Attribute graphischer Primitive

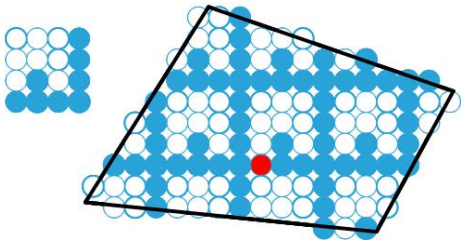
Graphikprimitive können mit vielerlei Eigenschaften erzeugt werden, sogenannten Attributen. Dieses Kapitel inkludiert auch Algorithmen zum Flächenfüllen, einem Attribut von Polygonen.

Attribute von (Punkten und) Linien

Neben allgemein bekannten Eigenschaften von Linien, wie Strichdicke, Strichlierungsmuster, Farbe oder Pinseltyp, gibt es noch ein paar Attribute, die einem of weniger bewusst sind. Dazu gehören etwa die Linienenden bei breiteren Linien sowie die Form von Ecken bei breiten Linien:

Weiters ist Antialiasing auch für Linien ein Thema, dazu werden etwas weiter unten Details gebracht.

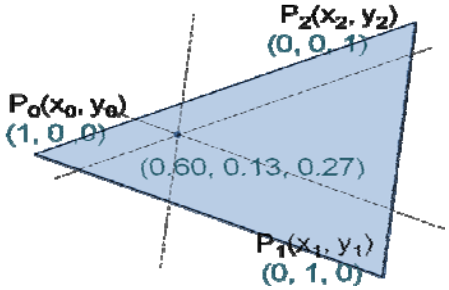
Attribute von (2D-) Polygonen und Flächen



Klarerweise sind die Attribute des Randes von Flächen dieselben wie die von Linien. Dazu kommt nun die Fläche selbst, die mit einer Füllung versehen werden kann. Muster werden dabei gewöhnlich durch repetitive Aneinanderreihung eines Grundmusters ausgehend von einem Referenzpunkt (auch Seed-Point) erzeugt.

In vielen Anwendungen ist es auch notwendig, eine Kombination des neu gezeichneten Musters mit dem Hintergrund zu erzeugen. Hier gibt es viele Varianten, die oft auf logischen Verknüpfungen aufbauen: AND, OR, XOR. Das Mischen von Farben erfolgt meist durch Linearkombination der vorhandenen Hintergrundfarbe B mit der zu zeichnenden Vordergrundfarbe F: $P = tF + (1-t)B$

Dreiecke rasterisieren



Um Dreiecke zu füllen verwendet man oft *baryzentrische Koordinaten*. Jeder Punkt der Ebene wird dabei als gewichtetes Mittel der 3 Eckpunkte des Dreiecks dargestellt: $P=\alpha P_0 + \beta P_1 + \gamma P_2$. (α, β, γ) nennt man dann die baryzentrischen Koordinaten des Punktes P, wobei immer gilt: $\alpha+\beta+\gamma=1$.
Alle Punkte mit ($0 < \alpha, \beta, \gamma < 1$) liegen innerhalb des Dreiecks.
Zum Füllen eines Dreieckes berechnet man für jedes Pixel einer (möglichst engen) Umgebung die baryzentrischen Koordinaten und zeichnet alle mit ($0 < \alpha, \beta, \gamma < 1$). Dabei kann man sehr einfach beliebige Eckpunktwerte in jedem Pixel mit (α, β, γ) gewichtet eintragen, das entspricht einer linearen Interpolation dieser Werte. Wenn $l_{12}(x,y) = a_{12}x+b_{12}y+c_{12}=0$ die Trägergerade durch die Punkte P_1 und P_2 ist, dann berechnet sich α des Punktes P(x,y) zu $\alpha = l_{12}(x,y) / l_{12}(x_0,y_0)$, β und γ analog.
Um das doppelte Zeichnen der Kanten aneinandergrenzender Dreiecke zu vermeiden, werden nur Pixel gezeichnet, die innerhalb eines Dreiecks liegen. Pixel genau auf einer Kante sind speziell zu behandeln.

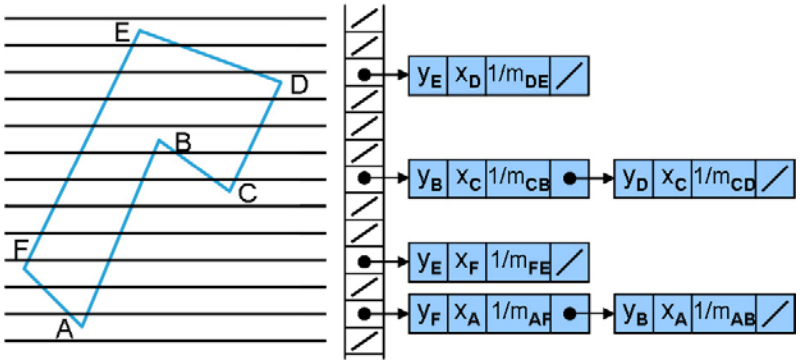
Beim Füllen eines komplexeren Polygons muss klargestellt sein, welche Teile „innen“ sind und welche „außen“. Siehe dazu Textblatt 01. Zum Füllen selbst gibt es zwei Klassen von Verfahren. Bei den Scanline-Verfahren wird jede Scanline mit dem Polygon (oder mit der Flächenumrandung) geschnitten, und innere Teile (Spans) werden gefüllt. Bei den Floodfill-Verfahren wird die Fläche ausgehend von einem Anfangspunkt in alle Richtungen gefüllt, bis man das Ende erkennt.

Scanlinien-Flächenfüllen

Alle Kanten des Polygons werden nach ihrem niedrigeren y-Wert sortiert (Bucketsort). Man speichert für jede Kante:

[max. y-Wert, Anfangs-x-Wert, Steigung]

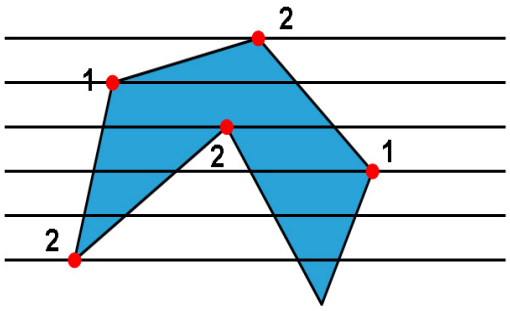
Aus dieser Datenstruktur erzeugt man für jede Scanlinie von unten nach oben eine Liste der „aktiven“ Kanten, das sind jene, die die Scanlinie schneiden. Dann wird einfach die Scanlinie vom 1. zum 2. Schnittpunkt gefüllt, vom 3. zum 4., vom 5. zum 6. usw.



Die aktiven Kanten erzeugt man inkrementell. Am Beginn sind keine Kanten aktiv. Aus der sortierten Kantenliste sieht man für jeden y-Wert, ob dort eine neue Kante beginnt, diese wird zur aktiven Liste hinzugefügt. Gleichzeitig werden alle Kanten, deren maximales y überschritten wurde, entfernt. Die Liste selbst ist immer nach ihren Schnittpunkten von links nach rechts sortiert, sodass das Zeichnen unmitelbar erfolgen kann.

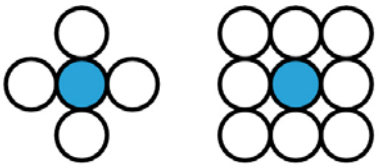
Die Schnittpunkte kann man ebenfalls inkrementell erzeugen. Ausgehend vom (exakten) Schnittpunkt (x_k,y_k) einer Scanlinie mit einer Kante erhält man den nächsthöheren Schnittpunkt (x_{k+1},y_{k+1}) durch $x_{k+1}= x_k+1/m$ und $y_{k+1}= y_k+1$. Man sieht nun, warum der Anstieg 1/m in den Knoten mitgespeichert wird.

Wenn ein Polygoneckpunkt genau auf einer Scanlinie zu liegen kommt, dann muss darauf geachtet werden, dass die Anzahl der Schnittpunkte an dieser Stelle korrekt ist. Manche Punkte müssen dann einfach gezählt werden, andere doppelt (siehe Abbildung). Um diesem Problem aus dem Weg zu gehen werden häufig die Punktkoordinaten um einen kleinen Wert ε nach oben oder unter verschoben. Dieser Wert ist so kleine, dass man es nicht sieht, aber groß genug, dass der Punkt neben der Scanlinie liegt.



Floodfill-Algorithmus

Ausgehend von einem Startpunkt (Referenzpunkt, Saatpunkt) wird in alle Richtungen gefüllt, bis man an eine Grenze stößt. Diese Grenze kann entweder explizit definiert sein, zum Beispiel durch eine Umrandung in einer bestimmten Farbe, oder aber implizit, z.B. dadurch, dass eine bereits gefärbte Fläche umgefärbt wird. Mischvarianten kommen auch vor. Diese Unterscheidung verändert aber lediglich das Abbruchkriterium beim Füllen. Wir wollen oBdA davon ausgehen, dass die zu füllende Fläche in einer „alten“ Farbe bereits gezeichnet ist und umgefärbt werden soll.

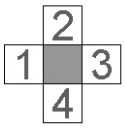


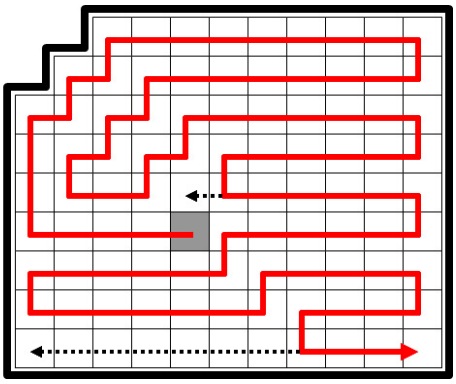
Grundlage für das Weitergehen in alle Richtungen ist die Definition der erlaubten Richtungen. 4-verbunden heißt, dass eine Verbindung nur über die 4 Hauptrichtungen definiert ist, 8-verbunden sieht auch diagonale Pixel als verbunden an. Man kann sich leicht überlegen, dass für eine 4-verbundene Fläche eine 8-verbundene Grenze reicht, eine 8-verbundnee Fläche aber eine 4-verbundene Grenzlinie benötigt.

Floodfill für 4-verbundene Flächen lässt sich ganz leicht rekursiv implementieren:

```
void floodFill4 (int x, int y, int new, int old)
{ int color;
    /* set current color to new */
    getPixel (x, y, color);
    if (color = old) {
        setPixel (x, y);
        floodFill4 (x-1, y, new, old);  /* left */
        floodFill4 (x, y+1, new, old);  /* up */
        floodFill4 (x+1, y, new, old);  /* right */
        floodFill4 (x, y-1, new, old)   /* down */
}}
```

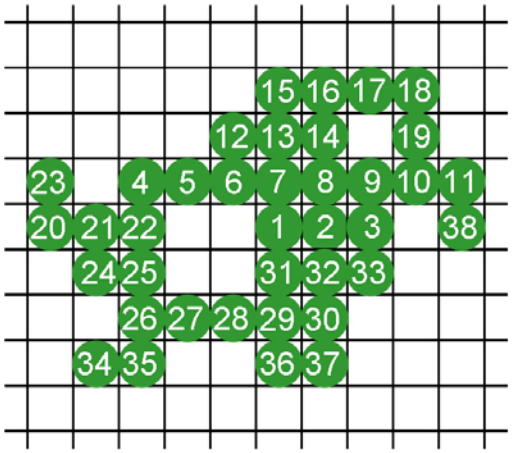
Diese Prozedur erzeugt jedoch eine Füllreihenfolge, die zu einer sehr hohen Rekursionstiefe führt, im allgemeinen Fall bis zur Anzahl der zu füllenden Pixel. In der Abbildung ist links oben die Rekursionsreihenfolge angegeben, und der durchgezogene Pfeil im Polygon zeigt, wie tief die Rekursion in diesem Fall geht.





Um das zu verhindern füllt man meist in horizontaler Richtung iterativ und wendet die Rekursion nur nach oben und unten an. Natürlich muss man dabei aufpassen, dass *alle* Spans oberhalb und unterhalb rekursiv aufgerufen werden (ein Span ist eine horizontale nicht unterbrochene Folge von Pixeln, die gemeinsam behandelt werden).

Das nebenstehende Beispiel demonstriert die Füllreihenfolge, wobei eines der Pixel 1, 2 oder 3 als Startpixel gewählt wurde. Die Rekursion geht zuerst nach oben und dann nach unten. Bei jedem Aufruf müssen alle Spans in diese Richtung erwischt werden. Nach dem Span 4-11 wird nach oben von links nach rechts aufgerufen und danach nach unten ebenfalls von links nach rechts. Teile, die bereits gefüllt wurden, werden erkannt, und die Rekursion bricht dort ab (z.B. von 4-11 nach unten ist 1-3 schon fertig). Die Gesamtrekursionstiefe kann zwar theoretisch auch bei diesem Verfahren sehr hoch sein, in der Praxis ist sie aber proportional der Anzahl der Pixelzeilen (Scanlinien) des Polygons.



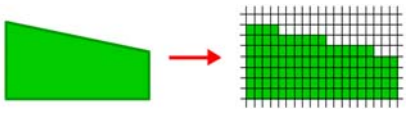
Attribute von Text

Die Eigenschaften, die Text annehmen kann, sind heute weitgehend Allgemeinwissen: Font (z.B. `Courier`, Arial, Times, **Broadway**, …), Stil (normal, **fett**, *kursiv*, unterstrichen, …), Größe, Richtung, Farbe, Bündigkeit (links, rechts, mittig, Blocksatz) und so weiter.

Attribute von 3D Polygonen

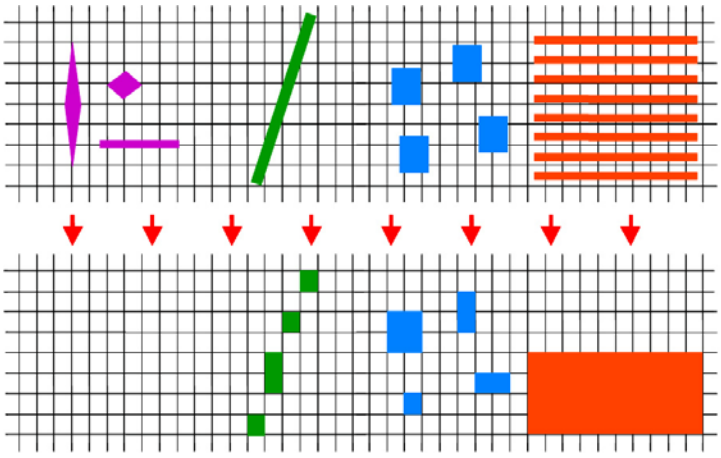
3D Polygone sind normalerweise Oberflächenelemente von Objekten im Raum. Dementsprechend sind deren Attribute auch hauptsächlich Eigenschaften der Oberfläche dieses Objektes: Farbe, Materialparameter (Rauheit, Absorption, …), Transparenz, Textur, geometrische Mikrostruktur, Reflexionsverhalten usw., die in Zusammenspiel mit einer definierten Beleuchtung szenenabhängige Effekte liefern. Zusätzlich werden an den Eckpunkten oft Normalvektoren und Texturkoordinaten angegeben, um eine schnelle und adäquate Berechnung des Aussehens des Polygons zu ermöglichen. Wir werden später sehen, wie man das alles effizient einsetzt.

Aliasing und Anti-Aliasing

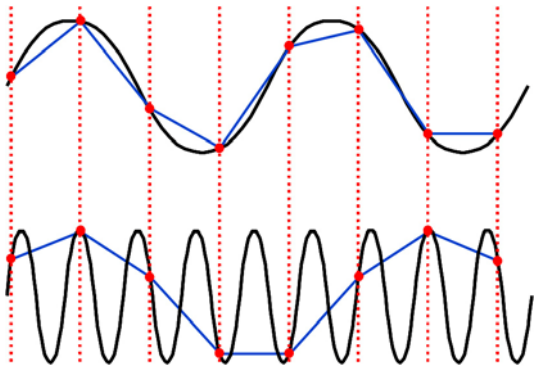


Aliasing-Effekte ['eiliæsiŋ] sind Fehler, die bei der Umwandlung (Diskretisierung) von analogen in digitale Informationen auftreten. Mit Aliasing bezeichnet man u.a. alle Unschönheiten in Rasterbildern die auftreten, weil ein Pixel nur einen Wert haben kann, aber in Wahrheit eine kleine Fläche repräsentiert.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

Anti-Aliasing 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 nebenstehende Abbildung).

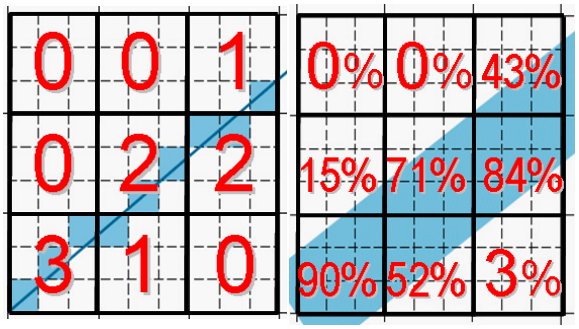


Die Ursache von Aliasing ist ungenügend feine Abtastung des wahren Bildes. Die theoretische Grundlage dazu bildet das Shannon'sche *Abtasttheorem*. Demnach kann eine Information nur dann korrekt rekonstruiert werden, wenn eine Abtastfrequenz (sampling rate) verwendet wird, die mindestens doppelt so hoch 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*).



Anti-Aliasing 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, wieviele 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").



Anti-Aliasing 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 $0 \le p \le 1$, 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.