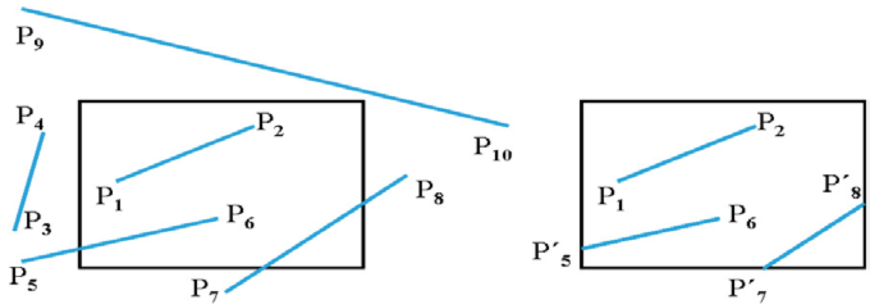
# 8. Clipping und Antialiasing

## Linien-Clipping

Clipping nennt man das Abschneiden von Bildteilen, die außerhalb des Darstellungsfensters liegen (siehe auch Abb. rechts). Je früher man die Clippingoperation in der Viewing-Pipeline durchführt, desto mehr unnötige nachfolgende Umformungen von ohnehin nicht sichtbaren Teilen erspart man sich:




- **Clipping in Weltkoordinaten** = analytische Berechnung zum frühestmöglichen Zeitpunkt,
- **Clipping in Clipkoordinaten** = analytische Berechnung an achsenparallelen Grenzen (einfach!),
- **Clipping bei der Rasterkonversion** = innerhalb des Algorithmus, der ein Graphikprimitiv in Punkte umformt.

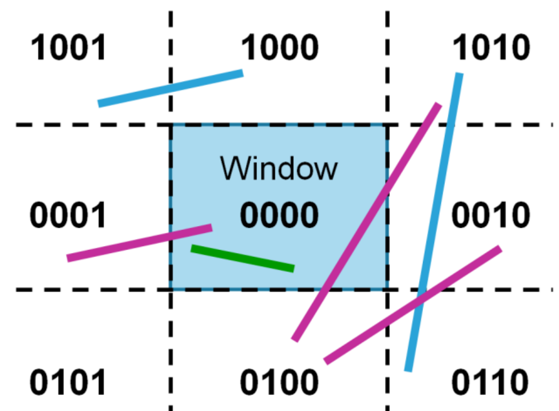
Clipping ist eine sehr häufige Operation, daher muss sie einfach und schnell sein.

### Clippen von Linien: Cohen-Sutherland-Verfahren

Algorithmen zum Clippen von Linien nützen i.A. die Tatsache aus, dass jede Linie in einem rechteckigen Fenster höchstens einen sichtbaren Teil besitzt. Weiters gilt es Grundprinzipien der Effizienz auszunutzen, etwa häufige einfache Fälle früh zu eliminieren und unnötige teure Operationen (Schnittpunkt-Berechnungen) zu vermeiden. Einfachstes Linienclipping könnte etwa so aussehen:

```
for endpoints (x0,y0), (xend,yend)
intersect parametric representation
      x = x0 + u*(xend - x0)
      y = y0 + u*(yend - y0)
with window borders:
      intersection ⇔ 0 < u < 1
```

Der Cohen-Sutherland-Algorithmus klassifiziert zuerst die Endpunkte einer Linie hinsichtlich ihrer Lage zum Clippingfenster: oben, unten, links, rechts, und codiert diese Information in 4 Bit. Nun kann man schnell überprüfen:




1. OR der beiden Codes = 0000 ⇒ Linie ganz sichtbar
2. AND der beiden Codes ≠ 0000 ⇒ Linie ganz unsichtbar
3. andernfalls mit einer relevanten Fensterkante schneiden, und den weggeschnittenen Punkt durch den Schnittpunkt ersetzen. GOTO 1.

Schnittpunktberechnungen:
mit vertikalen Fensterkanten:

$$y = y_0 + m(xw_{min} - x_0), \qquad y = y_0 + m(xw_{max} - x_0)$$

mit horizontalen Fensterkanten:

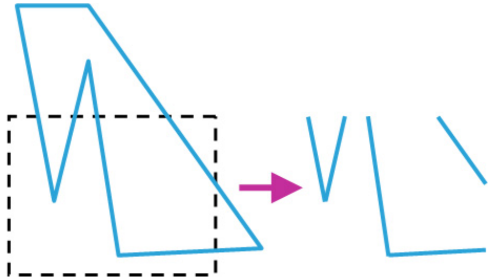
$$x = x_0 + (yw_{min} - y_0)/m, \qquad x = x_0 + (yw_{max} - y_0)/m$$

Punkte genau auf den Fensterkanten müssen natürlich als innerhalb gelten, dann kann es zu höchstens 4 Schleifendurchläufen kommen. Wie man auch sieht, werden nur dann Schnittpunktberechnungen durchgeführt, wenn es wirklich notwendig ist.

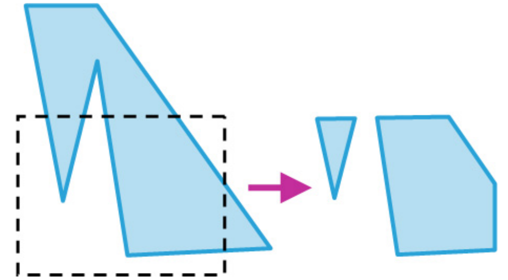
Für das Clippen von Kreisen gibt es ähnliche Verfahren. Man muss natürlich berücksichtigen, dass Kreise beim Clippen in mehrere Teile zerfallen können.

# Polygon-Clipping

Das Clippen von Polygonen hat zu berücksichtigen, dass nach dem Clippen als Resultat wieder *ein* Polygon erzeugt wird, auch wenn durch den Clipping-Vorgang mehrere Teile entstehen. Die linke Abbildung zeigt ein Polygon, das

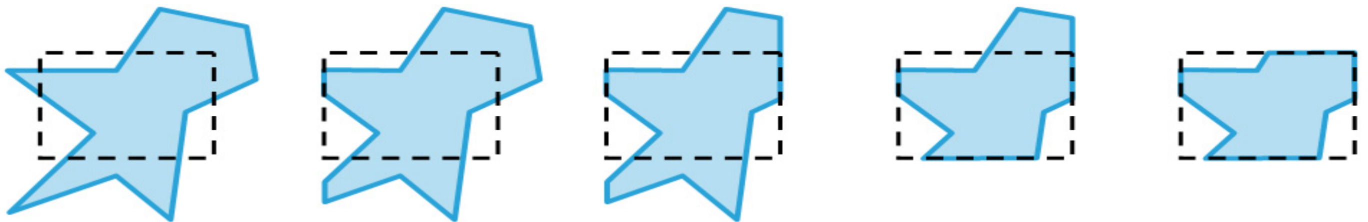


mit einem Linien-Clipping-Algorithmus geclippt wurde. Es ist nicht mehr erkennbar, was innen und was außen ist. Das rechte Bild zeigt das Ergebnis eines korrekten Polygon-Clipping-Verfahrens. Das Polygon zerfällt in mehrere Teile, die alle korrekt gefüllt werden können.

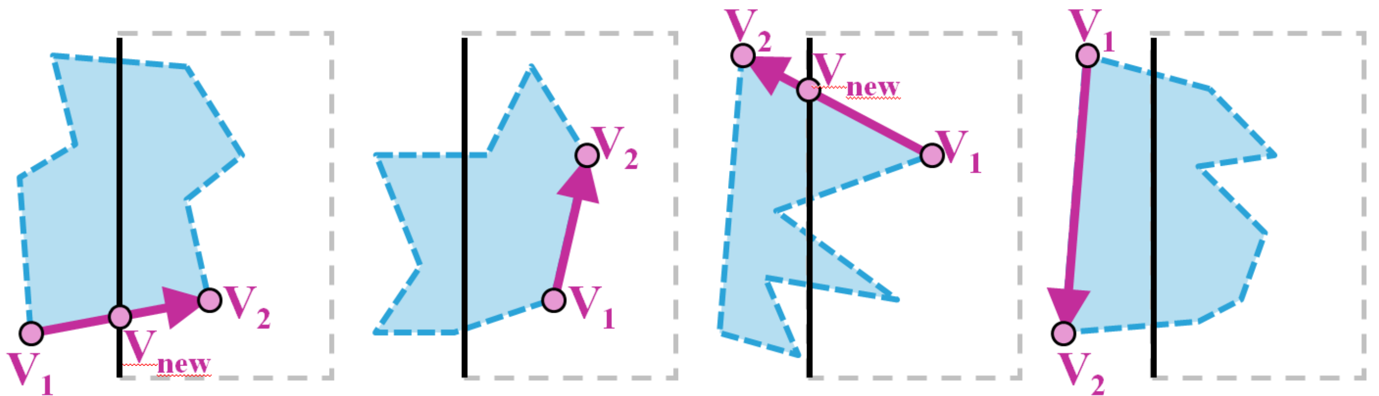


## Clippen von Polygonen: Sutherland-Hodgman-Verfahren

Die Grundidee kommt von der Erkenntnis, dass beim Clippen an nur einer Kante keine gröberen Komplikationen entstehen. Daher wird das Polygon nacheinander an allen 4 Fensterkanten geclippt, und das Ergebnis jeweils als Input für die nächste Kante verwendet:

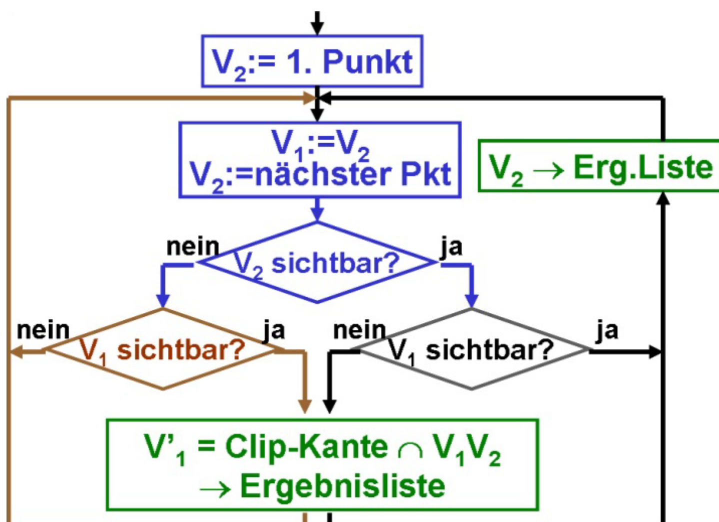


Es gibt 4 verschiedene Fälle, wie sich eine Kante ($V_1$, $V_2$) bezüglich einer Fensterkante verhalten kann. Bei der sequenziellen Abarbeitung der Polygonkanten können dabei folgende Fälle auftreten:




1. out→in (Output: $V_{new}$, $V_2$) 2. in→in (Output: $V_2$) 3. in→out (Output: $V_{new}$) 4. out→out (kein Output)

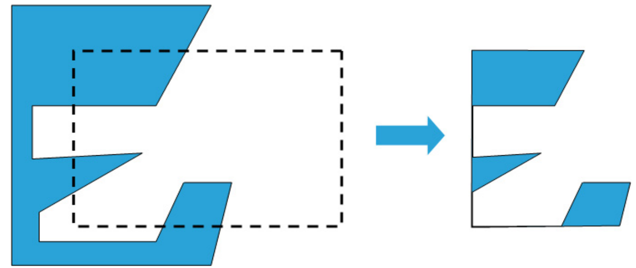
Der Algorithmus ***für <u>eine</u> Fensterkante*** läuft dann so ab:




Die Eckpunkte des Polygons werden sequenziell bearbeitet. Für jede Polygonkante wird festgestellt, welchem der 4 Fälle sie zuzurechnen ist und der entsprechende Eintrag in die Ergebnisliste erzeugt. Die Ergebnisliste enthält nach dieser Bearbeitung die Eckpunkte des an dieser Kante geclippten Polygons, ist also wieder ein gültiges Polygon und dient als Input für die Clippingoperation an der nächsten Fensterkante.

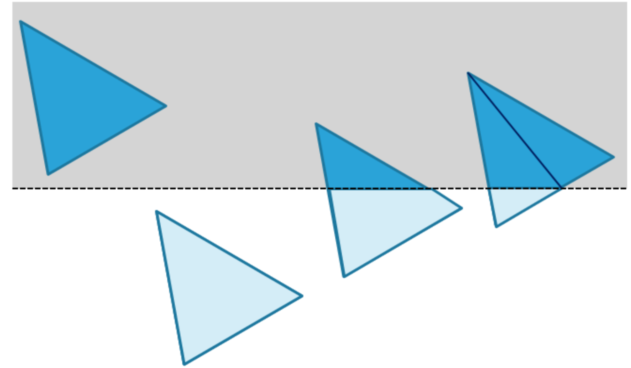
Diese drei Zwischenresultate kann man vermeiden, indem man die Prozedur für die 4 Fenstergrenzen *rekursiv* aufruft, und so jeden Ergebnispunkt sofort wieder als nächsten Eingabepunkt der nächsten Clippingoperation verwertet. Alternativ kann natürlich

auch eine Pipeline durch die 4 Operationen geschleust werden, sodass am Ende nur ein Polygon entsteht – das an dem Fenster korrekt geclippte Polygon.

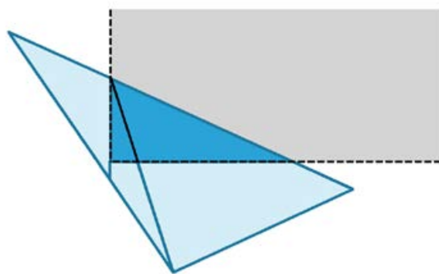


Wenn ein Polygon beim Clippen in mehrere Teile zerfällt, dann erzeugt dieses Verfahren Verbindungskanten entlang des Clippingfensters. Eine nachträgliche Kontrolle und eventuelle Nachbearbeitung ist in solchen Fällen notwendig.

## Clipping von Dreiecken



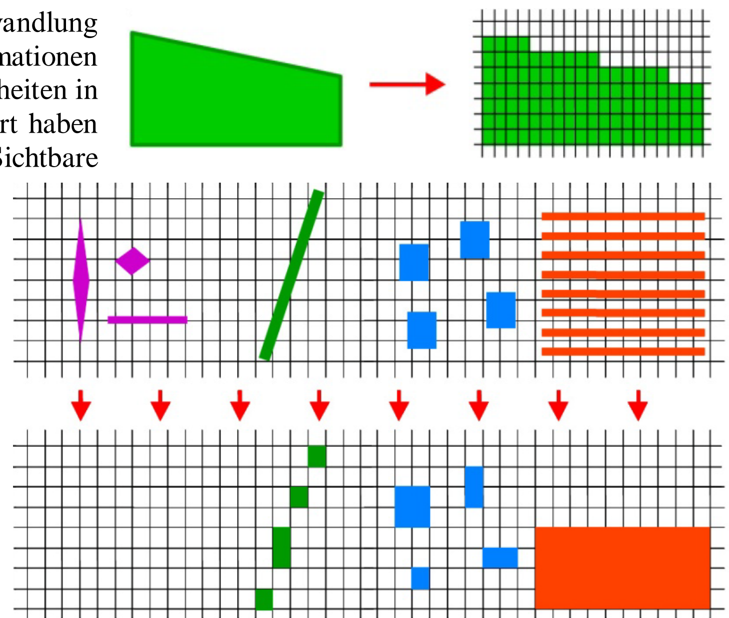
In der Praxis bestehen geometrische Daten sehr häufig nur aus Dreiecken, und der Renderingprozess hat kein Wissen mehr über deren Zusammenhang. Man spricht dann von einer „Triangle Soup", also einer „Dreiecks-Suppe". In diesem Fall ist es wichtig, dass es außer Dreiecken keine anderen Primitive gibt. Daher müssen auch beim Clipping immer wieder Dreiecke entstehen. Beim Clippen eines Dreieckes gegen eine Kante kann in einem von vier möglichen Fällen (siehe Abb. rechts) auch ein Viereck entstehen. Dieses muss sofort in zwei Dreiecke zerteilt werden, um eine Weiterverarbeitung zu gewährleisten. Dabei kann es an den Ecken des Clip-Fensters passieren, dass mehr Dreiecke erzeugt werden als notwendig wäre (z.B. Abb. links), dies wird jedoch durch die Einfachheit des Algorithmus mehr als kompensiert.



## Clipping in Clipkoordinaten

Im Clip-Space sind die Begrenzungsflächen des View-Frustums (also des Bereiches, an dem geclippt werden muss) alle achsenparallel ($x = \pm 1$, $y = \pm 1$, $z = \pm 1$), wodurch sich die Feststellung, ob ein Punkt innerhalb oder außerhalb so einer Fläche liegt, auf einen einzigen Vergleich zweier Zahlen reduziert. Indem man diesen Schritt schon vor dem Homogenisieren der Punktkoordinaten durchführt, indem man an den Ebenen $x = \pm h$, $y = \pm h$, $z = \pm h$ clippt (was genauso einfach ist), stellt man sicher, dass Punkte, die hinter dem Kamerapunkt liegen, nicht projiziert werden (das wäre ja ganz falsch!), und außerdem erspart das die Homogenisierungsdivision für alle Punkte, die außerhalb des Clipbereiches lagen.

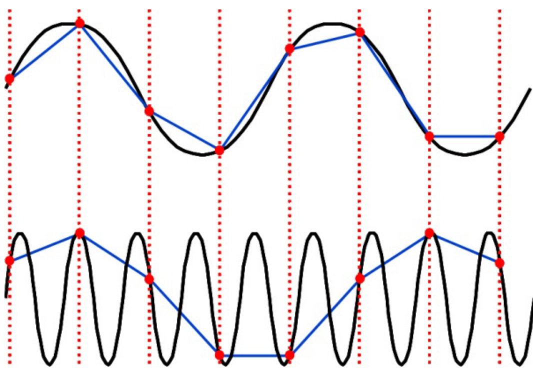
## Aliasing und Antialiasing



*Aliasing*-Effekte ['eiliæsiŋ] sind Fehler, die bei der Umwandlung (Diskretisierung) von analogen in digitale Informationen auftreten. Mit Aliasing bezeichnet man u.a. alle Unschönheiten in Rasterbildern die auftreten, weil ein Pixel nur einen Wert haben kann, aber in Wahrheit eine kleine Fläche repräsentiert. Sichtbare Aliasingeffekte haben z.B. folgende Ursachen: zu geringe Auflösung, zu wenige verfügbare Farben, zu wenige Bilder/sec, geometrische Fehler, numerische Fehler.

*Antialiasing* nennt man Methoden zur Reduktion unerwünschter Aliasing-Artefakte. Da eine Verbesserung der Hardware meist unrealistisch ist, werden hauptsächlich Software-Methoden eingesetzt. Im Folgenden wird nur auf Anti-Aliasing zur Behandlung des Auflösungsproblems eingegangen. Einige bekannte Effekte neben dem Treppeneffekt sind: Verschwinden kleiner Objekte, unterbrochene schmale Objekte,

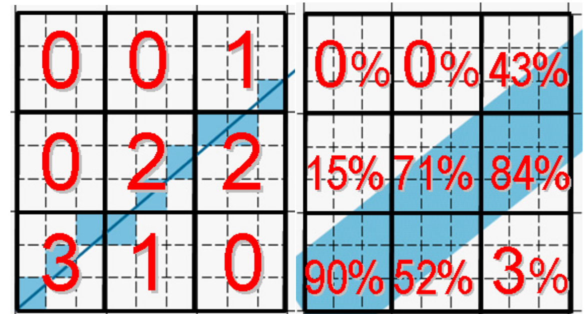
unterschiedliche Größe gleicher Objekte, völlige Zerstörung feiner Texturen (siehe Abbildung).



Die Ursache von Aliasing ist ungenügend feine Abtastung des wahren Bildes. Die theoretische Grundlage dazu bildet das *Nyquist-Shannon-Abtasttheorem*. Demnach kann eine Information nur dann korrekt rekonstruiert werden, wenn eine Abtastfrequenz (sampling rate) verwendet wird, die mindestens doppelt so hoch ist wie die höchste zu übertragende Informationsfrequenz. Diese Grenze heißt *Nyquist-Limit*. Die Abb. zeigt wie eine zu grobe Abtastrate eines Signals (Kurve) zu einer völlig falschen Rekonstruktion (Polygonzug) führen kann. Reduzieren lassen sich solche Fehler entweder durch Vorfilterung des Signals oder durch eine Nachbearbeitung des fertigen Bildes. Eine Nachbearbeitung ist der Vorfilterung aber jedenfalls unterlegen. Die zentrale Strategie beim Vorfiltern ist *Supersampling* (auch *Oversampling*).

## Antialiasing von Linien

Pixel, die von einer Linie weiter durchkreuzt werden, sollen mehr Linienfarbe bekommen, als Pixel, die von einer Linie nur leicht gestreift werden. Dazu unterteilt man jedes Pixel in Subpixel, zählt, wie viele Subpixel auf der Linie liegen, und wählt eine Intensität die proportional zu dieser Anzahl ist. Für breitere Linien berechnet man den Prozentsatz der Überdeckung des Pixels durch die Linie und wählt danach die Intensität der Linienfarbe. Aufbauend auf der Erkenntnis, dass die Mitte eines Pixels wichtiger ist als dessen Rand, werden auch manchmal die Subpixel in der Mitte stärker gewichtet als die am Rand („weighted oversampling“).




## Antialiasing von Polygonkanten

Für Polygonkanten gibt es dieselben Alternativen wie für Linien: entweder man arbeitet mit Supersampling oder man berechnet analytisch den Überdeckungsgrad des Pixels durch das Polygon. Die Berechnung des Überdeckungsgrades erfolgt gleichzeitig mit der Rasterkonversion, also der Erzeugung des Randes und der Füllung eines Polygons. Bei der Berechnung der Endpunkte der Spans beim Scanlinien-Füllverfahren hat man genug Informationen zur Verfügung, dass der Überdeckungsgrad fast gratis abfällt.

Wir erinnern uns an die Entscheidungsvariable $p_k$ beim Bresenham-Linien-Algorithmus, deren Vorzeichen Auskunft gab, welches Pixel als nächstes zu zeichnen war. Diese Variable lässt sich so transformieren, dass ihr Wert dem Überdeckungsgrad des letzten Pixels entspricht. $p' = y - y_{mid}$ , wobei $y_{mid} = (y_k + y_{k+1})/2$, hat die gleiche Vorzeicheneigenschaft wie $p_k$. Verwendet man $p = p' + (1 - m)$ , dann ist zwar der Vergleich nicht mit 0 sondern mit $(1 - m)$ durchzuführen, dafür gilt $0 \leq p \leq 1$, und p entspricht dem Überdeckungsgrad an der Stelle $x_k$. Auf diese Weise kann man das Anti-Aliasing inkrementell sehr rasch berechnen. Für andere Winkel arbeitet man mit Drehungen um 90° und/oder Spiegelungen dieses Verfahrens.