



3. Rendering

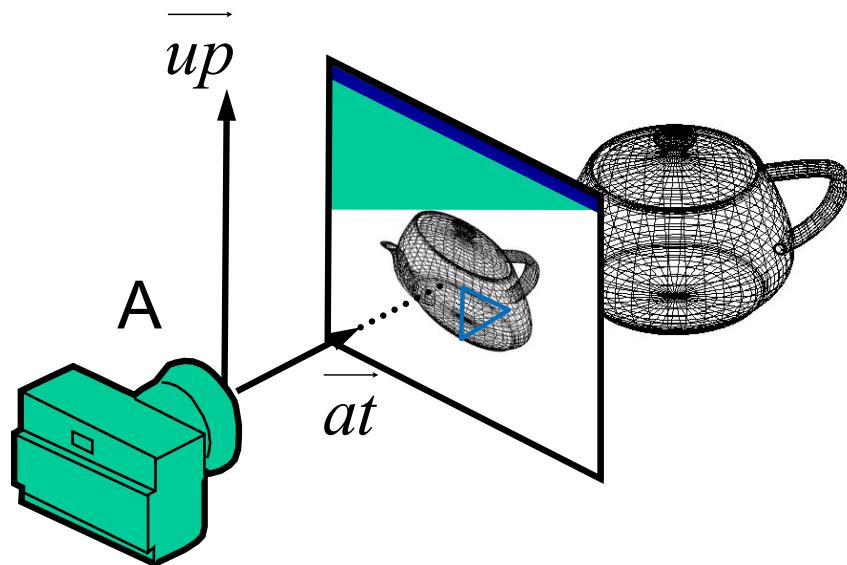
3.4 Culling und Clipping

3. Rendering

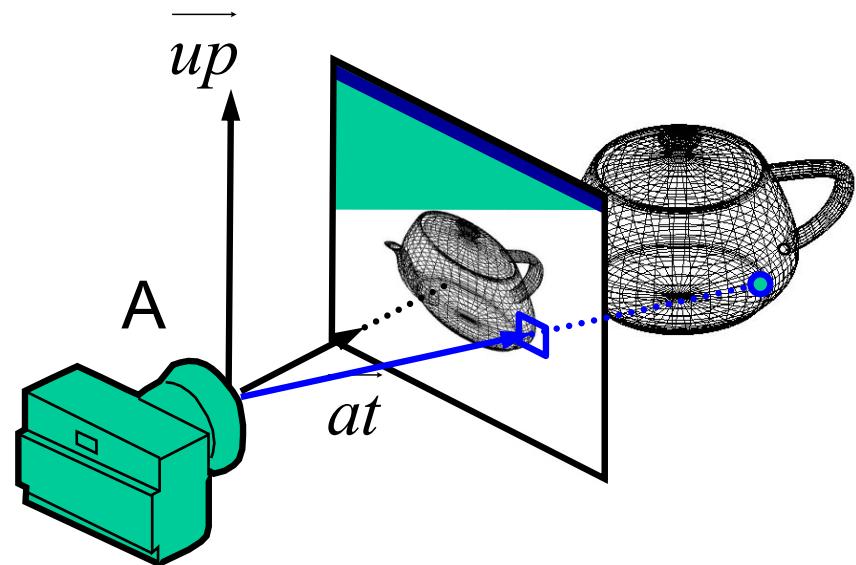
3.4 Clipping



Projektion/Rasterisierung



Ray Tracing



Die Polygone werden der Reihe nach rasterisiert

Das Bild wird durch Sehstrahlen abgetastet



- ***Projektion/Rasterisierung:*** graphische Objekte können ganz oder teilweise ausserhalb des viewing volumes liegen
- Ziel: schnelles rendern!
- Objekte ausserhalb des viewing volumes müssen erkannt und eliminiert werden
- Objekte können nur teilweise innerhalb des view volumes liegen, in diesem Fall müssen die Objekte "abgeschnitten" werden



- **Clipping:** ist der Prozess festzustellen, welche Primitive sich innerhalb des view volumes befinden
- Primitive die vollständig innerhalb des view volumes sind werden *accepted*. Primitive vollständig ausserhalb des view volumes werden eliminiert oder *rejected*.
- Ein Primitiv welches nur teilweise im view volume ist muss *geclipped* werden. Das heisst, Teile ausserhalb des view volume werden "abgeschnitten".



- **Clipping** – Anforderungen:
 - Geometrie-Typ kann sich ändern
 - Abschneiden eines vertex kann aus Dreieck ein Viereck machen
 - Diese müssen möglicherweise wieder trianguliert werden
 - Polygone müssen nach Clippen geschlossen sein



- **Culling:**

- **Schnelles und frühzeitiges Ausschliessen von Objekten, die sicher nicht sichtbar sind**
- Konservativ: Objekte die beim cullen rejected werden (geculled) sind sicher vollständig ausserhalb des view volumes
- Es kann passieren, dass Objekte vollständig ausserhalb des view volumes den Test passieren
- “Quick and dirty” test



- ***Wo findet Clipping und culling statt:***

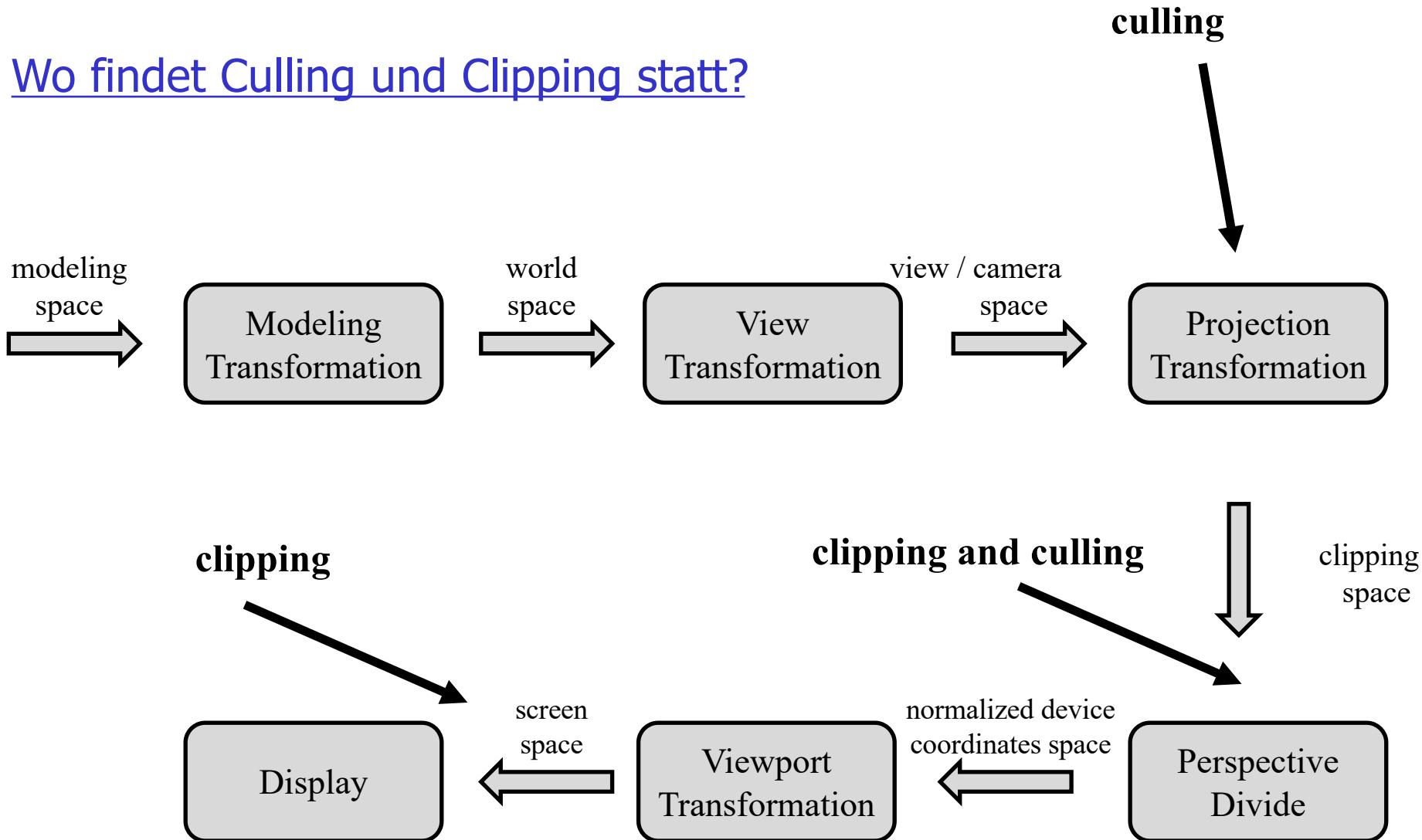
- Ziel 1: reject Objekte so früh wie möglich, damit nicht sichtbare Teile nicht transformiert werden müssen
- Ziel 2: rejection sollte effizient durchgeführt werden
- Ziele 1 und 2 widersprechen sich teilweise, da clipping in normierten Koordinatensystemen schneller ist
- *Culling*: möglichst früh in der viewing pipeline, damit nicht sichtbare Teile nicht transformiert werden müssen
- *Clipping*: später in der pipeline, da clipping algorithmen dann effektiver

3. Rendering

3.4 Clipping



Wo findet Culling und Clipping statt?

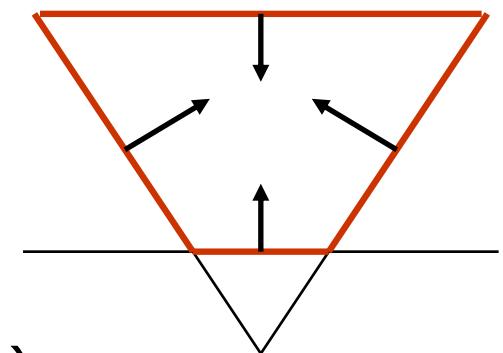


3. Rendering

3.4 Clipping



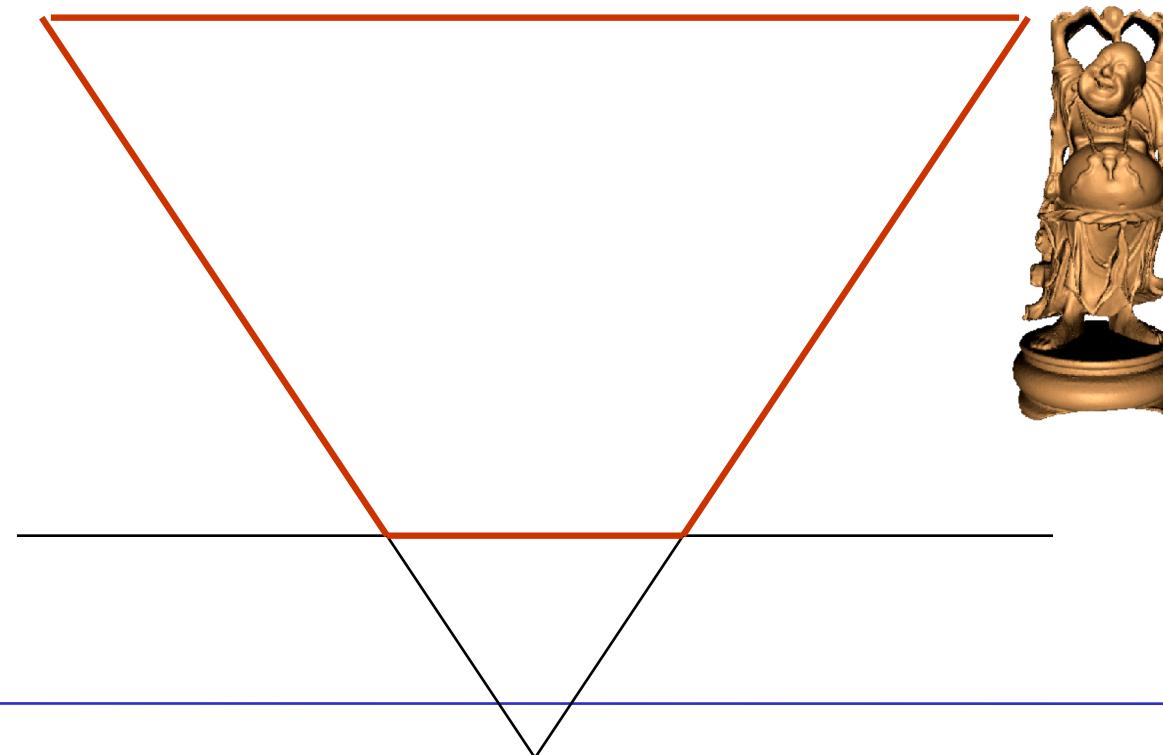
- **Culling**
- Reject Objekte möglichst zeitig
 - Ausgeführt durch Anwendungsprogramm (oder application framework)
 - Pre-processing
- Frustum wird gebildet durch 6 Seiten (Halbräume)
 - Teste Schnitt mit Frustum-Seiten
 - Jedes Polygon welches das Frustum nicht schneidet wird geculled



Was ist schlecht mit diesem einfachen Algorithmus?



- **Ineffizientes per-Polygon Processing**
- Was ist wenn eine Million Polygone komplett ausserhalb des View Frustums sind?
 - Wir wollen nicht jedes einzelne testen!

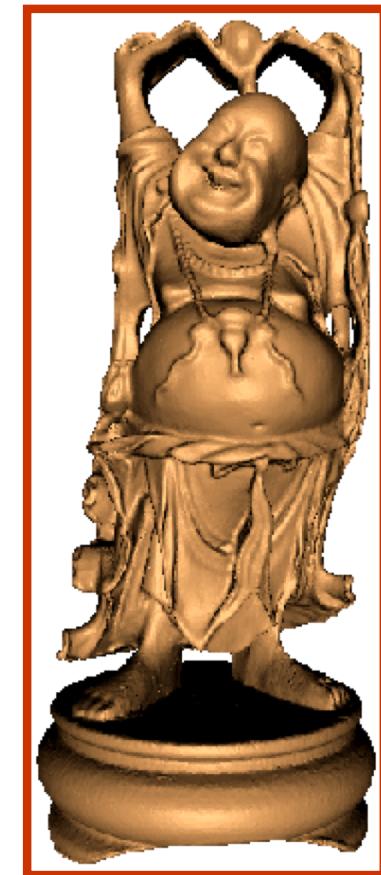


3. Rendering

3.4 Clipping



- **Culling mit Bounding Volumes**
- Wir bestimmen ein einfaches umschliessendes Objekt (Hüllvolumen):
 - **bounding sphere**
 - Kompakte Repräsentation
 - Oft kein enges Umschliessen des Objektes
 - **bounding box**
 - Ausgerichtet an Koordinatenachsen oder mit dem Objekt
 - **convex polyhedron**
 - Erlaubt dichtestes Umschliessen
 - Grösster Aufwand in der Verarbeitung
- Zuerst Test mit einfachem umschliessendens Objekt
 - Ausserhalb der Frustrums: reject aller Primitive des Objektes
 - Sonst: individuelle Behandlung aller Primitive



3. Rendering

3.4 Clipping



- **Hierarchical Bounding Volumes**
- Dies kann noch verbessert werden durch Hierarchien von Hüllvolumen
- Beginne Test an der Wurzel:
 - if outside, reject aller Objekte
 - sonst, recursiver Test der Kind-Knoten
- Wie bestimmt man eine optimale Hierarchie? Verschiedene Heuristiken möglich, nicht hier behandelt.



3. Rendering

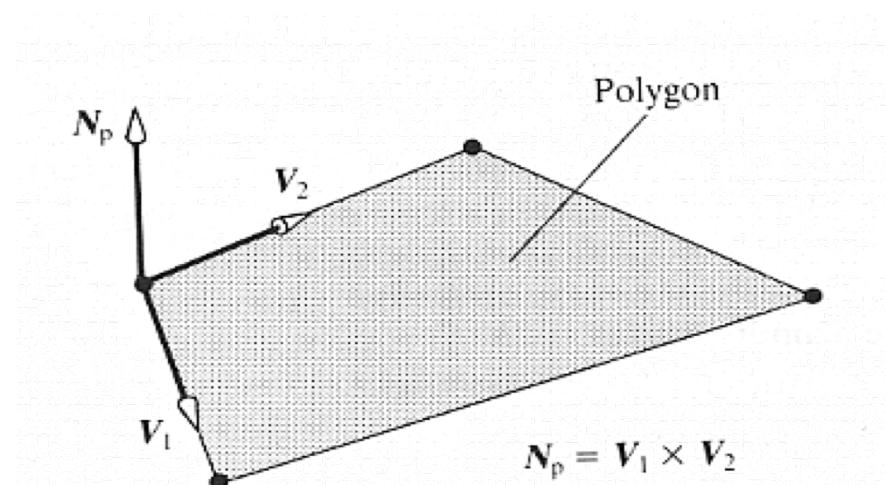
3.4 Clipping



- **Backface Culling**
- Auch Polygone innerhalb der Frustums können geculled werden wenn man annimmt dass die Objekte geschlossen sind
- Wir betrachten die Polygon-Normale

$$\mathbf{N}_P = \mathbf{V}_1 \times \mathbf{V}_2$$

- Orientierte Polygon-Kanten $\mathbf{V}_1, \mathbf{V}_2$
- Wenn \mathbf{N}_P in Richtung des Auges zeigt, können wir das Polygon prinzipiell sehen
- Wenn nicht: reject!



3. Rendering

3.4 Clipping



- **Backface Culling**

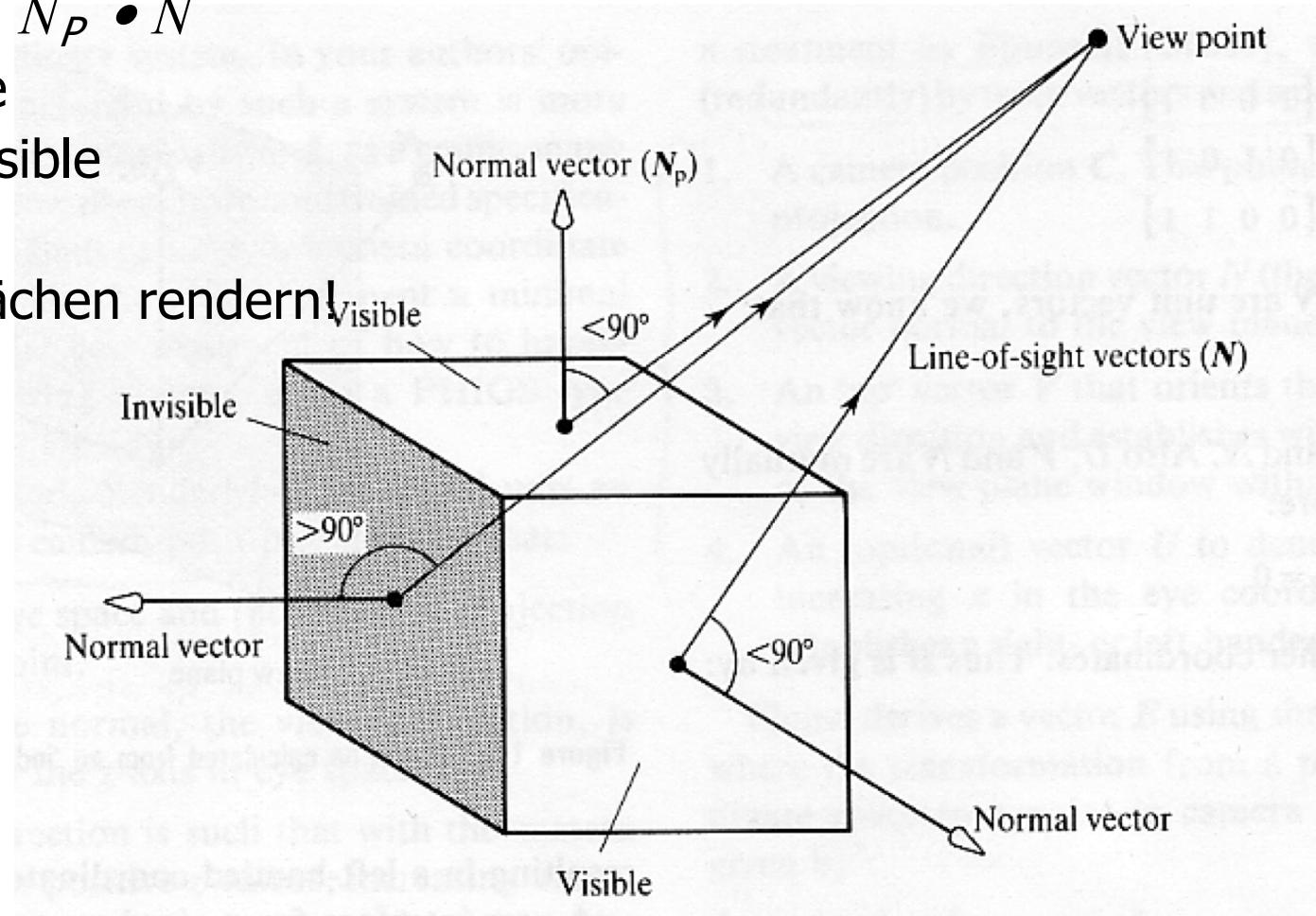
- Line-of-sight vector N

- $N_P \bullet N$

> 0 : surface visible

< 0 : surface not visible

⇒ nur sichtbare Flächen rendern!

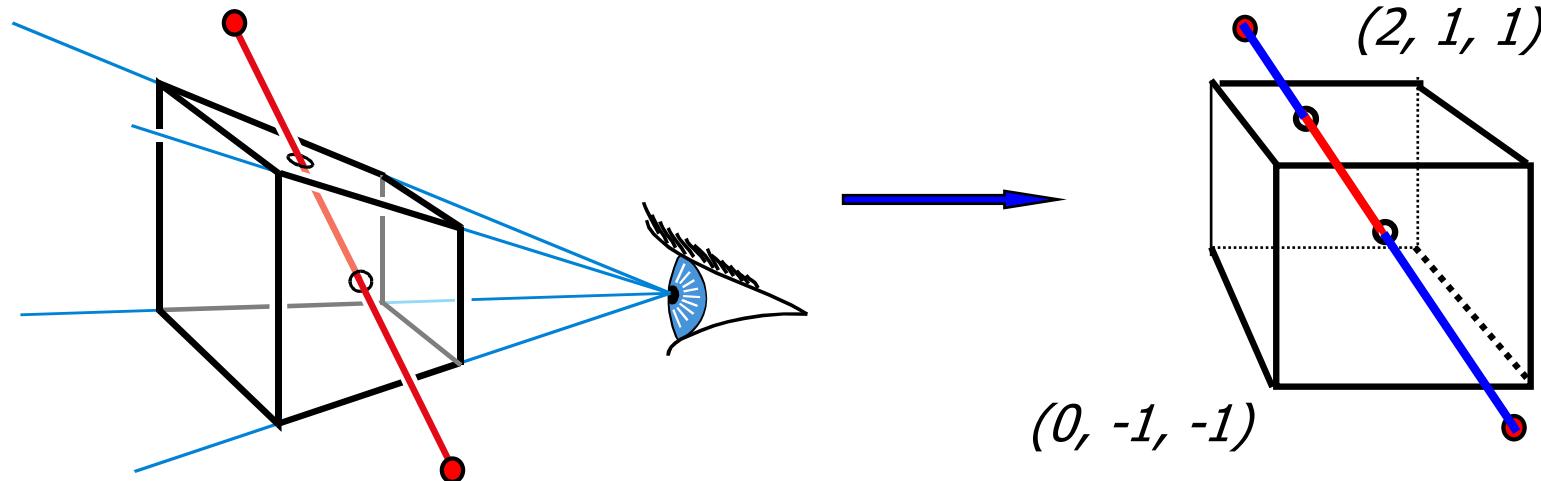


3. Rendering

3.4 Clipping



- **Clipping im Clipping space**
- Clipping findet nach der View Transformation statt, da im kanonischen view volume das clipping effektiver ist!

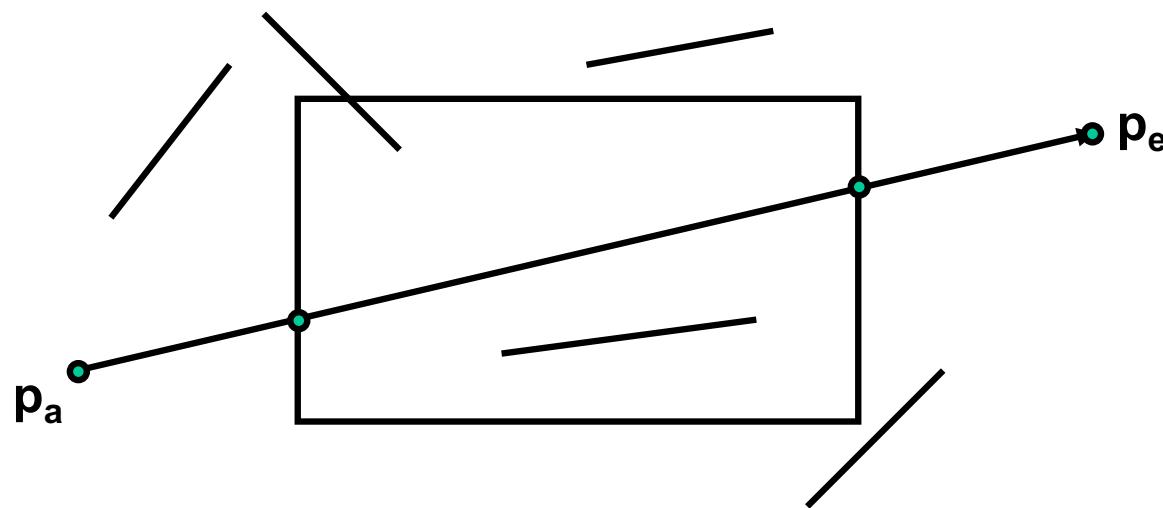


3. Rendering

3.4 Clipping



- **2D Line Clipping**
- 4 Fälle
 - Beide Endpunkte innerhalb des windows
 - Ein Punkt drin, ein Punkt draussen
 - Beide Endpunkte ausserhalb, Linie schneidet Window nicht
 - Beide Endpunkte ausserhalb, Linie schneidet Window



3. Rendering

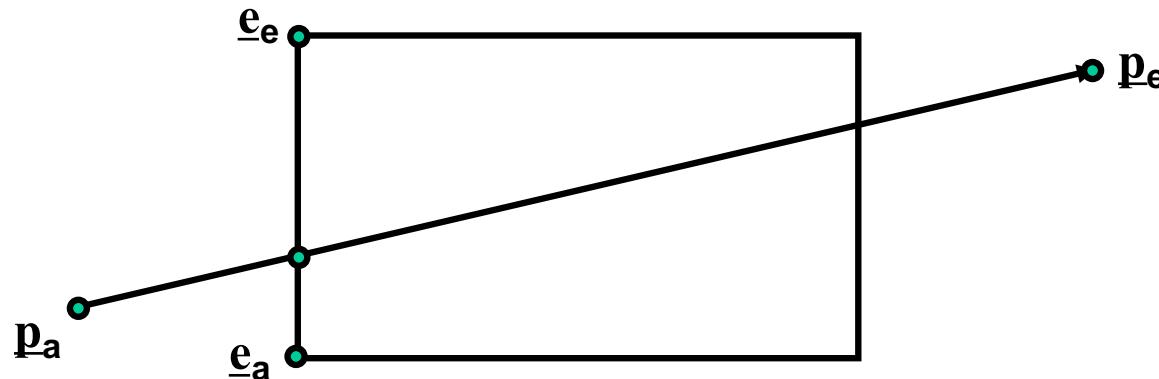
3.4 Clipping



- **Naives 2D Line Clipping**
- Brute-Force
 - wenn beide Punkte \underline{p}_a und \underline{p}_e innerhalb des windows,
 - Akzeptiere komplette Linie
 - Sonst: clipping der Linie an jeder Kante des windows

$$(1 - t_{line}) \underline{p}_a + t_{line} \underline{p}_e = (1 - t_{edge}) \underline{e}_a + t_{edge} \underline{e}_e$$

- Schnittpunkt falls $0 \leq t_{line}, t_{edge} \leq 1$
- Ersetze Endpunkt ausserhalb der Halbebene durch Schnittpunkt



3. Rendering

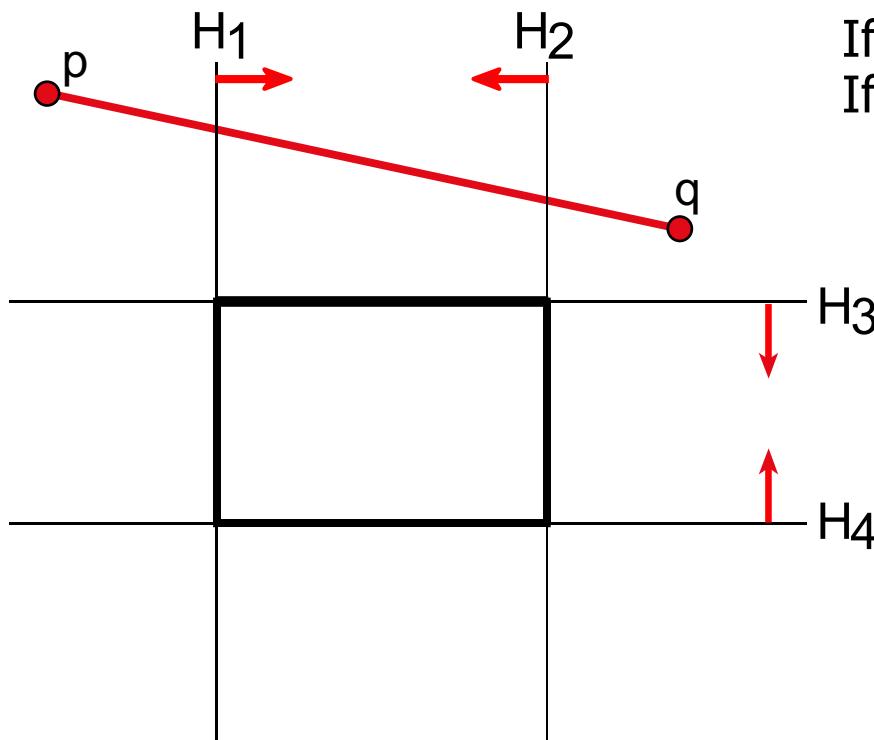
3.4 Clipping



2D Line Clipping: Naiver Ansatz

Algorithm:

```
For each frustum plane H  
  If  $H_p < 0$  and  $H_q < 0$ , clipped out; break  
  If  $H_p > 0$  and  $H_q > 0$ , pass through  
    break  
  If  $H_p < 0$  and  $H_q > 0$ , clip p to H  
  If  $H_q < 0$  and  $H_p > 0$ , clip q to H
```

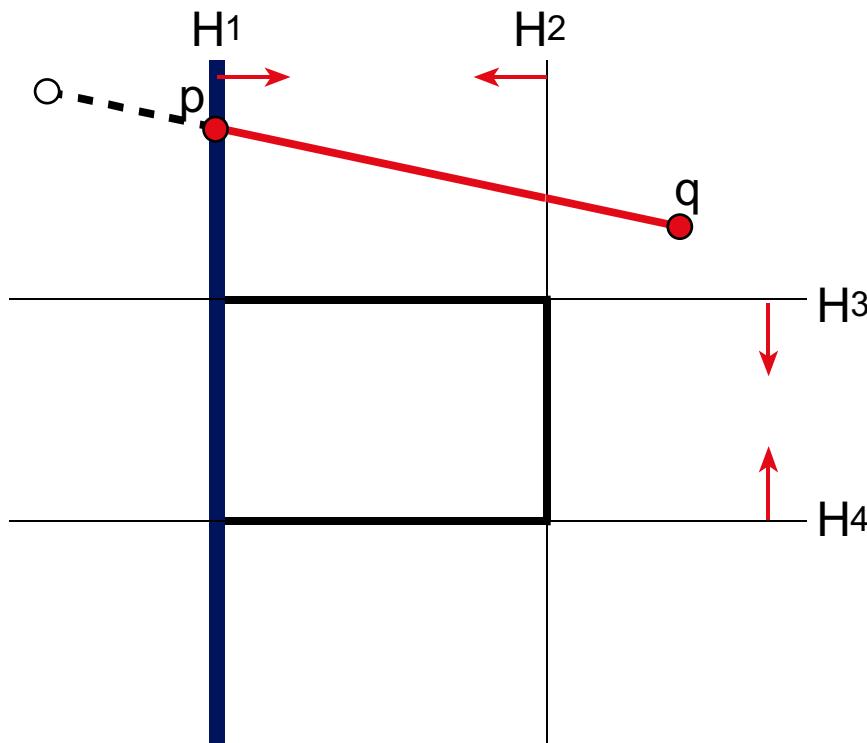


3. Rendering

3.4 Clipping



- Effizient?
- Bei einer Abarbeitung **left-right-top-bottom**:



- For each frustum plane H
 - If $H_p < 0$ and $H_q < 0$, clipped out; break
 - If $H_p > 0$ and $H_q > 0$, pass through break
 - If $H_p < 0$ and $H_q > 0$, clip p to H
 - If $H_q < 0$ and $H_p > 0$, clip q to H

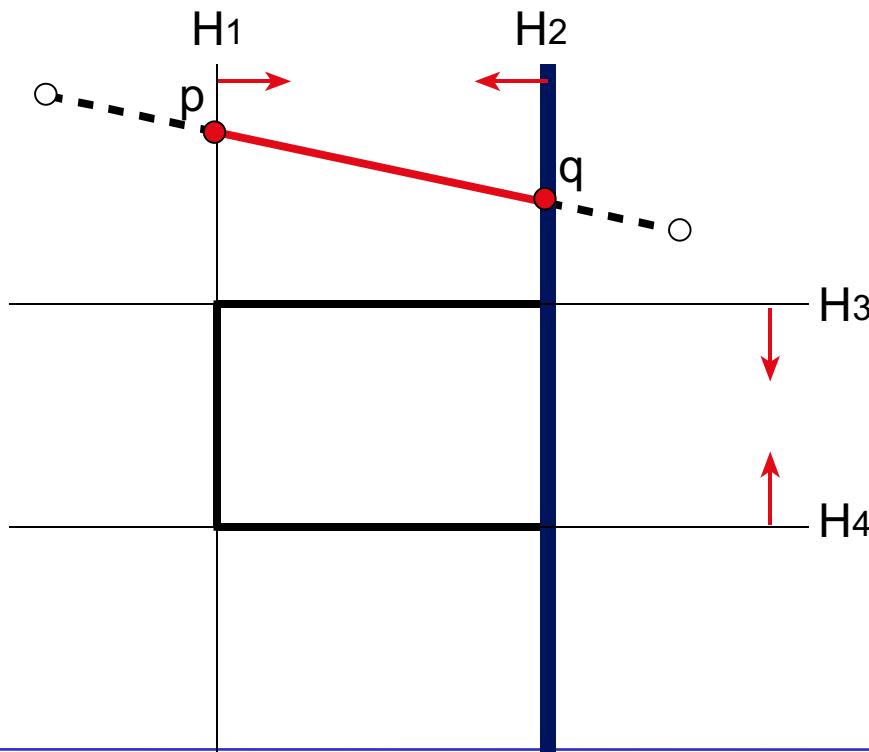
3. Rendering

3.4 Clipping



- Effizient?
- Bei einer Abarbeitung **left-right-top-bottom**:

- For each frustum plane H
 - If $H_p < 0$ and $H_q < 0$, clipped out; break
 - If $H_p > 0$ and $H_q > 0$, pass through
 - break
 - If $H_p < 0$ and $H_q > 0$, clip p to H
 - If $H_q < 0$ and $H_p > 0$, clip q to H

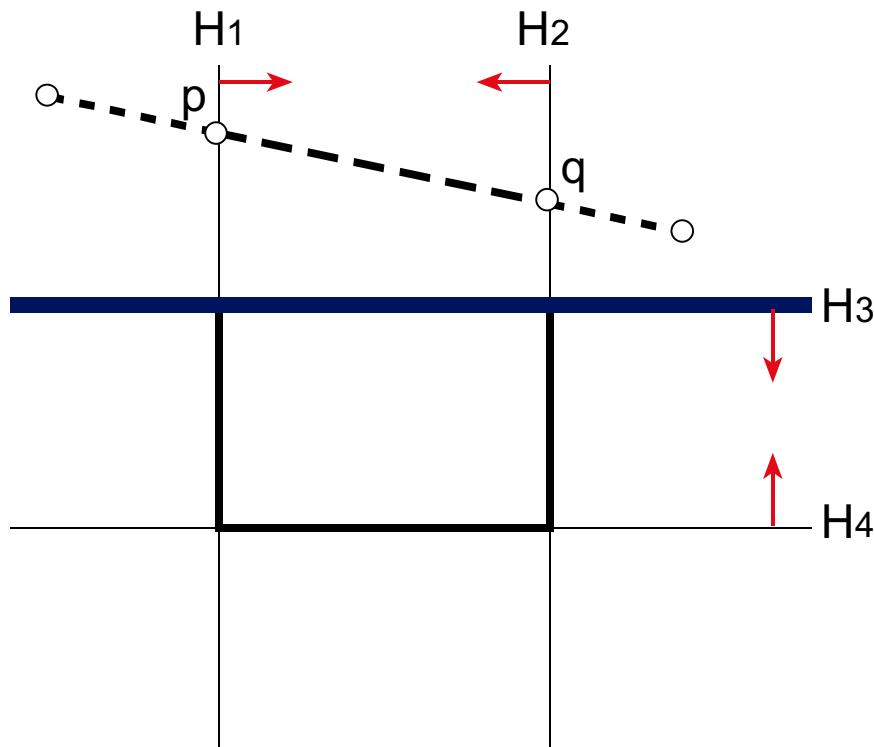


3. Rendering

3.4 Clipping



- Effizient?
- Bei einer Abarbeitung **left-right-top-bottom** :
 - For each frustum plane H
 - If $H_p < 0$ and $H_q < 0$, clipped out; break
 - If $H_p > 0$ and $H_q > 0$, pass through
break
 - If $H_p < 0$ and $H_q > 0$, clip p to H
 - If $H_q < 0$ and $H_p > 0$, clip q to H



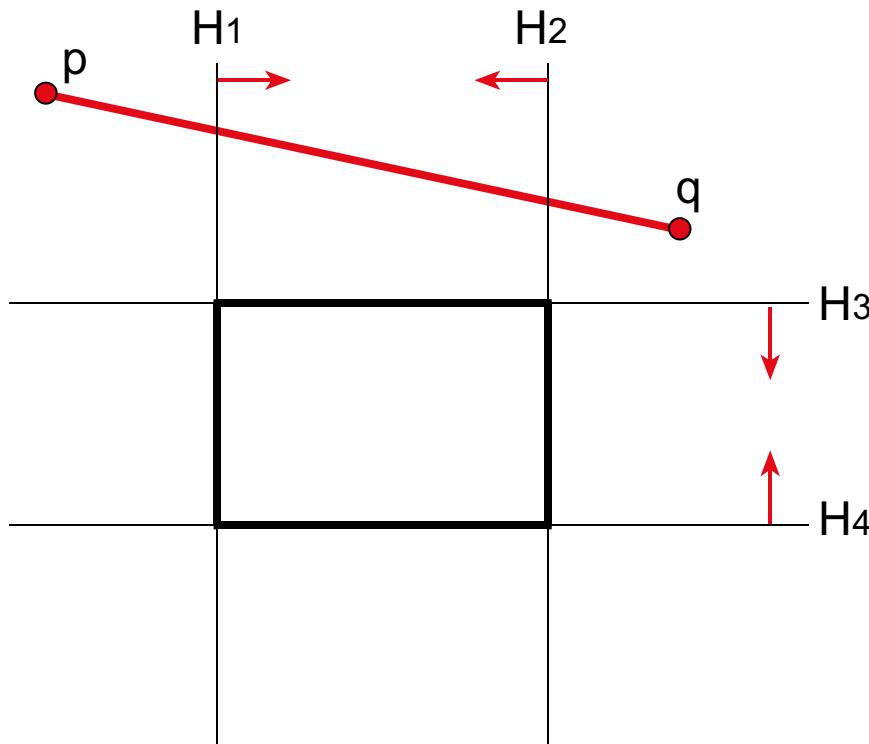
Was ist das Problem?

3. Rendering

3.4 Clipping



- For each frustum plane H
 - If $H_p < 0$ and $H_q < 0$, clipped out; break
 - If $H_p > 0$ and $H_q > 0$, pass through
break
 - If $H_p < 0$ and $H_q > 0$, clip p to H
 - If $H_q < 0$ and $H_p > 0$, clip q to H



Was ist das Problem?

Viele Berechnungen sind unnötig wenn wir mit der oberen Kante begonnen hätten

Wie können wir das frühzeitig erkennen?

3. Rendering

3.4 Clipping



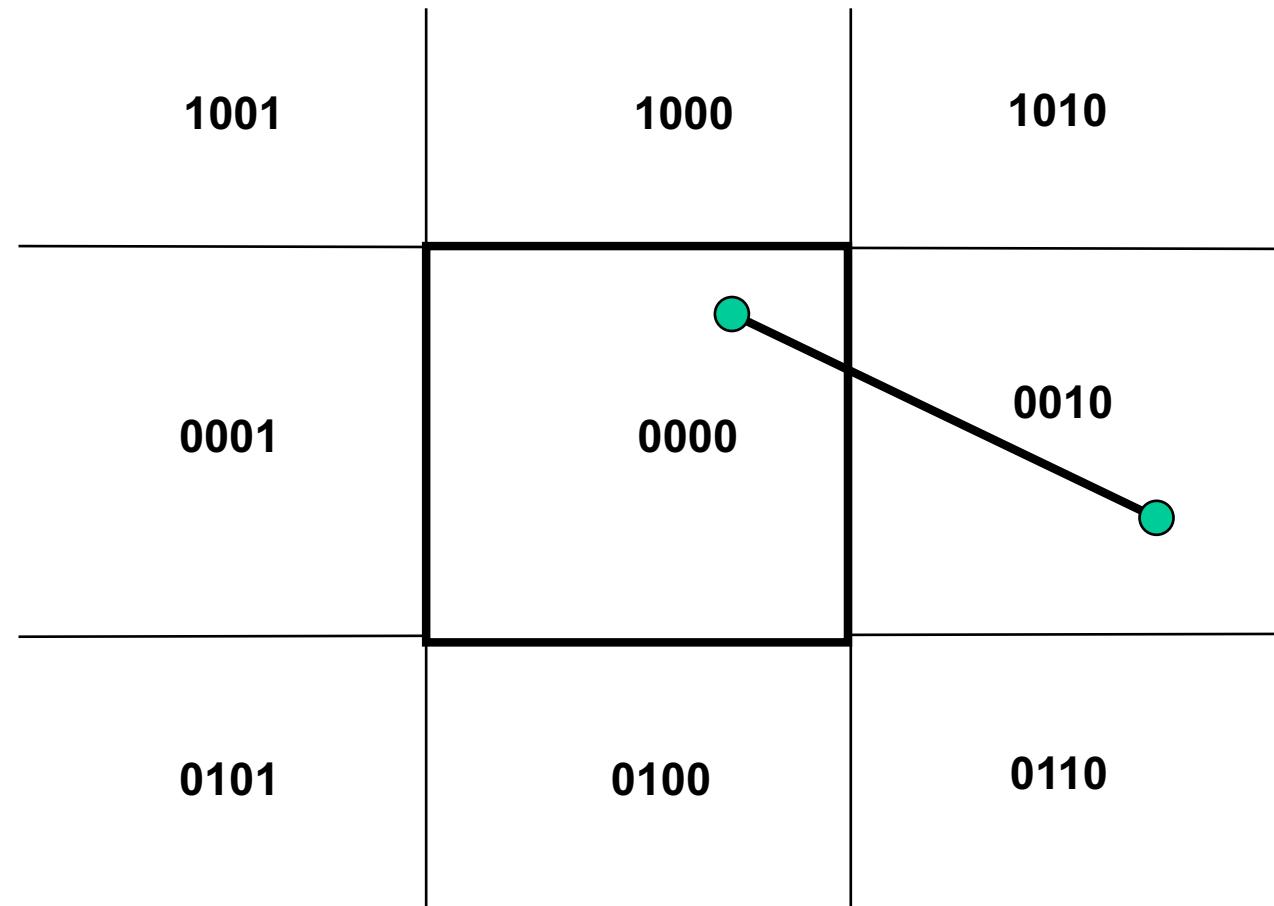
- **Cohen-Sutherland ('74)**
- Schnelles Identifizieren von Linien, die komplett drin oder komplett draussen sind
 - Einführung eines 4-bit (in 2D) region codes oder *outcode* für jeden Linien-Endpunkt
- Erweiterte die Kanten des windows bis unendlich
 - Aufteilung der Ebene in 9 Regionen
 - Beschrieben durch 4-bit outCode
- Vorteil: divide and conquer
 - Effizientes trivial accept und trivial reject
 - Nichttriviale Fälle: divide und neuer Test
- Jedes Bit des 4-bit outCode wird auf true/false gesetzt durch folgende Bedingungen:
 - bit 1 true \Rightarrow if point is *above* of window
 - bit 2 true \Rightarrow if point is *below* of window
 - bit 3 true \Rightarrow if point is *right* of window
 - bit 4 true \Rightarrow if point is *left* of window

3. Rendering

3.4 Clipping



- 2D Cohen-Sutherland





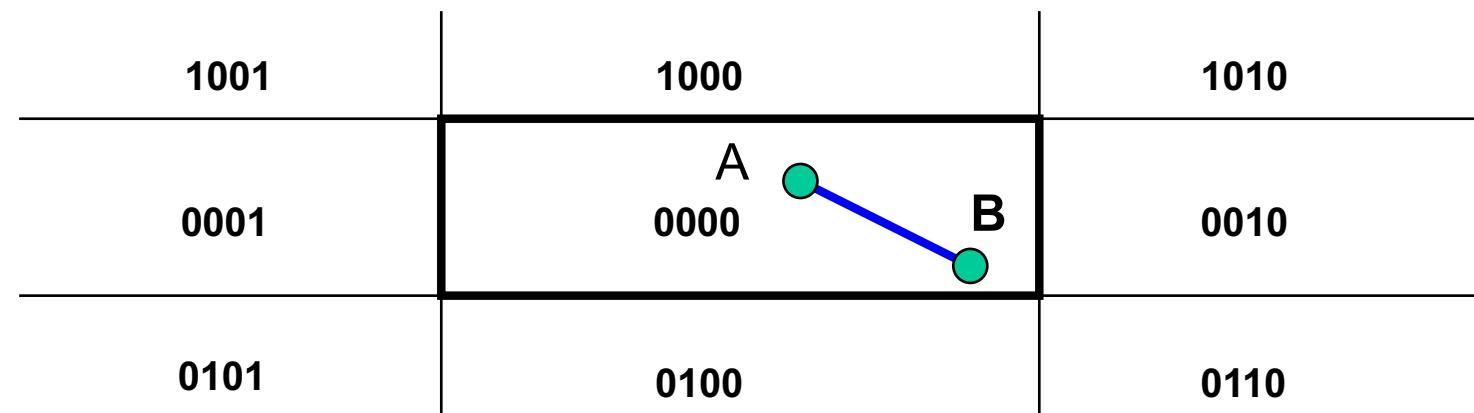
- **2D Cohen-Sutherland**
- 3 Fälle:
 - Eine Linie wird *trivially accepted* wenn *outcodes* beider Endpunkte *0000* sind.
 - Eine Linie wird *trivially rejected* wenn das *bitwise AND* der *outCodes* der beiden Endpunkte NICHT *0000* ist. (Linie ist vollständig ausserhalb mind. einer Window-Kante)
 - Sonst: Unterteilung der Linie und weitere Verarbeitung



- **Trivial Acceptance**

- $\text{outCode}(A) == \text{outCode}(B) == 0000$

Linie zur weiteren Verarbeitung



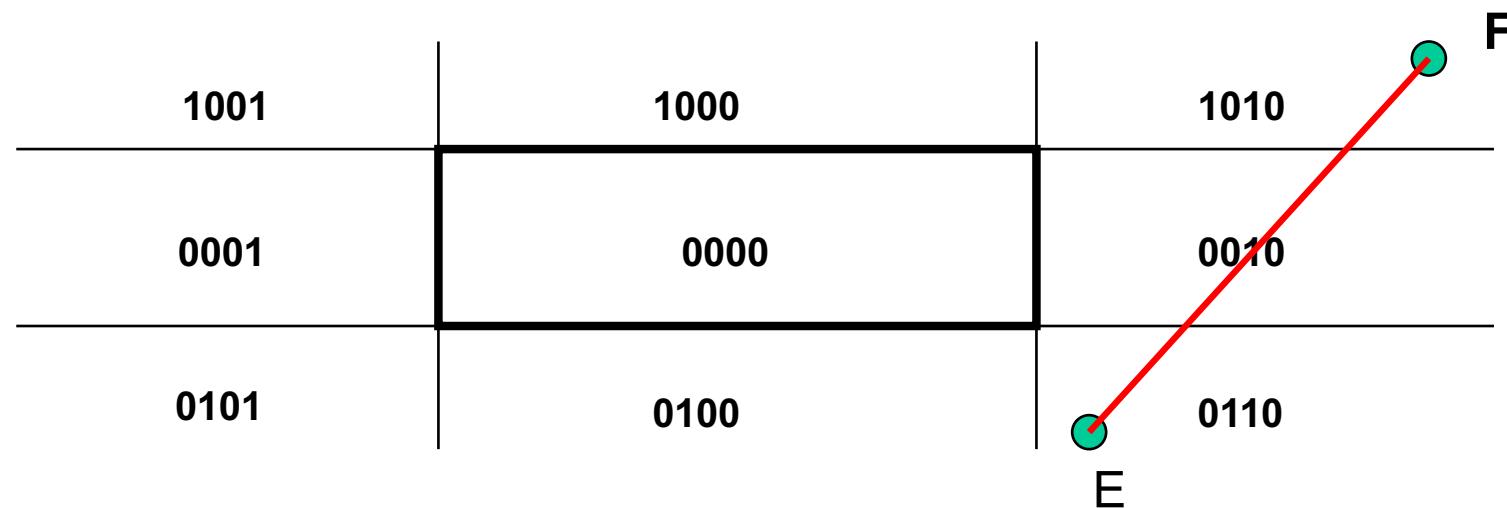
3. Rendering

3.4 Clipping



- **Trivial Rejection**

- $\text{outCode}(E) = 0110$
- $\text{outCode}(F) = 1010$
- $\text{outCode}(E) \&& \text{outCode}(F) = 0010$ (!!!)



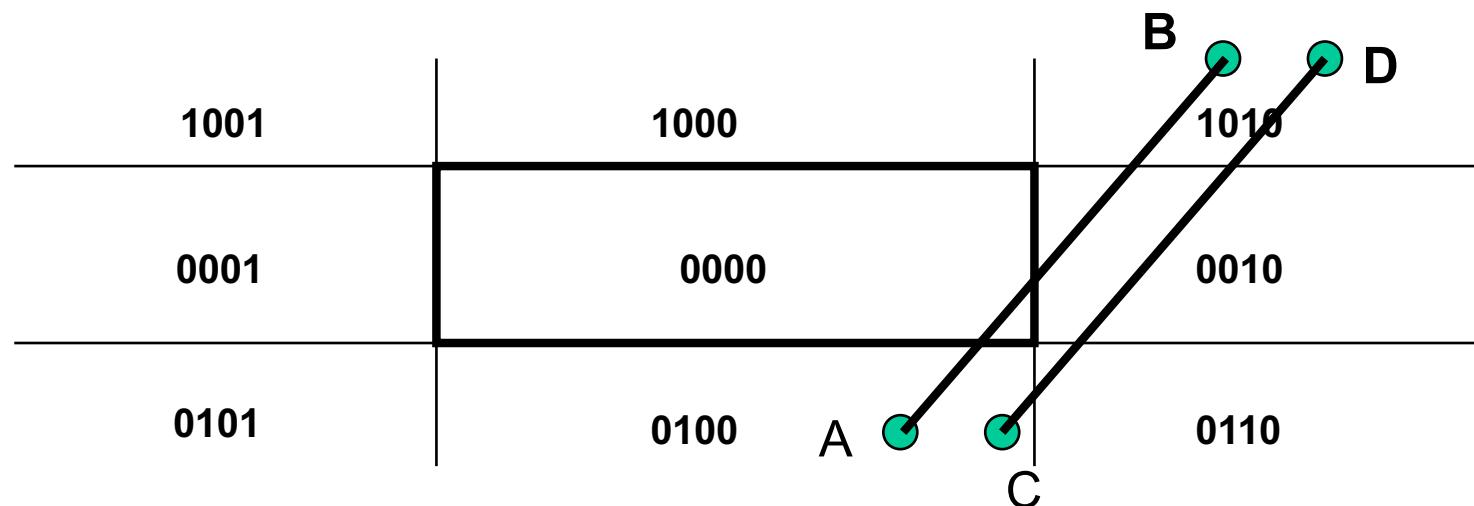
3. Rendering

3.4 Clipping



- **Unknown Cases**

- $\text{outCode}(A) = \text{outCode}(C) = 0100$
- $\text{outCode}(B) = \text{outCode}(D) = 1010$
- the bit-wise ‘and’ operation gives 0000

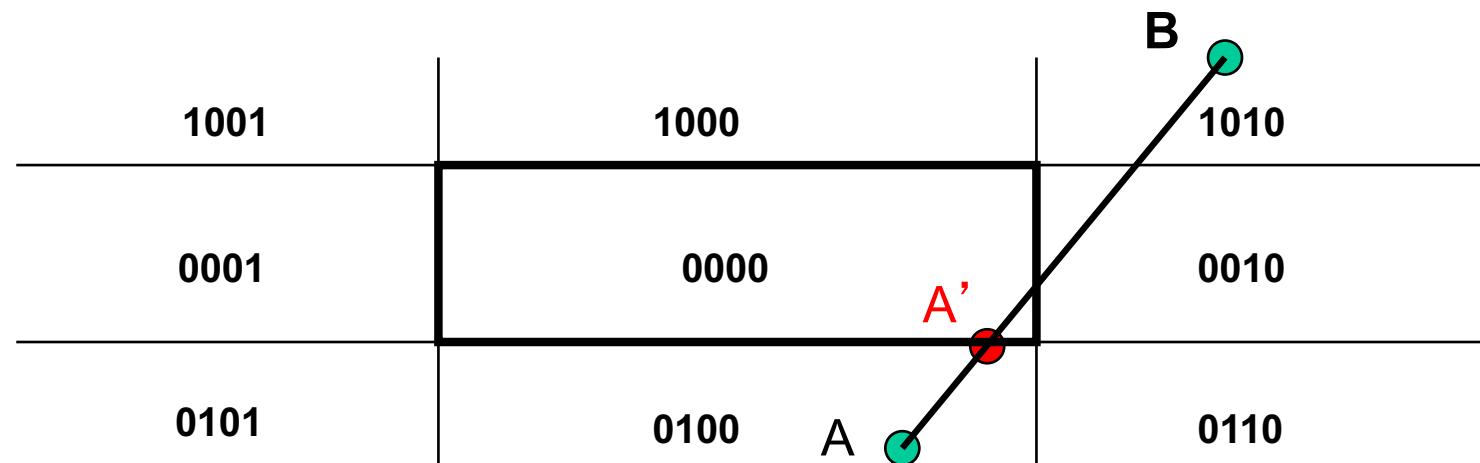


3. Rendering

3.4 Clipping



- **Unknown Cases**
- Wie fortfahren?
 - Berechne Schnitt A' mit einer Window-Kante (outcode gibt Information welche optimal ist) -> 2 neue Liniensegmente AA' und A' B
 - reject AA' und clippe A' B rekursiv



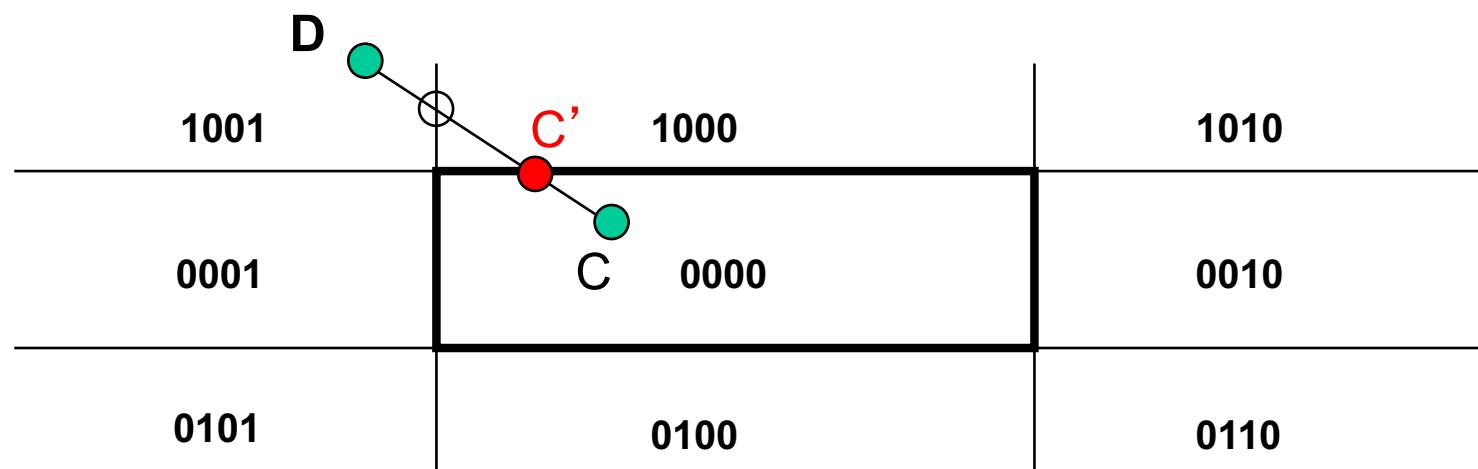
3. Rendering

3.4 Clipping



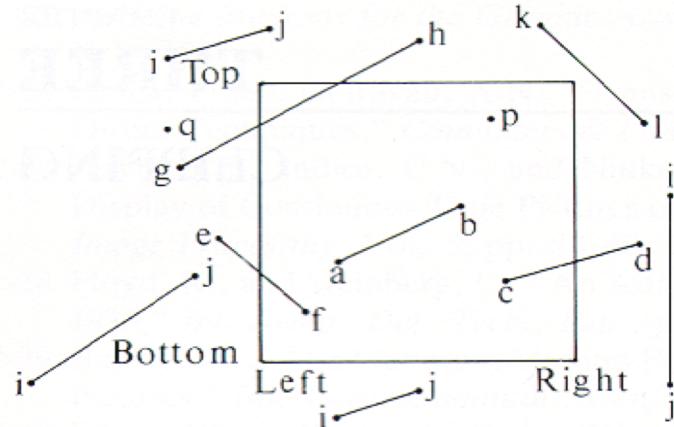
- **Unknown Cases**

- $\text{outCode}(C) = 0000, \text{outCode}(D) = 1001$
- bit-wise `and' operation gives 0000
- Berechne C' für 2 neue Liniensegmente DC' und $C'C$
- reject DC' und clippe $C'C$ rekursiv



3. Rendering

3.4 Clipping



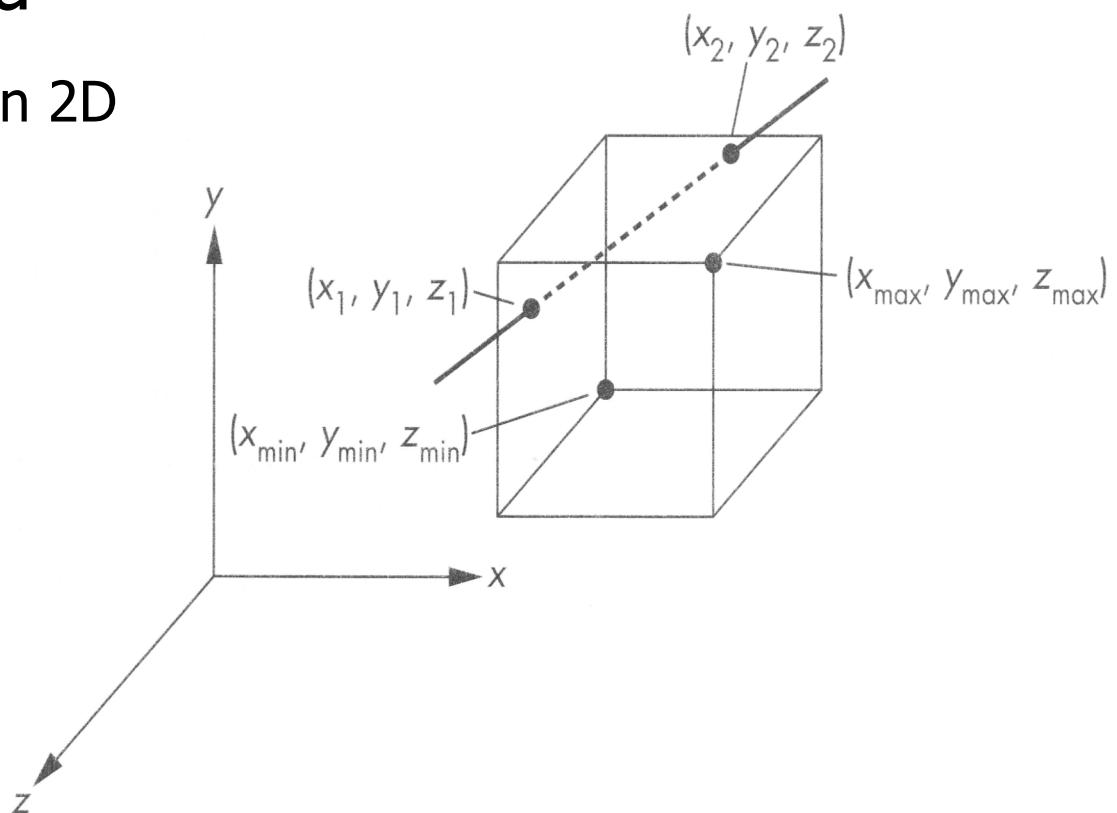
Cohen-Sutherland

Table 3-1 End Point Codes

Line (see Fig. 3-1)	End point codes (see Fig. 3-2)	Logical intersection	Comments
ab	0000 0000	0000	Totally visible
ij	0010 0110	0010	Totally invisible
ij	1001 1000	1000	Totally invisible
ij	0101 0001	0001	Totally invisible
ij	0100 0100	0100	Totally invisible
cd	0000 0010	0000	Partially visible
ef	0001 0000	0000	Partially visible
gh	0001 1000	0000	Partially visible
kl	1000 0010	0000	Totally invisible



- **3D Line Clipping**
- 3D Cohen-Sutherland
 - Direkte Erweiterung von 2D nach 3D
 - 9->27 verschiedene Regionen
 - 4->6-bit outCode

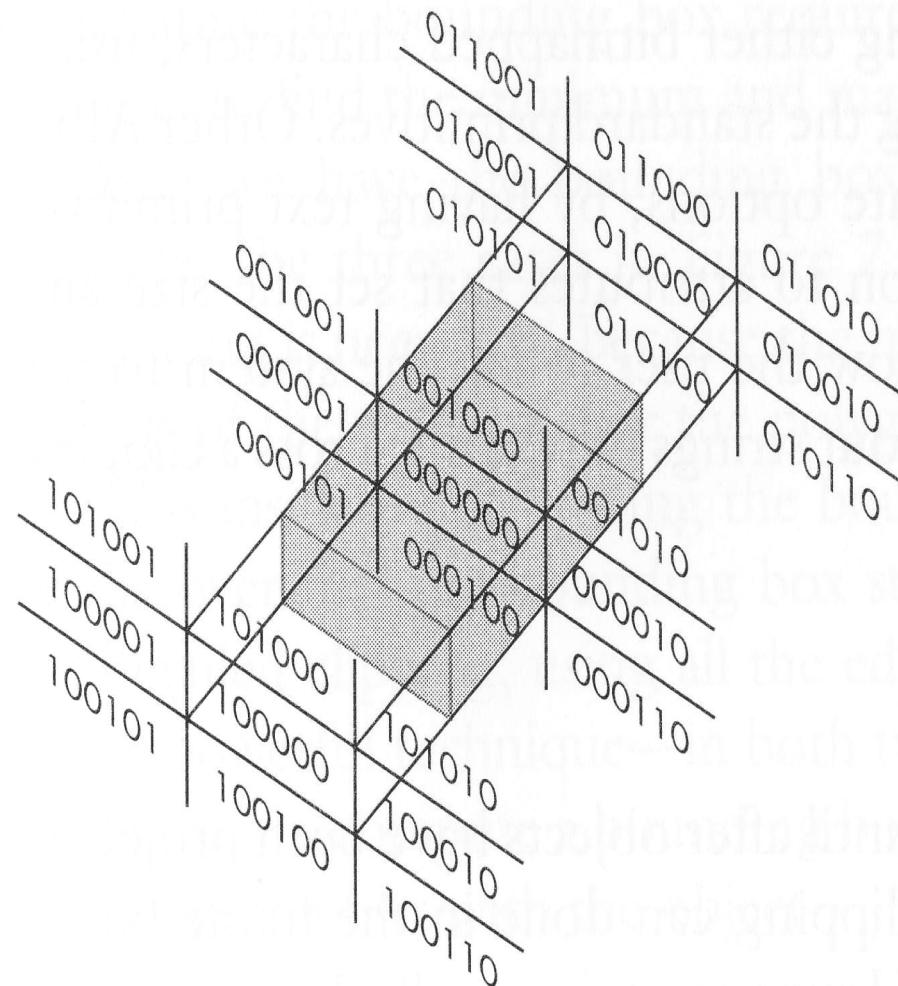


3. Rendering

3.4 Clipping

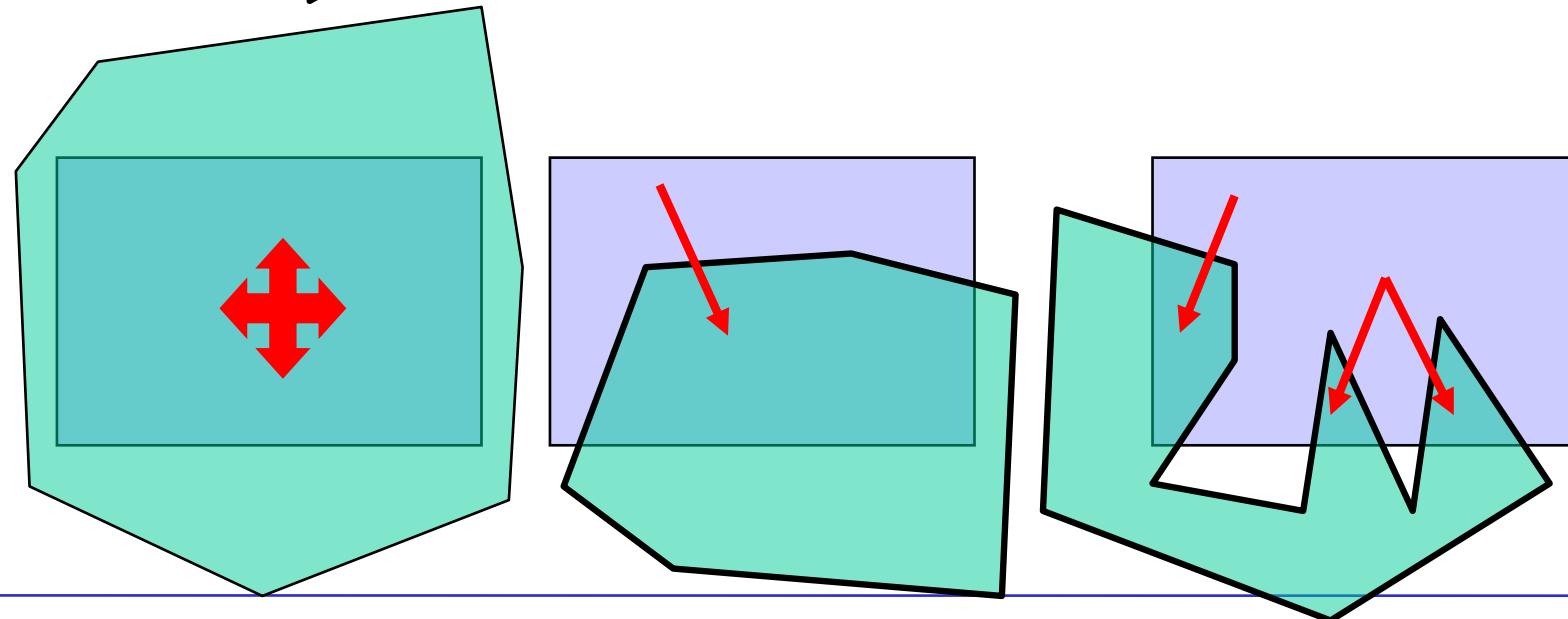
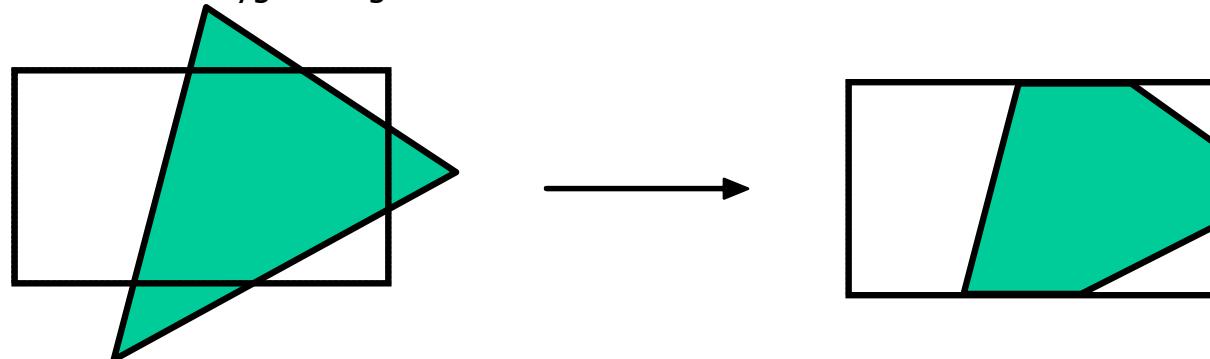


- OutCode in 3D





- **2D Polygon Clipping**
- Gegeben: Initialpolygon, gesucht: Teile innerhalb des viewports
 - Dies kann ein oder mehrere Polygone ergeben

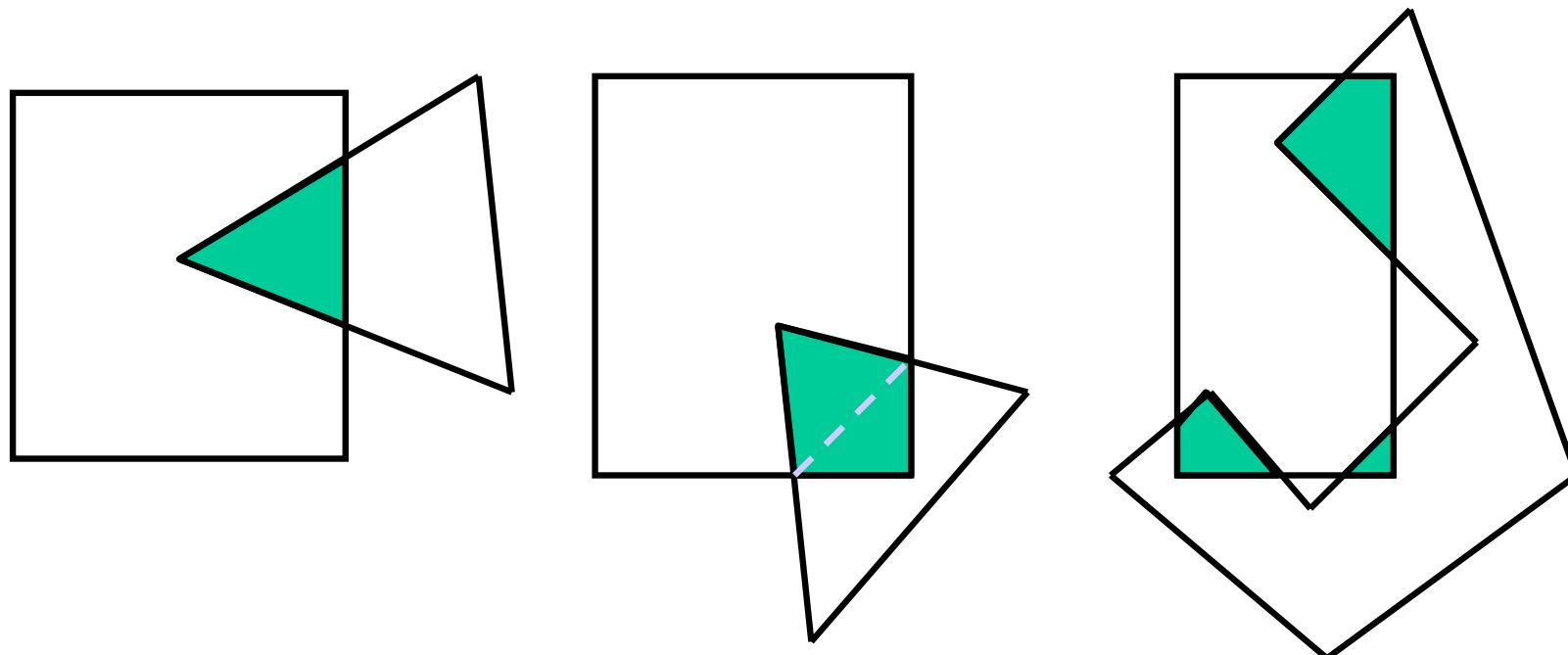


3. Rendering

3.4 Clipping



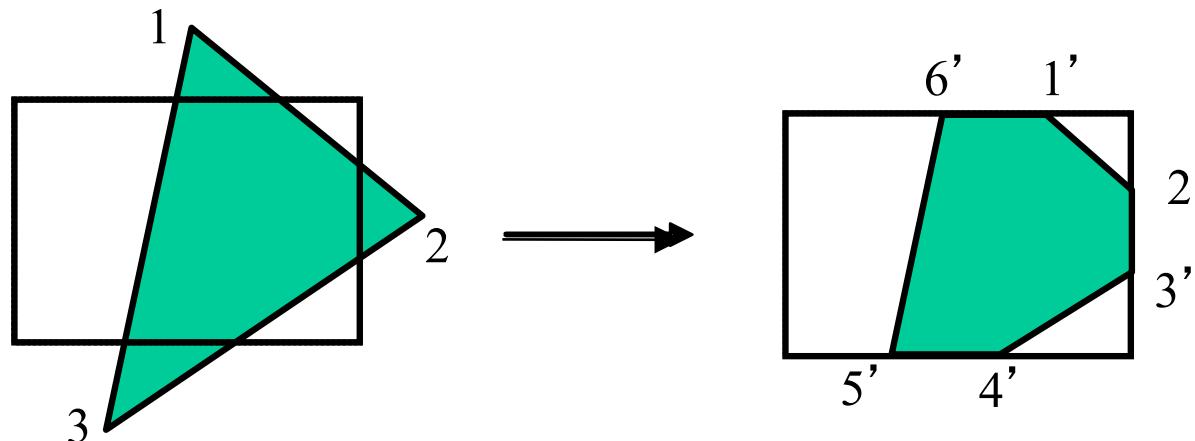
- **2D Polygon Clipping**
- In Wirklichkeit in 3D (4D) Clipping space
- Erweiterung von line clipping
 - Polygone müssen geschlossen bleiben
 - (Filling, hatching, shading, ...)





▪ **Polygon Clipping**

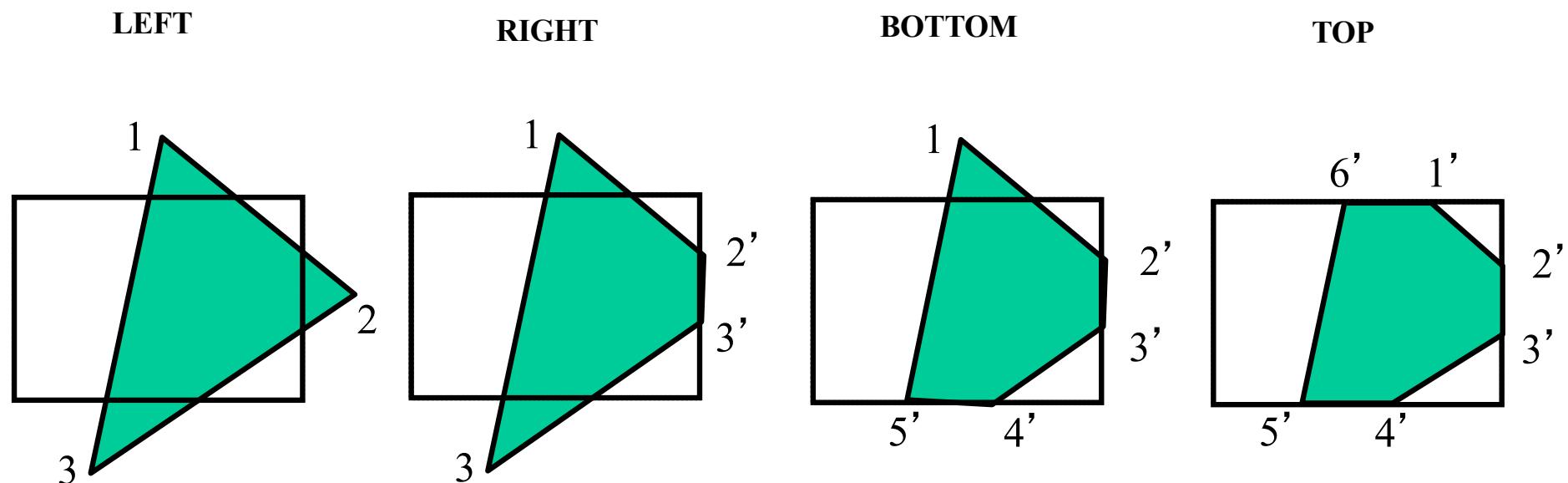
- Ein *Polygon* kann dargestellt werden als Folge von vertices
- Ziel: Clippen und Ausgeben einer neuen Folge von vertices, welche alle auf der sichtbaren Seite von jeder clipping plane sind:
 - input: $<1, 2, 3>$
 - output: $<1', 2', 3', 4', 5', 6'>$





- **Sutherland-Hodgman Algorithm**

- Idee: es ist leicht, ein Polygon gegen eine einzelne clipping plane zu clippen.
- Clippe Polygon gegen jede der 4 Kanten einzeln! (hier: left, right bottom, top)





- **Sutherland-Hodgman Algorithm**

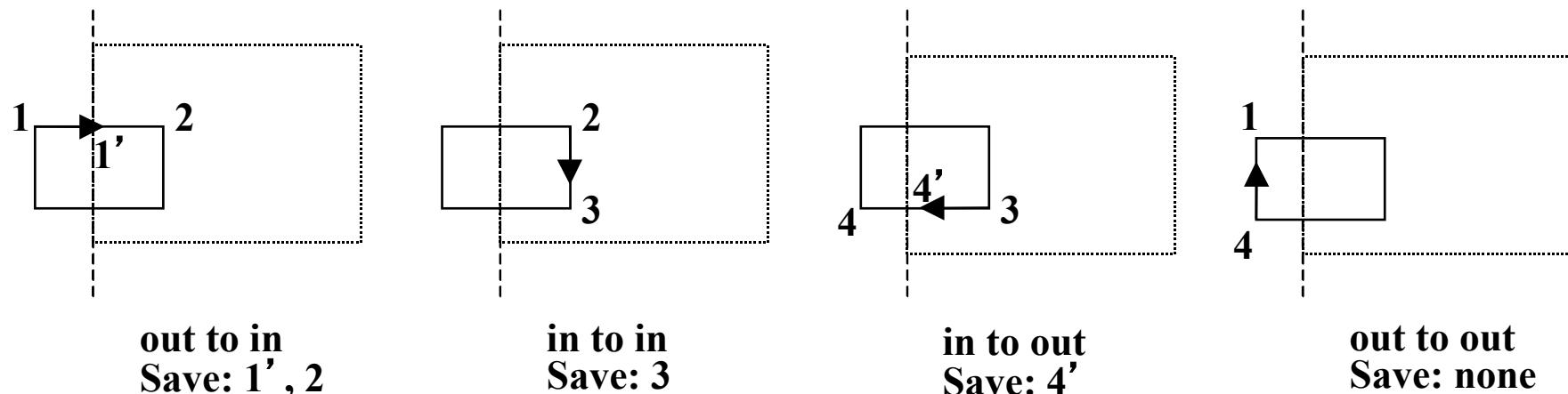
- Wie clippt man Polygon gegen einzelne window-Kante (clipping plane)?
- Traversiere Polygon in Uhrzeiger-Richtung, jede Kante des Polygons einzeln behandelt
- Dadurch muss nur die Relation zwischen jeweils einer Polygonkante und einer Clipping plane berechnet werden.
- Die Ordnung in welcher das Polygon gegen die Window-planes geclipped wird ist uninteressant.

3. Rendering

3.4 Clipping



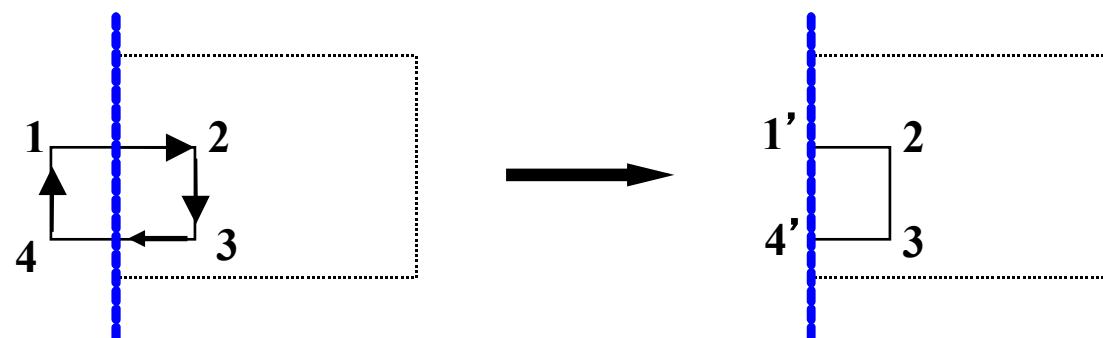
- **Sutherland-Hodgman**
- **Beim Durchlaufen eines Polygons gibt es immer nur 4 Möglichkeiten für jede Kante:**
 - going in of the window
 - two endpoints are inside the window
(i.e. on visible side of clipping boundary)
 - going out of the window
 - two endpoints are outside the window
- **Output: Schnittpunkt und sichtbarer Endvertex**





- **Sutherland-Hodgman**

- Das Ergebnis des Clipping des Polygons 1234 mit der linken Randfläche ergibt 1' 234'. Diese neue Folge von Vertices wird zum Clippen mit der nächsten Randfläche gesendet.

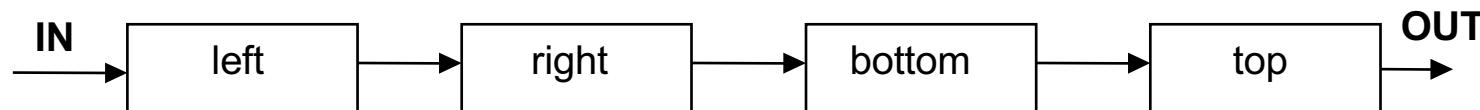
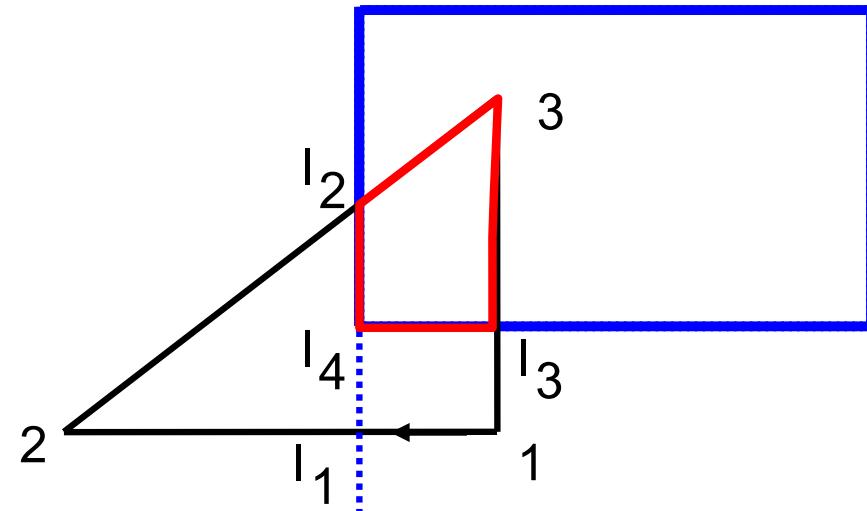


3. Rendering

3.4 Clipping

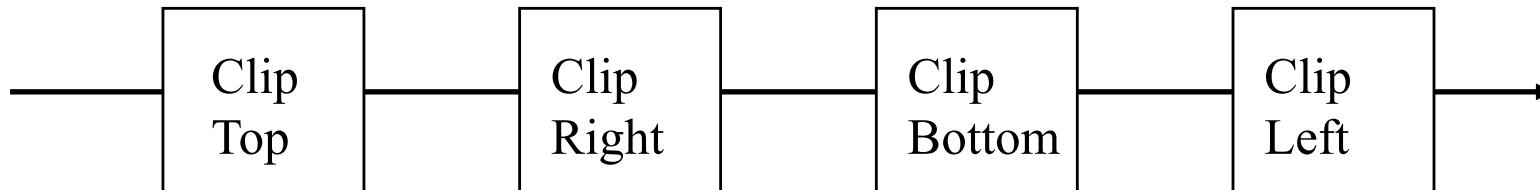


- Sutherland-Hodgman



3. Rendering

3.4 Clipping

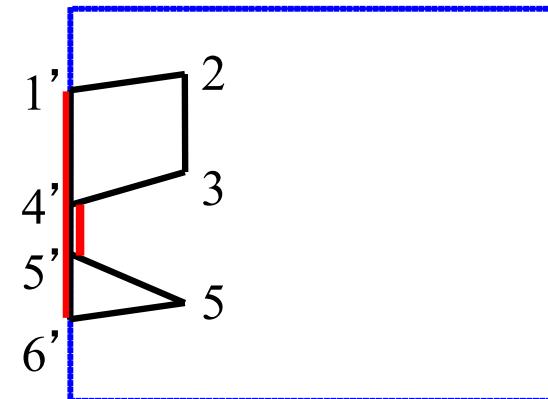
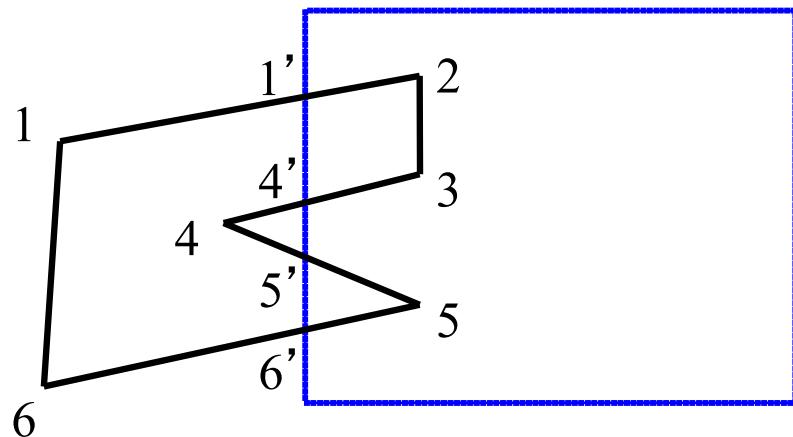


- **Pipelined Polygon Clipping**

- Da das clippen gegen eine clipping plane unabhängig von allen anderen clipping planes ist, können diese in einer **Pipeline** angeordnet werden.
- Dadurch können 4 Polygone gleichzeitig geclipped werden. Dies ist in Hardware implementiert.



- **Sutherland-Hodgman**
- Bemerkung: Sutherland-Hodgman clippt konvexe Polygone korrekt, bei konkaven Polygonen können ausgeartete Polygone entstehen:



- Dies ist so weil geclipppte konkave Polygone aus mehreren Teilen bestehen können, Sutherland-Hodgman aber immer nur ein geschlossenes Polygon liefert.



- **2D Polygon: Sutherland-Hodgman**

- **Für konkave Polygone:**

- teile Polygon in 2 oder mehr konvexe Polygone, und behandle diese separat.
- modifiziere Sutherland-Hodgman
- Nutze allgemeineren Polygon-clipper

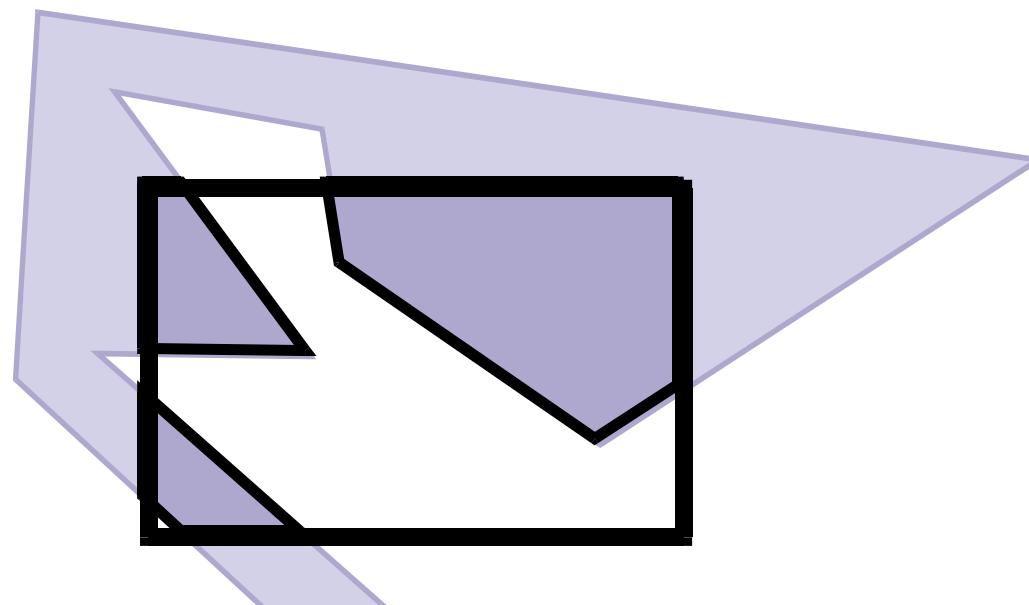


- **Zusammenfassung Sutherland–Hodgeman**
 - Clippe Polygon gegen jede Halbebene nacheinander
 - Funktioniert für jede konvexe Clipping-Region
 - Erweiterung von 2D zu 3D leicht
 - Nachbearbeitung kann nötig sein
 - Ausgeartete Polygone, unendlich dünn
 - Split in mehrere Polygone nötig



■ **Weiler-Atherton Algorithm**

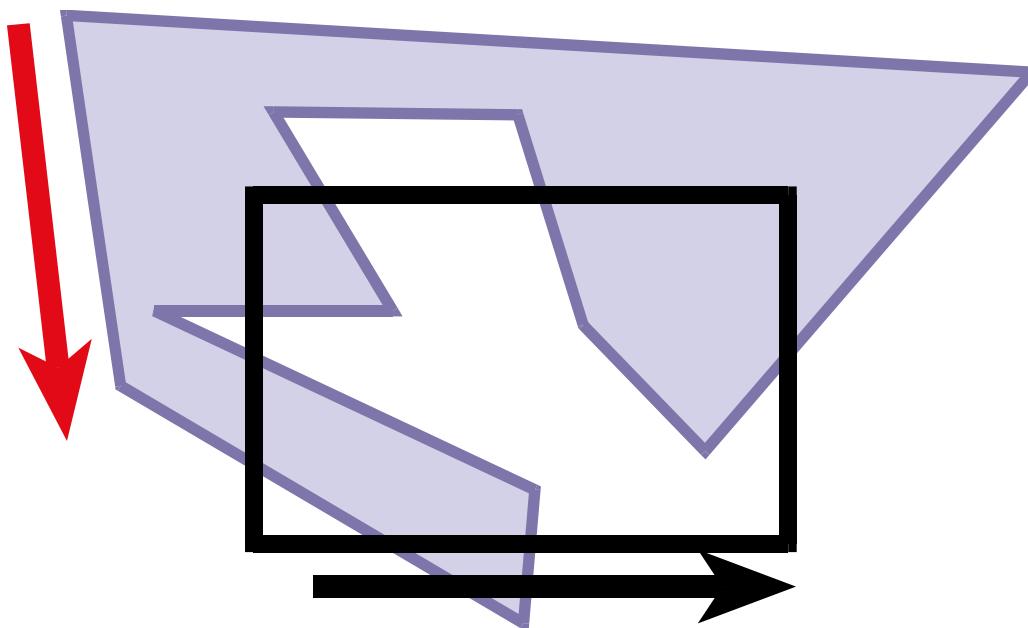
- **Erkenntnis:** Polygon clipping ist kompliziert für konkave Polygone
- **Jetzt:** generischer Polygon-Clipping Algorithmus
- Mächtig, aber auch komplexer





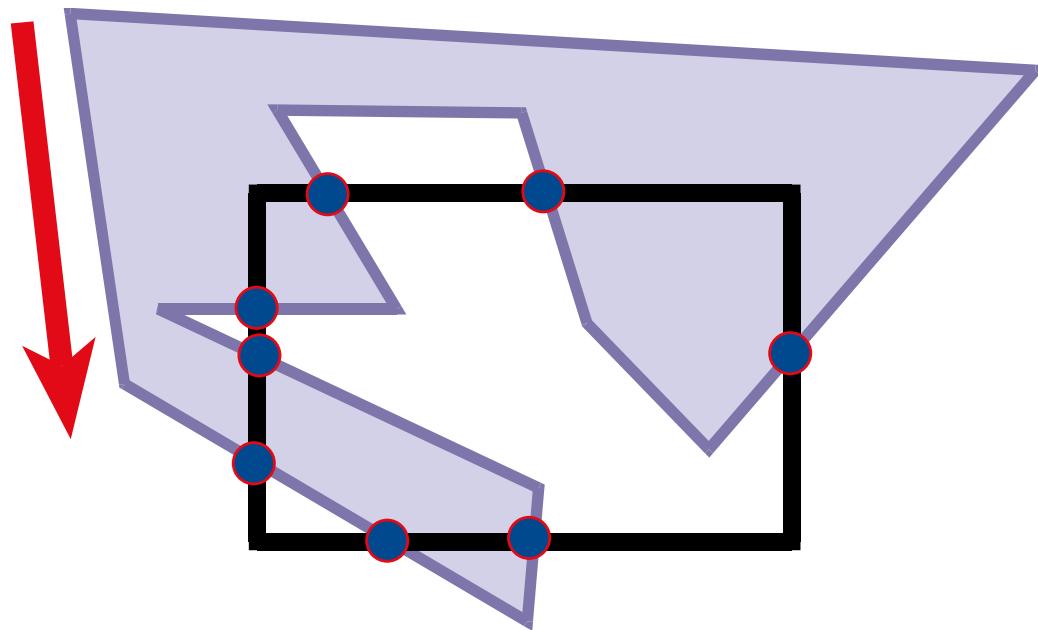
- **Weiler-Atherton Clipping**

- Strategie: "Walk" **polygon/window** boundary
- Polygone sind orientiert (**CCW**)





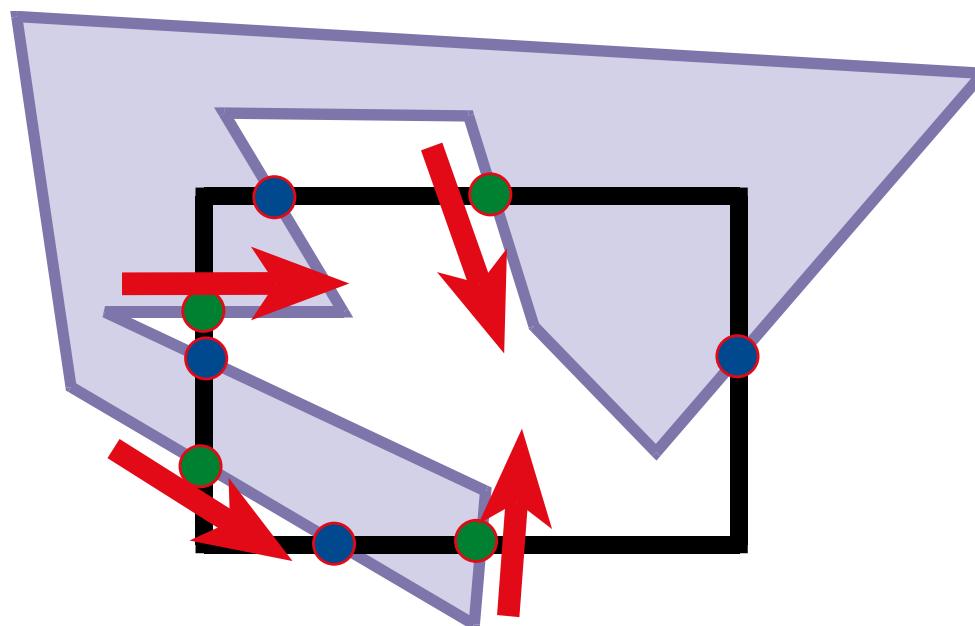
- **Weiler-Atherton Clipping**
 - Berechne Schnittpunkte beider Polygone





- **Weiler-Atherton Clipping**

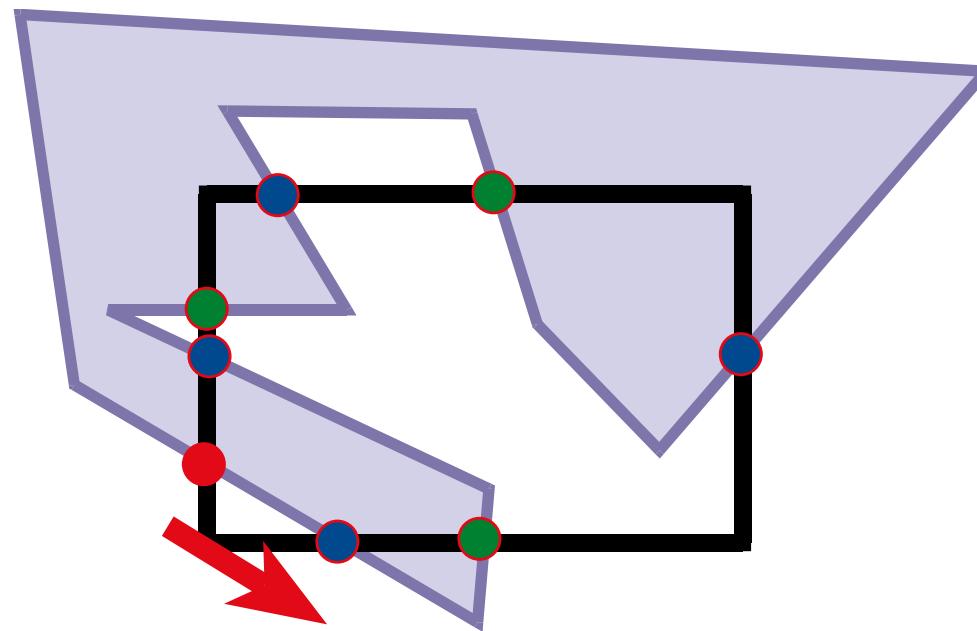
- Berechne Schnittpunkte
- Markiere Punkte an denen das Polygon in window eintritt (grün hier). Und Punkte wo das Polygon das window verlässt (Blau hier)
- Schaffe 2 Listen von orientierten Kanten: **polygon and window boundary lists** (mit grünen/blauen Schnittpunkten)





- **Clipping**

- While there is still an unprocessed entering intersection
- " Walk " polygon/window boundary



3. Rendering

3.4 Clipping



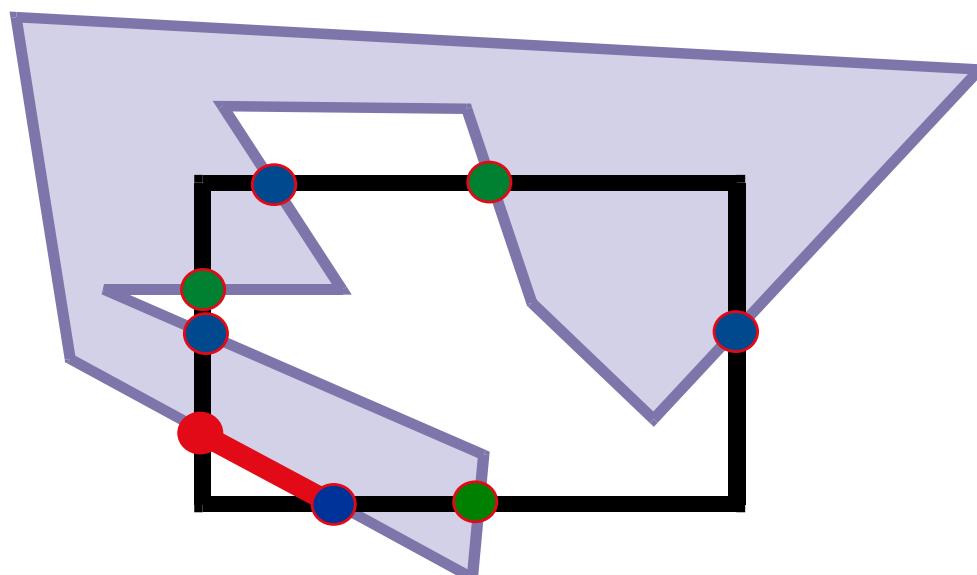
- **Walking rules**

- **Out-to-in:**

- Record clipped point
 - Follow polygon boundary (ccw)

- **In-to-out:**

- Record clipped point
 - Follow window boundary (ccw)



3. Rendering

3.4 Clipping



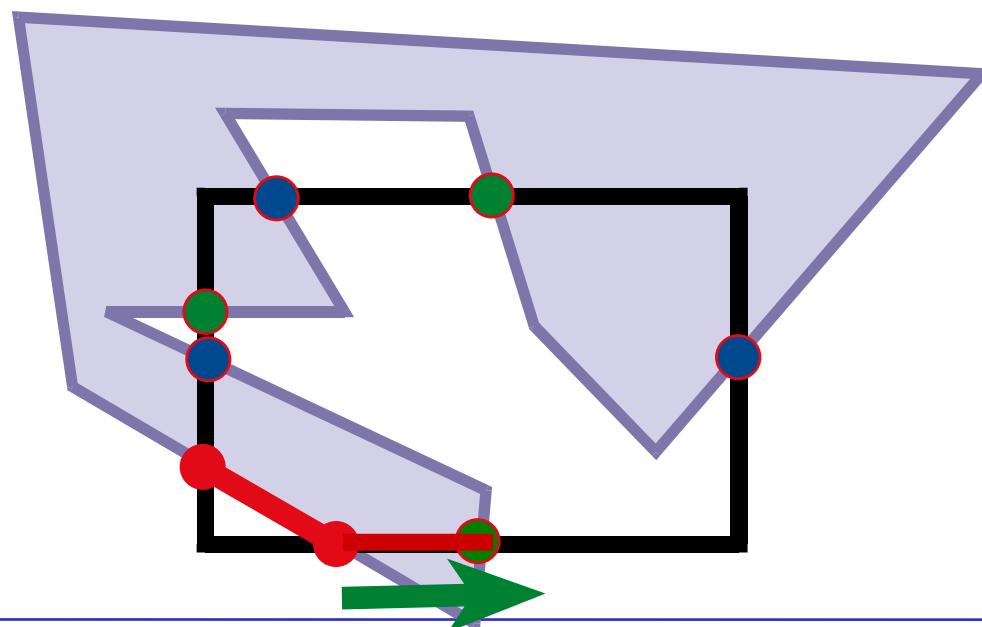
- **Walking rules**

- **Out-to-in:**

- Record clipped point
 - Follow polygon boundary (ccw)

- **In-to-out:**

- Record clipped point
 - Follow window boundary (ccw)



3. Rendering

3.4 Clipping



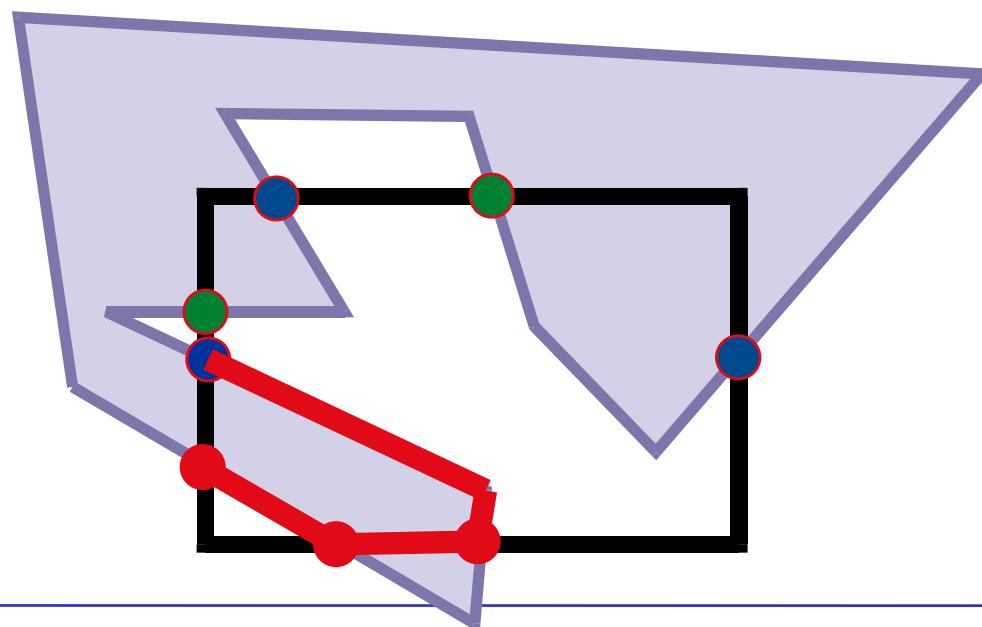
- **Walking rules**

- **Out-to-in:**

- Record clipped point
 - Follow polygon boundary (ccw)

- **In-to-out:**

- Record clipped point
 - Follow window boundary (ccw)

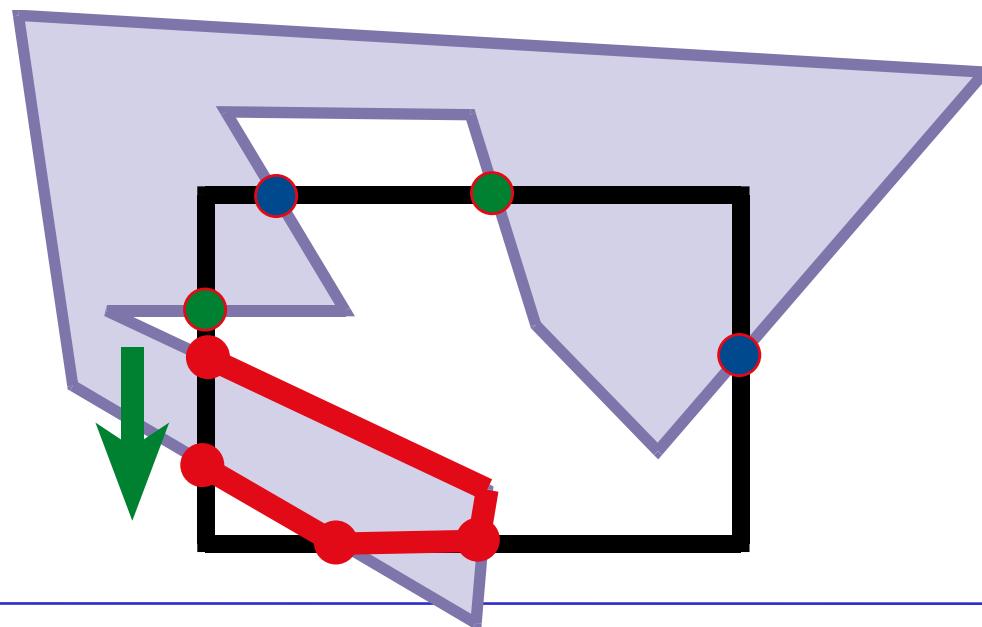


3. Rendering

3.4 Clipping



- **Walking rules**
- **Out-to-in:**
 - Record clipped point
 - Follow polygon boundary (ccw)
- **In-to-out:**
 - Record clipped point
 - Follow window boundary (ccw)

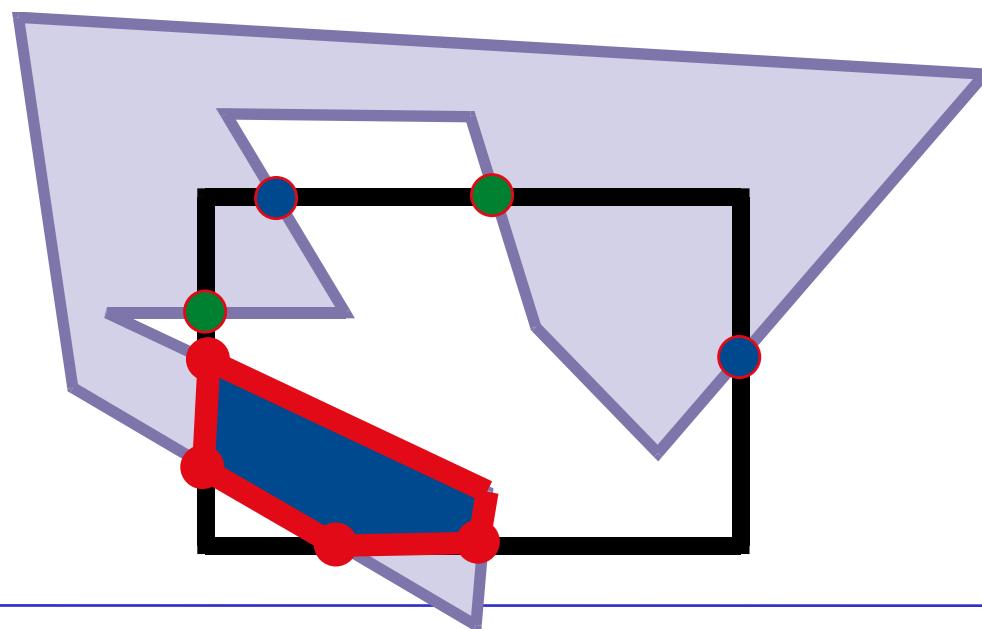


3. Rendering

3.4 Clipping



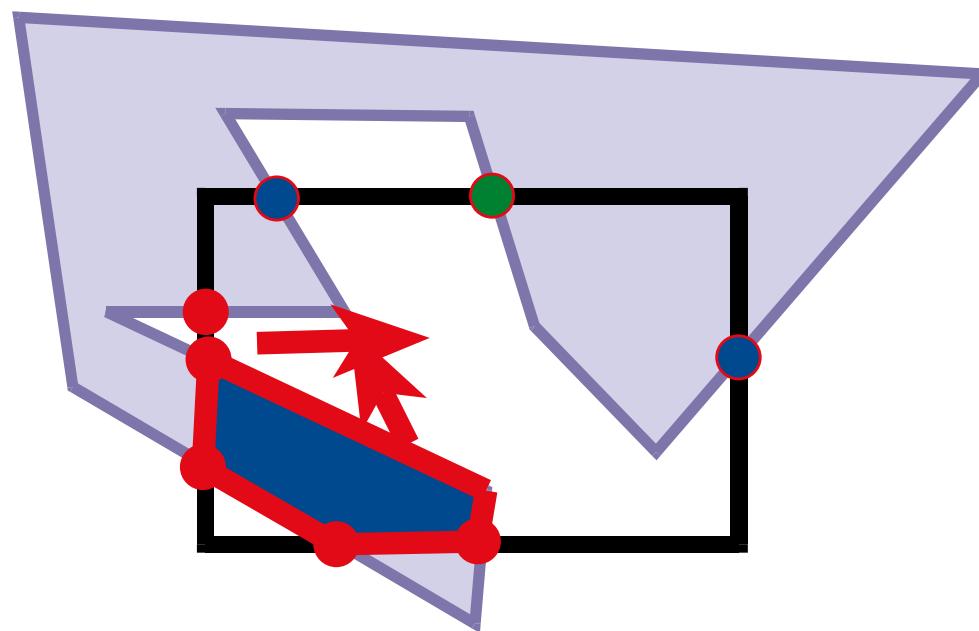
- **Walking rules**
- **Out-to-in:**
 - Record clipped point
 - Follow polygon boundary (ccw)
- **In-to-out:**
 - Record clipped point
 - Follow window boundary (ccw)





- **Walking rules**

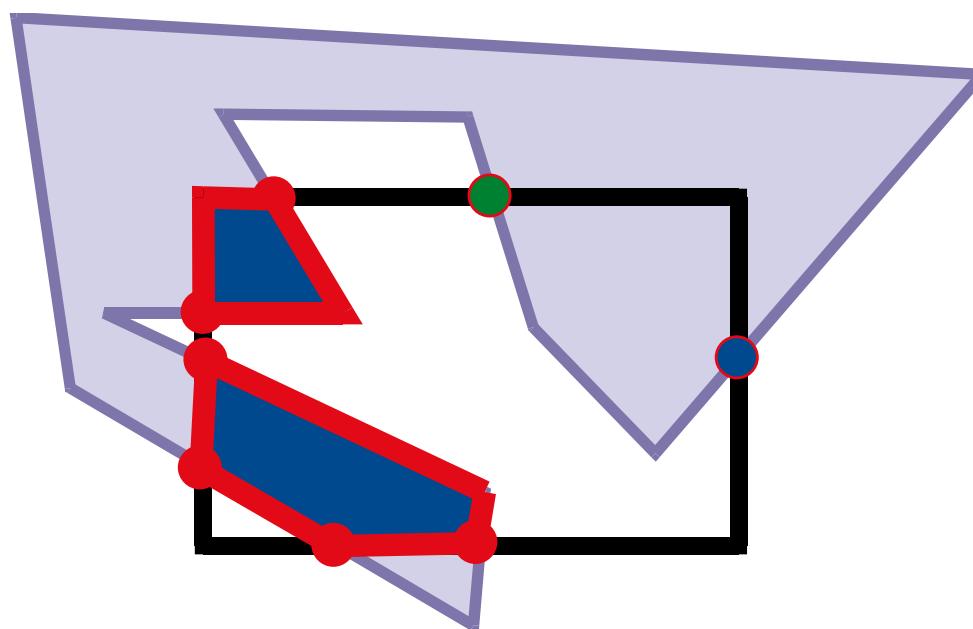
- While there is still an unprocessed entering intersection
 - "Walk" polygon/window boundary





- **Walking rules**

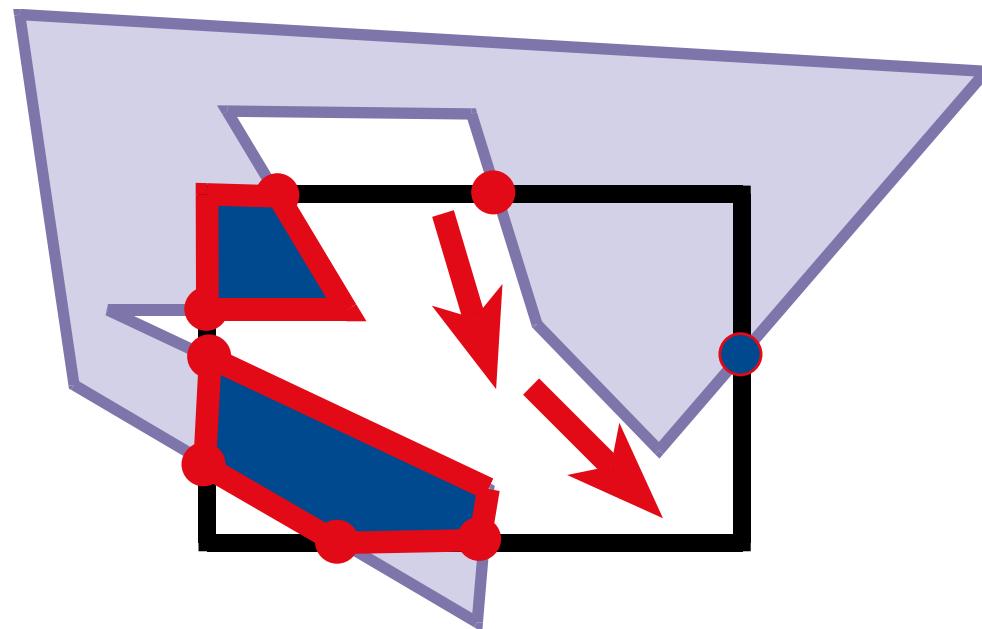
- While there is still an unprocessed entering intersection
- "Walk" polygon/window boundary





- **Walking rules**

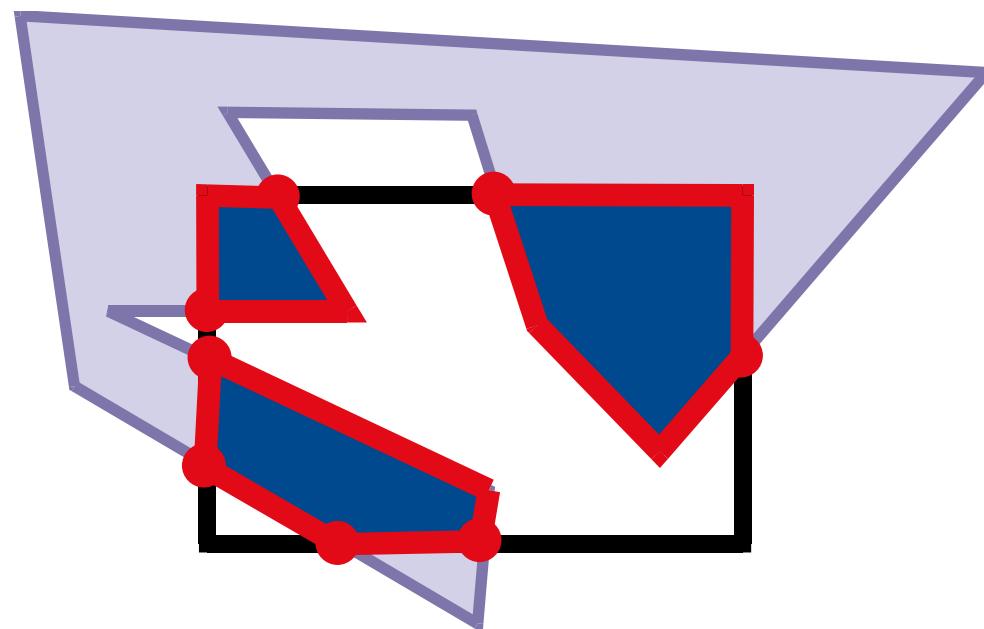
- While there is still an unprocessed entering intersection
 - "Walk" polygon/window boundary





- **Walking rules**

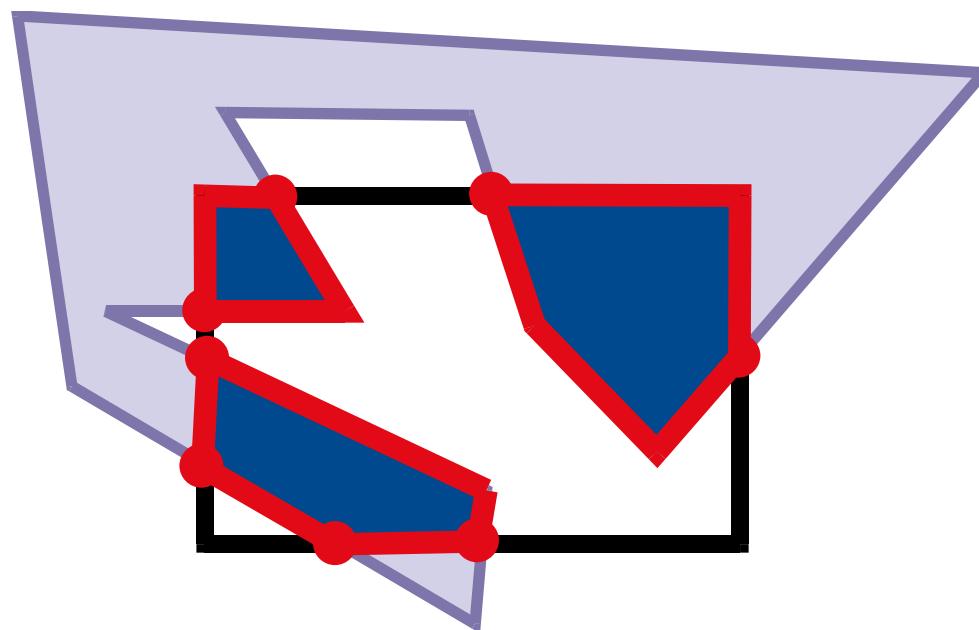
- While there is still an unprocessed entering intersection
 - "Walk" polygon/window boundary





- **Weiler-Atherton Clipping**

- Important: good adjacency data structure (here simple oriented edge list).



3. Rendering

3.4 Clipping



- **Summary**
 - **Culling**
 - Konservatives schnelles “Wegwerfen” ganzer Primitive/Objekte
 - Backface culling, (hierarchical) bounding boxes
 - **Clipping**
 - Behandlung teilweise sichtbarer Primitive/Objekte
 - In clipping coordinates
 - **Line Clipping**
 - Cohen-Sutherland
 - **Polygon Clipping**
 - Sutherland-Hodgman
 - Weiler-Atherton