am Rand des Windows
Beispiel:
D
A
C
B
G. Zachmann Computer-Graphik 1 – WS 10/11
Clipping 45


P1
P8
P6
P7
P5
P4
P2
P3
Eingabe:
P1 P2 P3 P4 P5 P6 P7 P8
G. Zachmann Computer-Graphik 1 – WS 10/11 Clipping 47


Etwas komplexeres Beispiel
P1
P8
P6
I3
I4
I1
I2
P7
P4
P5
P2
P3
Eingabe:
P1 P2 P3 P4 P5 P6 P7 P8
Ausgabe:
I1 P2 P3 P4 P5 I2 I3 P7 I4
G. Zachmann Computer-Graphik 1 – WS 10/11 Clipping 48

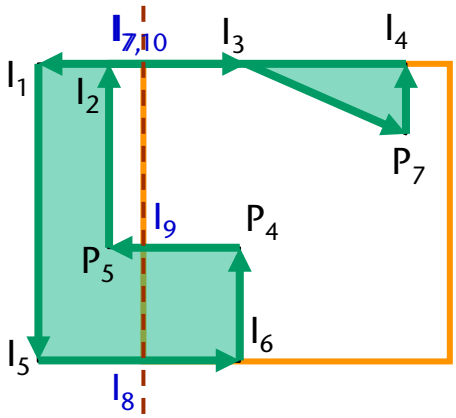


Eingabe:

$I_1\ P_2\ P_3\ P_4\ P_5\ I_2\ I_3\ P_7\ I_4$

Ausgabe:

$I_1\ I_5\ I_6\ P_4\ P_5\ I_2\ I_3\ P_7\ I_4$

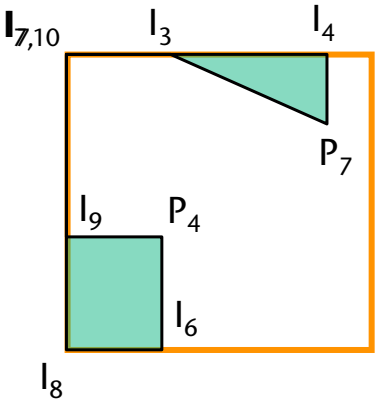


Eingabe:

$I_1\ I_5\ I_6\ P_4\ P_5\ I_2\ I_3\ P_7\ I_4$

Ausgabe:

$I_7\ I_8\ I_6\ P_4\ I_9\ I_{10}\ I_3\ P_7\ I_4$

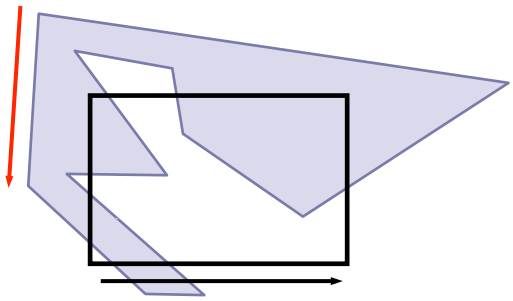


Ausgabe:

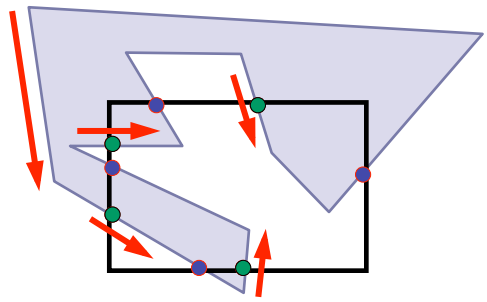
$I_7\ I_8\ I_6\ P_4\ I_9\ I_{10}\ I_3\ P_7\ I_4$

Der Algoithmus von Weiler-Atherton

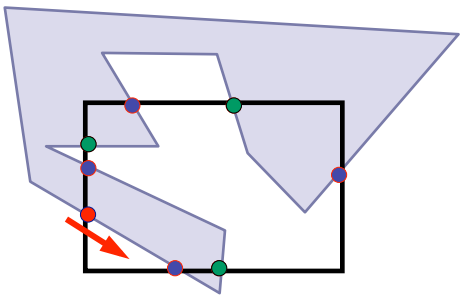
- Strategie: "Wandere" auf dem Polygonzug oder dem Window-Rand
- Konvention (wie immer): Polygone sind CCW orientiert



- Berechne alle Schnittpunkte
- Markiere die Punkte, an denen das Polygon in das Clipping-Window eintritt (hier grün)

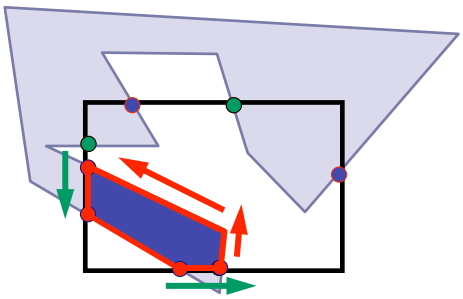


- Solange noch ein unbearbeiteter Eintrittsschnittpunkt vorhanden ist wird das Polygon weiter umlaufen
- Die Ausgabe ist (wie bei Sutherland-Hodgman) ein oder mehrere Listen von Punkten (Eckpunkte des Polygons, Schnittpunkte, und/oder Eckpunkte des Windows)
  - Unterschied: die Eingabe wird nur einmal abgearbeitet

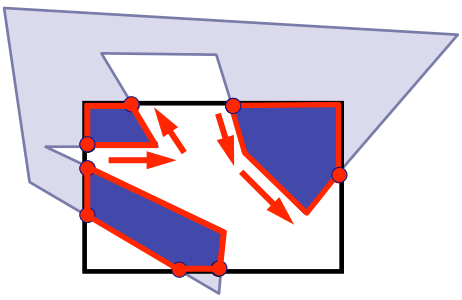


Umlauf-Regeln

- Trifft man beim Umlauf auf einen Schnittpunkt, dann:
  - Füge Schnittpunkt zur Ausgabe hinzu
  - Falls Schnittpunkt = "Out-to-in": folge dem Polygonzug (ccw)
  - Falls Schnittpunkt = "In-to-out": folge dem Window-Rand (ccw)



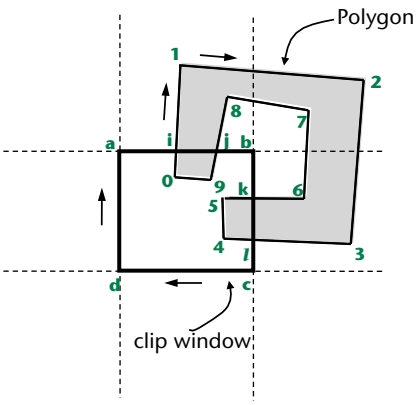
- Solange noch ein unbearbeiteter Eintrittsschnittpunkt vorhanden ist wird das Polygon weiter umlaufen

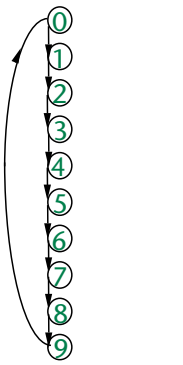


- Der Weiler-Atherton-Algorithmus erzeugt echt separate Polygonzüge für jedes sichtbare Fragment

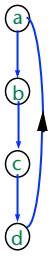
Implementierung des Algorithmus

- Eingabe (hier ist die Konvention *clockwise ordering*):



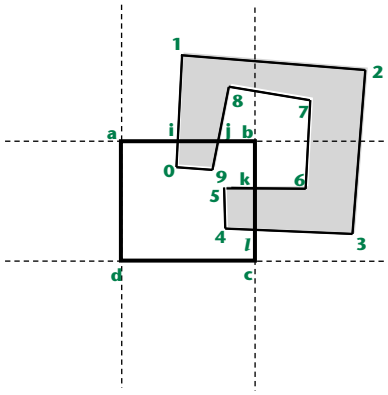


Zyklische Liste der Vertices des Polygons

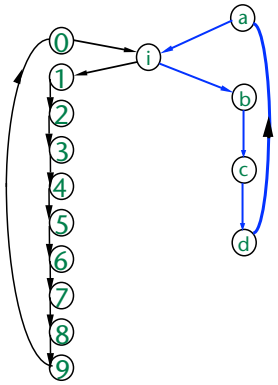


Zyklische Liste der Vertices des Clip-Windows

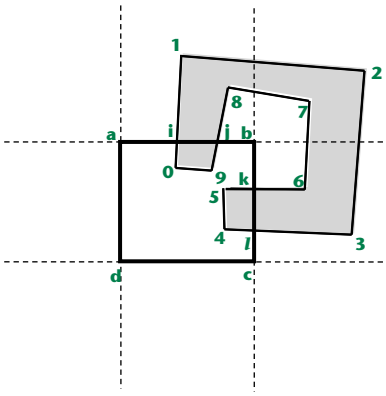
- Finde die Schnittpunkte und füge sie in beide Listen ein



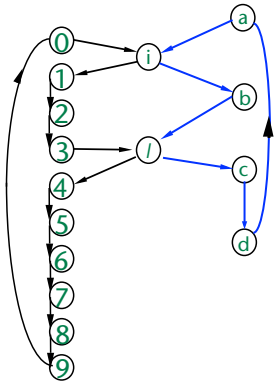
Add Vertex i:



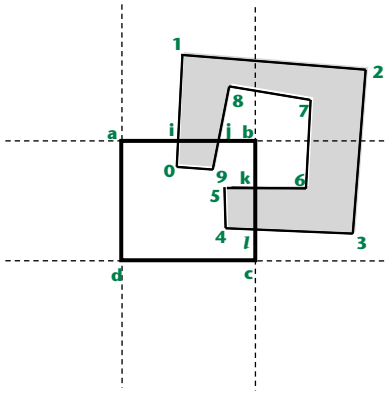
- Finde die Schnittpunkte und füge sie in beide Listen ein



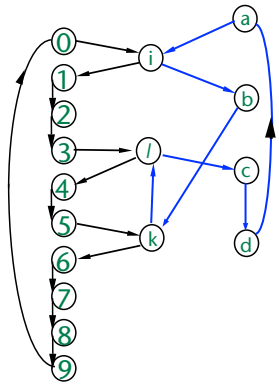
Add Vertex *l*:



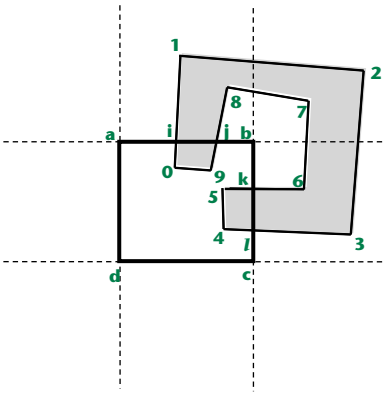
- Finde die Schnittpunkte und füge sie in beide Listen ein

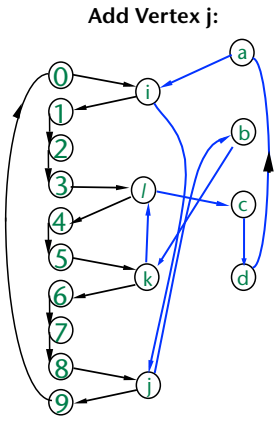


Add Vertex k:

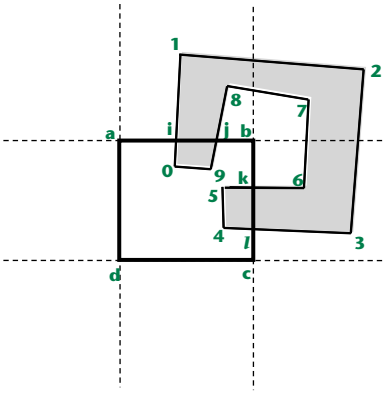


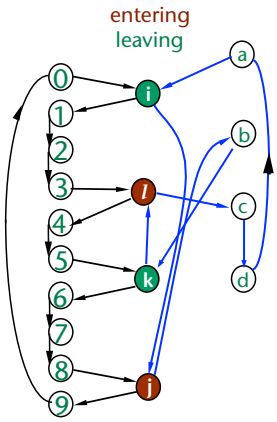
- Finde die Schnittpunkte und füge sie in beide Listen ein



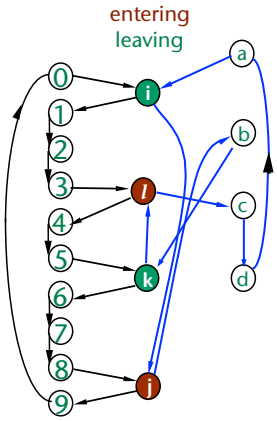


- Klassifiziere jeden Schnittpunkt als „*entering*“ oder „*leaving*“



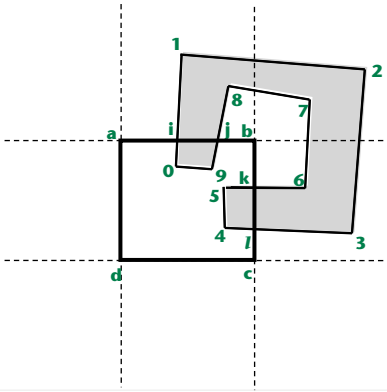


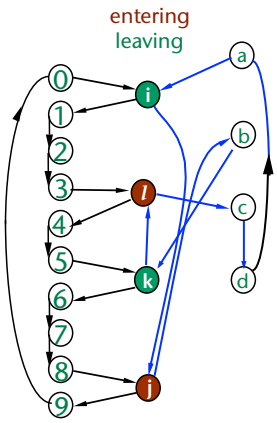
- Erzeuge das geclippte Polygon:
  - Starte bei einem "entering"-Vertex
  - Falls man auf einen "leaving"-Vertex trifft, dann wechsle auf die Liste des Clip-Polygons (blaue Zeiger)
  - Falls man auf einen "entering"-Vertex trifft, dann wechsle auf die Liste des Polygons (schwarze Zeiger)
  - Eine Polygonzug ist beendet, wenn der Startpunkt wieder erreicht ist
  - Wiederhole, solange noch unbesuchte "entering"-Vertices vorhanden sind



Beispiel

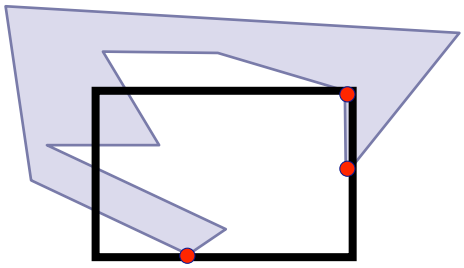
- Polygonzug 1 = *l*, 4, 5, k
- Polygonzug 2 = j, 9, 0, *l*





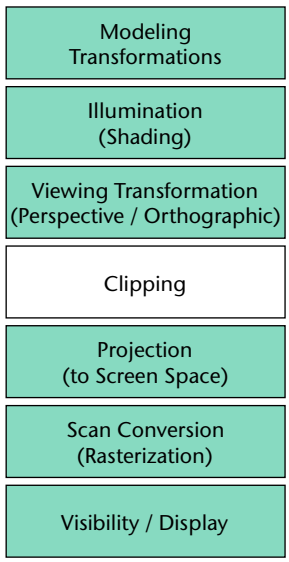
Robustheit, Präzision, Entartungen

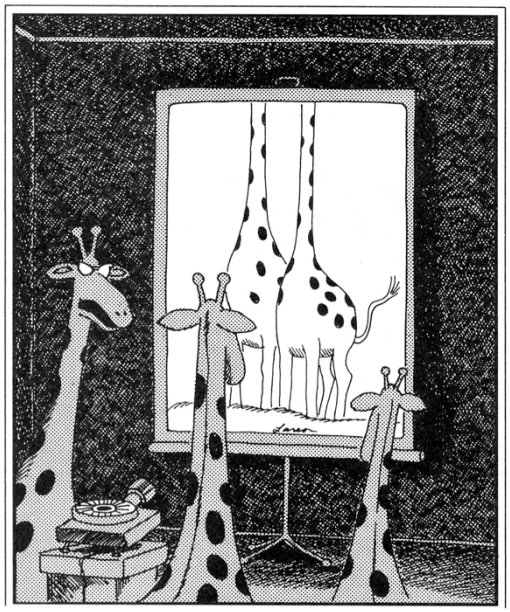
- Die üblichen (leidigen) Fragen:
  - Was passiert wenn ein Vertex (beinahe) auf dem Rand des anderen Polygonzuges liegt?



Absch(l)ießende Bemerkungen

- Ältere Hardware machte volles Clipping
- Moderne Hardware vermeidet Clipping weitgehend:
  - Nur bzgl. der Ebene $z=z_0$
- Im Allgemeinen ist es nützlich, Clipping zu kennen, da es viele ähnliche geometrische Probleme und Algorithmen gibt, z.B.:
  - Zur Bestimmung, welche Objekte innerhalb eines "Picking-Frustums" liegen
  - Schnittpunkte zwischen Objekten
  - Berechnung analytischer Schatten





"Oh, lovely — just the hundredth time you've managed to cut everyone's head off."