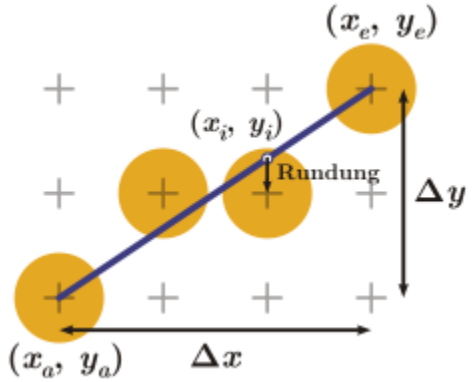
Computergraphik 1 LU (186.095)

Ausarbeitung Abgabe 1

Inhaltsverzeichnis

Theorie - Naive Methode der Rasterung mittels Rundung



Die einfachste Möglichkeit der Rasterung ist die direkte Umsetzung der Gleichung, die die Linie definiert. Wenn (x_a, y_a) der Anfangs- und (x_e, y_e) der Endpunkt der Linie ist, so erfüllen die Punkte auf der Linie die Geradengleichung $y = m(x - x_a) + y_a$, wobei $m = \frac{\Delta y}{\Delta x} = \frac{y_e - y_a}{x_e - x_a}$ die Steigung ist. Die Linie wird gezeichnet, indem in einer Schleife für jedes x von x_a bis x_e der entsprechende y-Wert gemäß dieser Formel berechnet und auf die nächstliegende Ganzzahl gerundet wird. Das Pixel (x, y) wird dann eingefärbt.

Dieses Verfahren ist unnötig langsam, da innerhalb der Schleife eine Multiplikation ausgeführt wird, die auf den meisten Computern wesentlich mehr Rechenzeit als eine Addition oder Subtraktion erfordert. Eine schnellere Methode ergibt sich durch die Betrachtung der Differenz zwischen zwei aufeinanderfolgenden Schritten:

$$\begin{aligned} y_{i+1} - y_i &= (m(x_{i+1} - x_0) + y_0) - (m(x_i - x_0) + y_0) \\ &= m(x_{i+1} - x_i) \\ &= m. \end{aligned}$$

Demnach genügt es, mit (x_a, y_a) zu starten und bei jedem Schleifendurchlauf y um m zu erhöhen. Dieses Verfahren wird auch als Digital Differential Analyzer (DDA) bezeichnet.

Da die Rundung von y zur nächsten Ganzzahl dem Abrunden von $y + 0{,}5$ entspricht, lässt sich auch eine zusätzliche Kontrollvariable verwenden, die mit 0,5 initialisiert wird und zu der bei jedem Schleifendurchlauf m addiert wird. Jedes Mal, wenn die Kontrollvariable den Wert 1,0 erreicht oder übersteigt, wird y um 1 erhöht und von der Kontrollvariable 1,0 abgezogen. Dadurch ist keine Rundung mehr nötig. Diese Methode kann so umformuliert werden, dass sie nur schnellere Ganzzahl-Operationen verwendet und sich elegant in Assemblersprache implementieren lässt.[1] Dennoch ist weiterhin eine langsame Division ($\Delta y / \Delta x$) zu Beginn nötig, die bei kurzen Linien nicht durch die schnelle Schleife aufgewogen werden kann.

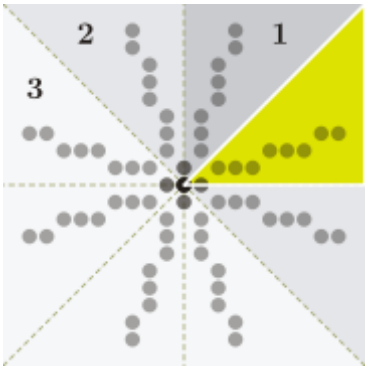
Theorie - Verallgemeinerung auf beliebige Richtungen

Die soeben beschriebenen Verfahren funktionieren nur bei Liniensteigungen zwischen 0 und 1, was einem Winkel von 0° bis 45° zur Horizontalen entspricht. Bei anderen Steigungen wird die Linie nicht oder falsch gezeichnet. Es genügt jedoch, einen Algorithmus nur für Steigungen zwischen 0 und 1 zu beschreiben, da andere Linien durch die Nutzung von Symmetrien korrekt dargestellt werden können.

Dies geschieht durch folgende drei Veränderungen:

Es wird zwischen zwei Fällen unterschieden, je nachdem, ob $\Delta x > \Delta y$ oder umgekehrt. Im ersteren Fall wird die Schleife wie bisher über *x* durchlaufen und im Schleifenrumpf *y* berechnet, ansonsten wird sie über *y* durchlaufen und *x* berechnet.

Innerhalb des Schleifenrumpfs wird *y* bzw. *x* nicht um 1 erhöht, sondern es wird der Vorzeichenwert von Δy bzw. Δx addiert. Falls $\Delta x < 0$ bzw. $\Delta y < 0$, so muss die Schleife rückwärts durchlaufen werden.



Rasterung beliebiger Linien durch Nutzung von Symmetrieeigenschaften. Der zulässige Bereich des Originalverfahrens ist farbig dargestellt, durch die drei Änderungen am Algorithmus werden auch die anderen Bereiche erschlossen.

Theorie - Bresenham-Algorithmus - Grundlage

Der Bresenham-Algorithmus ist ein Algorithmus in der Computergrafik zum Zeichnen (Rastern) von Geraden oder Kreisen auf Rasteranzeigen.

Das Besondere an seinem Algorithmus ist, dass er Rundungsfehler, die durch die Diskretisierung von kontinuierlichen Koordinaten entstehen, minimiert, und gleichzeitig einfach implementierbar ist, mit der Addition von ganzen Zahlen als komplexeste Operation, und somit ohne Multiplikation, Division und Gleitkommazahlen auskommt.

Theorie - Bresenham-Algorithmus – Erklärung 1

Die Idee des Bresenham-Algorithmus besteht darin, bei jedem Schritt zwischen den beiden Pixeln zu wählen, die rechts („östlich“) und rechts oben („nordöstlich“) vom zuletzt gezeichneten Pixel liegen. Es wird dasjenige Pixel gewählt, das näher an der idealen Linie liegt. Dazu betrachtet man in der Midpoint-Formulierung den Mittelpunkt *M* zwischen den Pixeln *O* und *NO*: befindet sich *M* über der idealen Linie, so liegt *O* näher, ansonsten *NO*.

Um die Position von *M* gegenüber der Linie zu bestimmen, wird eine andere Form der Geradengleichung verwendet:

$$F(x, y) = ax + by + c = \Delta y \cdot x - \Delta x \cdot y + c = 0$$

F(*x, y*) ist 0 für Punkte auf der Linie, positiv für Punkte unterhalb und negativ für Punkte oberhalb der Linie. Wenn in diese Gleichung die Koordinaten von *M* eingesetzt werden, so erhält man den Wert

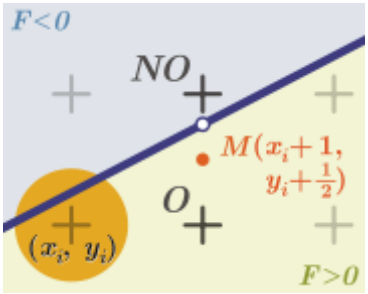
$$d_i = F(x_i + 1,\, y_i + \tfrac{1}{2}) = \Delta y(x_i + 1) - \Delta x(y_i + \tfrac{1}{2}) + c$$

Je nach Vorzeichen dieser Kontrollvariable d_i wird das Pixel *O* oder *NO* gewählt.
Um einen effizienten Algorithmus zu erhalten, wird die Kontrollvariable inkrementell berechnet, also schrittweise erhöht. Ihre Änderung zwischen zwei aufeinanderfolgenden Schritten hängt davon ab,

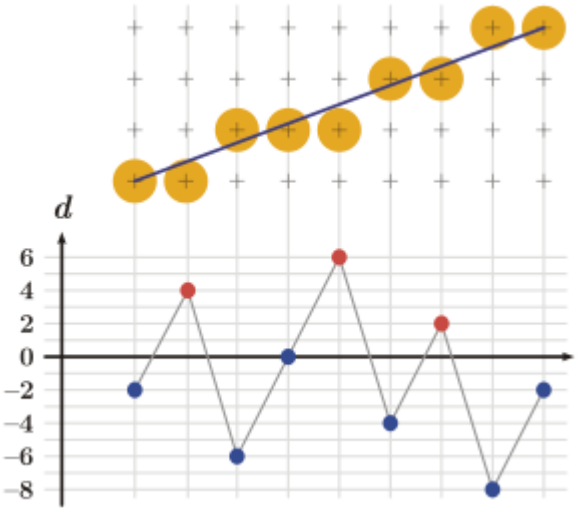
ob Pixel *O* oder *NO* gewählt wurde. Für jeden dieser Fälle betrachtet man die Differenz zwischen dem Wert der Kontrollvariable beim übernächsten und beim nächsten Pixel:

$$\begin{aligned}\Delta_O &= d_{i+1} - d_i = F(x_i + 2,\, y_i + \tfrac{1}{2}) - d_i = \Delta y \\ \Delta_{NO} &= d_{i+1} - d_i = F(x_i + 2,\, y_i + \tfrac{3}{2}) - d_i = \Delta y - \Delta x\end{aligned}$$

Bei jedem Schritt wird die Kontrollvariable je nach gewähltem Pixel um Δ_O oder Δ_{NO} erhöht. Es lässt sich außerdem leicht feststellen, dass der Anfangswert der Kontrollvariable $\Delta y - \Delta x / 2$ beträgt. Um diese Division zu beseitigen, werden alle Werte der Kontrollvariable verdoppelt; das Vorzeichen bleibt dabei erhalten. Damit lässt sich der Bresenham-Algorithmus für Linien mit einer Steigung zwischen 0 und 1 in nachfolgendem Pseudocode ausdrücken. Der Algorithmus benötigt nur Additionen innerhalb der Schleife; die einfachen Multiplikationen außerhalb der Schleife lassen sich ebenfalls durch eine Addition realisieren.



Wahl des nächsten Pixels beim Bresenham-Algorithmus



Veränderung der Kontrollvariable beim Bresenham-Algorithmus

Eine andere Interpretation des Algorithmus geht von der Feststellung aus, dass die gerasterte Linie $m = \Delta x - \Delta y$ Horizontal- und $n = \Delta y$ Diagonalschritte enthält. Um diese beiden Schritttypen zu „mischen“, wird bei jedem Schritt entweder m von der Kontrollvariable abgezogen oder n addiert. Es wird der entsprechende Schritttyp ausgeführt, bei dem der resultierende Betrag der Kontrollvariable geringer ist. Dies wird auch aus obiger Grafik deutlich, bei der die Kontrollvariable stets so nahe wie möglich an der Nullachse liegt.

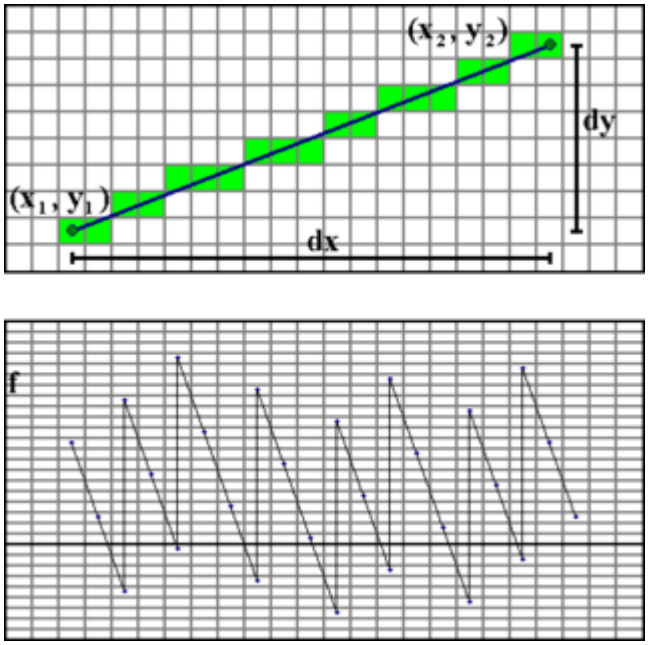
Obwohl der Bresenham-Algorithmus recht effizient ist, zeichnet er nur ein Pixel pro Schleifendurchlauf und benötigt dazu eine Addition.

Theorie - Bresenham-Algorithmus – Erklärung 2

Zum Verständnis des Algorithmus beschränkt man sich auf den ersten Oktanten, also eine Linie mit einer Steigung zwischen 0 und 1 von (xstart,ystart) nach (xend,yend). Seien dx=xend-xstart und dy=yend-ystart mit 0<dy<=dx. Für andere Oktanten muss man später Fallunterscheidungen über Vorzeichen von dx und dy und die Rollenvertauschung von x und y treffen.
Der Algorithmus läuft dann so, dass man in der „schnellen“ Richtung (hier die positive x-Richtung) immer einen Schritt macht und je nach Steigung hin und wieder zusätzlich einen Schritt in der „langsameren“ Richtung (hier y). Man benutzt dabei eine Fehlervariable, die bei einem Schritt in x-Richtung den (kleineren) Wert dy subtrahiert bekommt. Bei Unterschreitung des Nullwerts wird ein y-Schritt fällig und der (größere) Wert dx zur Fehlervariablen addiert. Diese wiederholten „Überkreuz“-Subtraktionen und -Additionen lösen die Division des Steigungsdreiecks in elementarere Rechenschritte auf.

Zusätzlich muss dieses Fehlerglied vorher sinnvoll initialisiert werden. Dazu betrachtet man den Fall von dy=1, wo man gern in der Mitte (nach der Hälfte von dx) den einen y-Schritt hätte. Also initialisiert man mit dx/2. (Ob dabei zu einer ganzen Zahl auf- oder abgerundet wird, spielt kaum eine Rolle.)

Mathematisch gesehen wird die Geradengleichung y=ystart+(x-xstart)*dy/dx aufgelöst in 0=dx*(y-ystart)-dy*(x-xstart) und die Null links durch das Fehlerglied ersetzt. Ein Schritt um 1 in x-Richtung (Variable x) bewirkt eine Verminderung des Fehlerglieds um ein Mal dy. Wenn das Fehlerglied dabei unter Null gerät, wird es durch einen Schritt um 1 in y-Richtung (Variable y) um ein Mal dx erhöht, was nach der Voraussetzung dx>=dy das Fehlerglied auf jeden Fall wieder positiv macht, bzw. mindestens auf Null bringt.
Der originale Ansatz nach Bresenham (s. u. in Literatur) geht übrigens mehr geometrisch vor, wodurch in seinen Iterationsformeln auf beiden Seiten (bis auf das Fehlerglied) ein zusätzlicher Faktor 2 mitgeschleppt wird und auch die Fehlergliedinitialisierung anders hergeleitet wird.



Rastern einer Linie nach dem Bresenham-Verfahren, unten der Verlauf der Fehlervariablen

Praxis – Bresenham - Vorteile

Eine Frage die oft beim Thema Bresenham auftaucht ist „Warum Bresenham und nicht einfach Geradengleichung (y = kx+d)?“.

Ein Vorteil von Bresenham liegt darin, dass man keine Multiplikation hat und somit viel Rechenzeit spart. Ein anderer Vorteil ist, dass man keine floats (Gleitkommazahlen) verwendet die „langsam“ sind, sondern ints (Ganzzahlen), die man auch nur addieren muss. Darüber hinaus kann man, da der Algorithmus sehr einfach ist, es auch billig, schnell und einfach in Hardware umsetzen bzw. durch Assembler Optimierungen in den Prozessorregistern machen.

Außerdem kann man den Bresenham auch auf Kreise und Kurven anwenden.

Praxis – Bresenham - Programmierung

Zuerst **lädt man in CGLine die Anfangs und Endpunkte** ein **sowie die Farbe** und weist sie dementsprechend zu.

In draw (CG!Canvas canvas) ist der eigentlich Bresenham Algorithmus zu implementieren. Zuerst werden die ganzen **Berechnungen** gemacht **die man ständig braucht**. Zum Beispiel in deltaX wird die „Distanz“ zwischen den 2 Punkten in X Richtung gespeichert, und dementsprechend in Y Richtung bei deltaY.

Nun muss man **schauen wie die zu zeichnende Linie verläuft**. Das Problem ist, dass man nur sehr eingeschränkt zeichnen kann (im 0 bis 45° Winkel und nur von links unten nach rechts oben). Der Trick besteht darin möglichst einfach auch andere Winkel zeichnen zu können bzw „rückwärts“.

Zuerst wollen wir wissen ob man **„rückwärts“** zeichnen muss oder nicht. Deswegen überprüft man zuerst **ob der zweite Punkt in X Richtung weiter unten ist bzw in Y Richtung der zweite Punkt weiter links liegt** (und man somit entweder bei X von rechts nach links zeichnen würde bzw in Y Richtung von oben nach unten was in beiden Fällen Probleme machen würde). Wenn dies der Fall ist, muss man die **Richtung ändern** da sonst nicht korrekt gezeichnet werden kann.

Nun muss man aber noch **überprüfen**, ob die zu zeichnende **Gerade auch zwischen 0 und 45° liegt** (davor überprüfte man ja nur die zu zeichnende Richtung, aber nicht den Winkel!). Dies macht man recht einfach in dem man **schaut ob deltaX größer ist als deltaY**. Der Sinn dahinter ist, dass wenn deltaX größer als deltaY ist, und es ja einen rechten Winkel gibt (zwischen deltaX und deltaY), **somit dann der Winkel kleiner als 45° im Anstieg sein muss**. Ist dies nicht der Fall, sprich hat die zu zeichnende Gerade mehr als 45°, muss man das speziell behandeln. Dazu später mehr.

Zuerst schauen wir ob **deltaX < deltaY** ist. Wenn ja, dann haben wir einen Winkel der kleiner 45° ist. Wir können also den **ersten Pixel zeichnen** und machen dann eine **Schleife die wir deltaX Mal durchlaufen**. Somit hat man dann **für jede X Stelle** auf der deltaX Strecke (also zwischen Punkt 1 und Punkt 2) **ein passendes Y berechnet**.

Nun muss man immer **schauen ob der Entscheidungsparameter $\geq$ 0 ist** wie auf Seite 97 bei 4b im Buch beschrieben. **Wenn ja, muss man Y erhöhen, ansonsten erhöht man nur X**. Nicht vergessen darf man, dass man **jedes Mal** auch den **Entscheidungsparameter neu berechnen** muss! Bei $\geq$ 0 ist der neue Entscheidungsparameter Entscheidungsparameter (alt) + 2*deltaY – 2*deltaX, ansonsten nur Entscheidungsparameter (alt) + 2*deltaY. Dann zeichnet man den Punkt und wiederholt es so lange bis man alle Punkte gezeichnet hat.

Ähnlich geht man vor wenn **deltaX <= deltaY** ist. Somit hat man eine Steigung von mehr als 45° und somit ein Problem beim zeichnen. Das erste Pixel kann man aber ohne Probleme ein Mal zeichnen. Da man mehr Y Koordinaten hat als X, berechnet man anhand der Y Koordinate die passende X Koordinate. Deswegen durchlaufen wir nicht wie zuvor die Schleife anhand deltaX, sondern anhand deltaY. Der Anfangs Entscheidunsparameter wird im Gegensatz zu vorhin statt mit 2 * deltaY – deltaX jetzt mit 2*deltaX – deltaY berechnet.

Jetzt fährt man fort wie zuvor bei einer Strecke < 45° und **schaut ob der Entscheidungsparameter ≥ 0 ist** und **wenn ja erhöht man das X, ansonsten das Y**. Somit ist es genau umgekehrt wie im Fall zuvor!

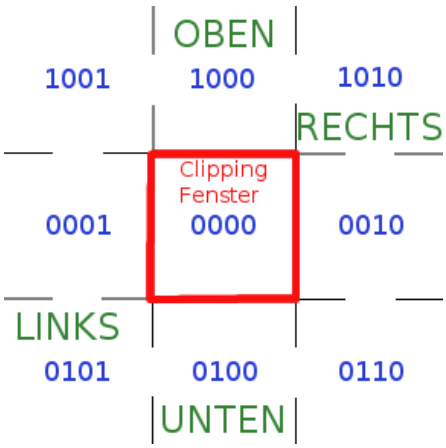
Genauso „umgetauscht“ ist es auch beim neuen Entscheidungsparameter der im ersten Fall Entscheidungsparameter (alt) + 2*deltaX – 2*deltaY ist, im zweiten Fall Entscheidungsparameter (alt) + 2*deltaX ist. Es werden also nur X und Y in der Berechnung „vertauscht“. Die Punkte werden nach und nach gezeichnet, sodass wenn man alle X Koordinaten für jede Y Koordinaten auf der deltaY Strecke berechnet hat, eine schöne Linie mit Hilfe des Bresenham Algorithmus haben sollte.

Theorie - Clipping

In der Computergrafik bezeichnet man mit Clipping (englisch to clip = abschneiden, kappen) die Begrenzung von grafischen Primitiven auf einen bestimmten Bereich. Ein Beispiel ist die Begrenzung einer darzustellenden Linie auf den rechteckigen Bildschirmbereich. Dank Clipping muss nur der tatsächlich den Bildschirmbereich überlappende Teil der Linie gezeichnet werden.

Theorie - Cohen-Sutherland-Algorithmus – Erklärung 1

Der **Cohen-Sutherland-Algorithmus** ist ein Algorithmus zum Clipping von Linien an einem Rechteck. Er eignet sich besonders für Fälle, in denen ein hoher Anteil der zu clippenden Linien außerhalb des Rechtecks liegt.



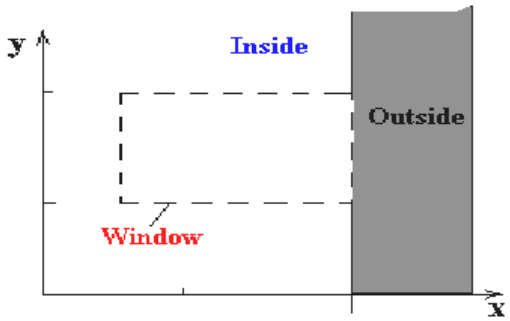
Zuerst werden für die beiden Endpunkte der zu zeichnenden Linie vier Flags ermittelt, die gesetzt werden, wenn sich der Endpunkt links vom Rechteck, rechts davon, darüber oder darunter befindet. Ist keines der Flags gesetzt, dann liegen beide Endpunkte innerhalb des Rechtecks, es ist kein Clipping notwendig und die Linie kann einfach gezeichnet werden.

Wenn dieser triviale Fall nicht eintritt, werden im nächsten Schritt die einander entsprechenden Flags beider Endpunkte angesehen. Ist mindestens eines dieser Flags bei beiden Endpunkten gesetzt, so befindet sich die gesamte Linie außerhalb des Rechtecks und die Linie braucht nicht gezeichnet zu werden.

Wenn auch dieser einfache Fall nicht auftritt, wird der Schnittpunkt einer (beliebigen) Rechteck-Seite mit dem Liniensegment berechnet und der überlappende Teil erneut getestet (und evtl. gekürzt), bis schließlich beide Punkte innerhalb des Rechtecks liegen.

Theorie - Cohen-Sutherland-Algorithmus – Erklärung 2

To determine whether endpoints are inside or outside a window, the algorithm sets up a half-space code for each endpoint. Each edge of the window defines an infinite line that divides the whole space into two half-spaces, the inside half-space and the outside half-space, as shown below.



As you proceed around the window, extending each edge and defining an inside half-space and an outside half-space, nine regions are created - the eight "outside" regions and the one "inside"region. Each of the nine regions associated with the window is assigned a 4-bit code to identify the region. Each bit in the code is set to either a 1(true) or a 0(false). If the region is to the left of the window, the first bit of the code is set to 1. If the region is to the top of the window, the second bit of the code is set to 1. If to the right, the third bit is set, and if to the bottom, the fourth bit is set. The 4 bits in the code then identify each of the nine regions as shown below.

1001	0001	0101
1000	0000 Window	0100
1010	0010	0110

For any endpoint (x , y) of a line, the code can be determined that identifies which region the endpoint lies. The code's bits are set according to the following conditions:

- First bit set **1** : Point lies to **left** of window $x < x_{min}$
- Second bit set **1** : Point lies to **right** of window $x > x_{max}$
- Third bit set **1** : Point lies below(**bottom**) window $y < y_{min}$
- fourth bit set **1** : Point lies above(**top**) window $y > y_{max}$

The sequence for reading the codes' bits is LRBT (Left, Right, Bottom, Top).

Once the codes for each endpoint of a line are determined, the logical AND operation of the codes determines if the line is completely outside of the window. If the logical AND of the endpoint codes is not zero, the line can be trivally rejected. For example, if an endpoint had a code of 1001 while the other endpoint had a code of 1010, the logical AND would be 1000 which indicates the line segment lies outside of the window. On the other hand, if the endpoints had codes of 1001 and 0110, the logical AND would be 0000, and the line could not be trivally rejected.

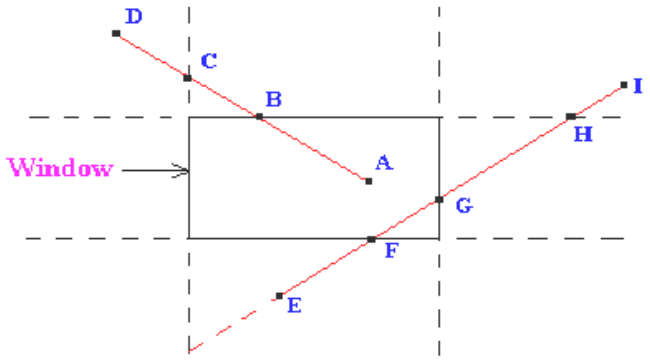
The logical **OR** of the endpoint codes determines if the line is completely inside the window. If the logical OR is **zero**, the line can be trivally accepted. For example, if the endpoint codes are 0000 and 0000, the logical OR is 0000 - the line can be trivally accepted. If the endpoint codes are 0000 and 0110, the logical OR is 0110 and the line can not be trivally accepted.

The Cohen-Sutherland algorithm uses a divide-and-conquer strategy. The line segment's endpoints are tested to see if the line can be trivally accepted or rejected. If the line cannot be trivally accepted or rejected, an intersection of the line with a window edge is determined and the trivial reject/accept test is repeated. This process is continued until the line is accepted.

To perform the trivial acceptance and rejection tests, we extend the edges of the window to divide the plane of the window into the nine regions. Each end point of the line segment is then assigned the code of the region in which it lies.

1. Given a line segment with endpoint $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$
2. Compute the 4-bit codes for each endpoint. If both codes are **0000**,(bitwise OR of the codes yields 0000) line lies completely **inside** the window: pass the endpoints to the draw routine. If both codes have a 1 in the same bit position (bitwise AND of the codes is **not** 0000), the line lies **outside** the window. It can be trivially rejected.
3. If a line cannot be trivially accepted or rejected, at least one of the two endpoints must lie outside the window and the line segment crosses a window edge. This line must be **clipped** at the window edge before being passed to the drawing routine.
4. Examine one of the endpoints, say $P_1 = (x_1, y_1)$. Read P_1's 4-bit code in order: Left-to-Right, Bottom-to-Top.
5. When a set bit (1) is found, compute the intersection I of the corresponding window edge with the line from P_1 to P_2. Replace P_1 with I and repeat the algorithm.

Before Clipping



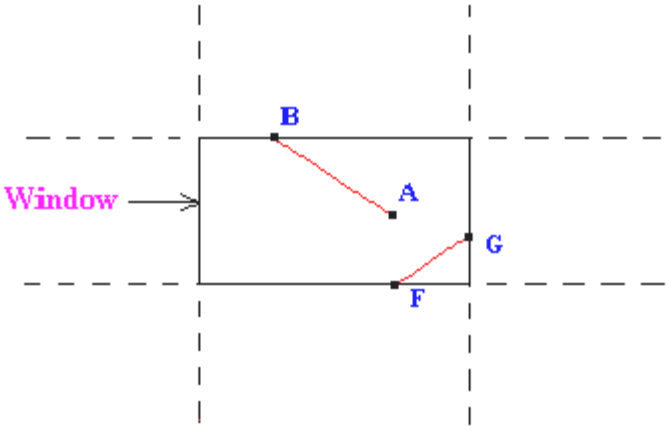
1. Consider the line segment **AD**.
   Point **A** has an outcode of **0000** and point **D** has an outcode of **1001**. The logical AND of these outcodes is zero; therefore, the line cannot be trivally rejected. Also, the logical OR of the outcodes is not zero; therefore, the line cannot be trivally accepted. The algorithm then chooses **D** as the outside point (its outcode contains 1's). By our testing order, we first use the top edge to clip **AD** at **B**. The algorithm then recomputes **B**'s outcode as **0000**. With the next iteration of the algorithm, **AB** is tested and is trivially accepted and displayed.

2. Consider the line segment **EI**
   Point **E** has an outcode of **0100**, while point **I**'s outcode is **1010**. The results of the trivial tests show that the line can neither be trivally rejected or accepted. Point **E** is determined to be an outside point, so the algorithm clips the line against the bottom edge of the window. Now

line **EI** has been clipped to be line **FI**. Line **FI** is tested and cannot be trivially accepted or rejected. Point **F** has an outcode of **0000**, so the algorithm chooses point **I** as an outside point since its outcode is**1010**. The line **FI** is clipped against the window's top edge, yielding a new line **FH**. Line **FH** cannot be trivally accepted or rejected. Since **H**'s outcode is **0010**, the next iteration of the algorthm clips against the window's right edge, yielding line **FG**. The next iteration of the algorithm tests **FG**, and it is trivially accepted and display.

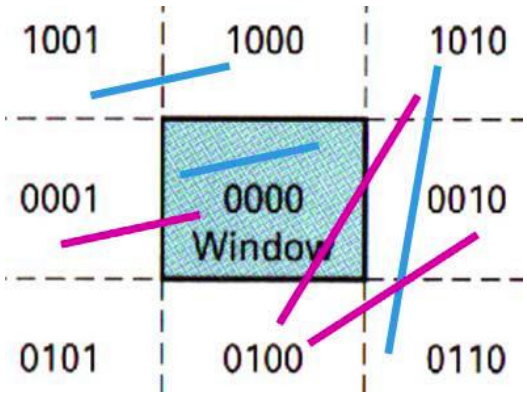
After Clipping

After clipping the segments **AD** and **EI**, the result is that only the line segment **AB** and **FG** can be seen in the window.



Theorie - Cohen-Sutherland-Algorithmus – aus „Skript“

Algorithman 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.



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 = y0 + m(xwmin – x0), y = y0 + m(xwmax – x0)
mit horizontalen Fensterkanten: x = x0 + (ywmin – y0)/m, x = x0 + (ywmax – y0)/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 jedoch berücksichtigen, dass Kreise beim Clippen in mehrere Teile zerfallen können.

Praxis – Clipping -Vorteile

Eine Frage die oft beim Thema Clipping auftaucht ist „ Ist Clipping performant?“.

Ich würde sagen das Clipping performant ist, da es durch die rekursiven Aufrufe keine Zwischenergebnisse erzeugt bzw. speichern muss. Darüber hinaus werden bevor intersectionPoints berechnet werden einige Tests gemacht, sodass man unnötige intersectionPoints Berechnungen erspart.

Ein „Feature Nachteil“ ist aber, dass man ausschließlich konvexe Polygone verwenden kann. Man kann es auch auf konkave Polygone anwenden muss aber Änderungen durchführen. Dann werden z.B. zusätzliche Kanten erzeugt, zerlegt das Polygon in mehrere konvexe Polygone, oder überprüft ob es mehr als 2 intersectionPoints auf einer Canvas Kante gibt und überprüft diese dann genauer.

Praxis - Clipping

Jede Kante rund um das Canvas wird als unendliche Linie angesehen (und deswegen auch nur über die x Minimum/ Maximum ODER y Minimum/ Maximum Koordinaten erstellt). Es wird nun geprüft ob der zu überprüfenden Punkt sich links/ rechts bzw. oberhalb/ unterhalb dieser Kante befindet. Daraus ergibt sich nun ob sich der Punkt „drinnen“ oder „draußen“ befindet, sprich ob man sich im Canvas befindet oder außerhalb.

Bei **inside (CG1Point point, int edge)** macht er genau das. Er schaut dort anhand der unendlichen Kanten ob sich der Punkt links/ rechts bzw. oberhalb/ unterhalb dieser Kante befindet. inside gibt also an ob der Punkt im Canvas liegt (inside == true) oder nicht (inside == false).

Im Fall **LEFT** schaut er zum Beispiel ob der **Punkt anhand der X Koordinate links vom Canvas** (X<wMin.x) **liegt und gibt false als return zurück, ansonsten true.** Also wenn der Punk links vom Canvas ist, liegt er nicht drinnen und inside ist somit false.

Bei **cross (CG1Point p1, CG1Point p2, int edge)** wird geschaut ob es zu einer „Überkreuzung“ einer Canvas Kanten kommt. Wenn ja, heißt das so viel wie dass ein Punkt im Canvas liegt und der andere nicht. Somit muss berechnet werden, wo genau die Gerade zwischen den 2 Punkten im Canvas zu sehen ist und wo nicht. Der intersectionPoint wird in intersect berechnet und ist der Punkt auf der Canvas „Grenze/ Kante“ der zwischen den beiden Punkten liegt die man untersucht, wobei einer außerhalb des Canvas liegt. Sprich ein Punkt ist außerhalb des Canvas, ist über eine Gerade mit einem anderen Punkt verbunden, und an der Stelle wo die Gerade die Canvas „Grenze“ schneidet, und somit ab da die Gerade dann sichtbar ist, liegt der intersectionPoint.

Bei **CG1Point intersect** wird dieser intersectionPoint berechnet. Dazu muss man sich ein Mal die Steigung zwischen den beiden Punkten ausrechnen, da man ansonsten es ziemlich schwer hätte den „Eintrittspunkt“ (intersectionPoint) in den Canvas zu berechnen. Dazu wird zuerst geschaut ob die beiden zu untersuchenden Punkte eh nicht gleiche Koordinaten haben, da man ansonsten eine Division durch 0 hat [(Punkt1Y-Punkt2Y)/(Punkt1X-Punk2X)]. Anschließend wird die Steigung berechnet (Punkt1Y-Punkt2Y)/(Punkt1X-Punk2X).

Jetzt wird der intersectionPoint berechnet indem man zuerst schaut an welcher Kante der Schnitt ist und eine „Fallunterscheidung“ macht. Wenn man den Fall LEFT bzw RIGHT hat weiß man das X schon ein Mal fix, da es genau auf der Canvasgrenze liegen muss und somit wMin.x i Fall LEFT ist bzw. im Fall RIGHT wMax.x entspricht. Man muss sich also nur das Y ausrechnen. Das erhält man indem man m*(wMin.x –Punkt2.x) + Punkt2.y rechnet bzw im Fall RIGHT m*(wMax.x –Punkt2.x) + Punkt2.y. Man macht also nichts anderes als sich die Differenz in X Richtung aus zu rechnen, diese mit der Steigung zu multiplizieren und anschließen die Y Koordinate des zweiten Punktes dazu zu zählen.

Bei den Fällen TOP und BOTTOM verfährt man ähnlich nur das man dort durch die Steigung dividiert!

clipPoint verwendet einige Methoden die vorher beschrieben wurden zum clippen (zum Beispiel cross, intersect,..). Zuerst wird über if (first[edge]== null) geschaut ob es schon einen Vorgängerpunkt gibt (in first wird der letzte Punkt mit dem man gearbeitet hat gespeichert) und wenn nein, wird der aktuelle Punkt verwendet. Ansonsten landet man in der else Schleife (hat also schon einen Vorgängerpunkt in first gespeichert) und schaut dort ob es zu einer Überkreuzung einer Kante kommt (indem man dies über cross prüft). Liefert es true zurück, sprich kommt es zu einer „Überquerung“ einer Canvas Grenze, geht man in die Schleife hinein und berechnet dort den intersectionPoint. Dazu ruft man einfach intersect auf (mit point, s[edge] und edge). Somit weiß man jetzt wo der „Eintrittspunkt“ der Strecke im Canvas liegt.

Anschließend muss man noch den clipPoint berechnen (weiß im Moment nicht wofür der genau noch mal da war ☹). Dazu schaut man ob man alle Canvaskanten überprüft hat via if (edge < TOP) und wenn man in die Schleife hinein geht, also NICHT alle Kanten überprüft hat (da es noch Kanten zum überprüfen gibt), dann berechnet man den clipPoint mithilfe des intersectionPoint und der nächsten edge (edge+1). Das macht man so lange bis man alle Kanten durch hat.

Ansonsten speichert man in clipped [cnt] [X] bzw. clipped [cnt] [Y] den intersectionPoint und rundet (+0.5). Da man als Datentyp int hat, sprich ganze positive und negative Zahlen hat, muss man dies beachten! Int ist deswegen in dem Fall sehr sinnvoll, da man am Monitor nur „ganze“ Pixel zeichnen kann und nicht zum Beispiel 0,23453123424 Pixel. Das Problem ist aber, das bei der Berechnung auch Ergebnisse ungleich ganzer Zahlen rauskommen können (und auch oft werden). Bei int wird deswegen automatisch auf ganze Zahlen gerundet, was nett ist und „arbeit“ erspart, hat aber den Haken, dass er meistens nicht so rundet wie man will. Deswegen „muss“ man 0,5 dazu zählen, damit er auch korrekt rundet (ich habe darauf vergessen, aber im Canvas kann man diesen Fehler nicht wirklich sehen).

Wenn man aber beim Aufruf von clipPoint, wie zuvor schon beschrieben, über if (first[edge]== null) geschaut hat ob es schon einen Vorgängerpunkt gibt und wenn das nicht der Fall ist, kommt man in die zuvor beschrieben Schleife (da es ja else ist) nicht hinein. Das hat den einfachen Grund, dass wenn man noch keinen Punkt untersucht hat, und man nur einen Punkt hat, man logischerweise keine Strecke „aufbauen“ kann, und folglich man auch keinen intersectionPoint berechnen kann.

Deswegen überprüft man nun ob der Punkt im Canvas ist und speichert ihn dann wie zuvor über clipped [cnt] [X] bzw. clipped [cnt] [Y] ab. Nur das man dieses Mal nicht einen intersectionPoint

berechnen muss und via intersectionPoint.x bzw. intersectionPoint.x speichert, sondern da es ein normaler Punkt ist über point.x bzw. point.x.

closeClip () arbeitet sehr ähnlich wie clipPoint (nur das man keinen Punkt „mitgibt“ beim Methodenaufruf). Man überprüft wieder, wie zuvor bei clipPoint, alle Kanten und beginnt bei LEFT und durchläuft die for schleife bis man bei TOP, also der letzten Kante, erreicht hat.

Nun überprüft man via cross (s[edge], first[edge], edge) ob es zu einer Überquerung einer Canvaskante kommt und wenn ja, muss man erneut einen intersectionPoint berechnen. Man durchläuft nun alle Kanten und wenn man fertig ist, also in der else Schleife landet, dann speichert man wieder die Punkte. Nicht aufs Runden dort vergessen (wie zuvor beschrieben +0.5)!

In clip() findet die „Arbeitsverteilung“ statt. Zuerst wird geschaut wie groß das Canvas ist, und in wMin und wMax gespeichert. Konkret macht man das bei wMin indem mach (1,1) zuweist, und bei wMax es sich ausrechnet wie groß das Canvas ist. X rechnet man sich aus indem man über canvas.getWidth() die Canvas Breite holt, und dann – 2 rechnet. In Y Richtung macht man das selbe, nur mit canvas.getHeigth() und dann ebenfalls – 2. Das -2 hat glaube ich damit zu tun, dass man nicht bei 0, 0 beginnt, sondern bei 1,1, und auch am anderen Ende des Canvas 1 Pixel weniger hat da man ja dann schon darüber hinaus ist.

Anschließend „füllt“ man in einer for Schleife die first und s mit null, da man ja oft im Programm überprüft über „ist das was ich untersuche null“ ob man schon was gearbeitet hat oder erst beginnt. Dann wird ein Counter (bei mir cnt) auf 0 gesetzt da über ihn in clip, closeClip und clipPoint viel gemacht wird. Da wir ja viele Punkte haben, kommt nun eine fo Schleife die alle numVertex durch rennt und jeweils ein clipPoint ausführt (dabei vertices[i] übergibt und die Canvaskante LEFT da immer mit LEFT gestartet wird (LEFT=> RIGHT=> BOTTOM, und zum Abschluss TOP)). numVertex sind die Ursprünglichen Eckpunkte eines Polygons und deswegen überprüft man eben auch alle durch.

Am Schluss wird noch closeClip () aufgerufen um das ganze „Polygon“ zu schließen, da man ja IntersectionPoint für alle „Kanten“ berechnet hat, aber es ja auch eine zwischen dem ersten und dem letzten Punkt geben kann.

Praxis – Clipping – Wo ist im Buch in der closeclip Methode der Bug?

Falls ein Polygon komplett außerhalb des Canvas liegt, wird niemals ein letzter Vertex (im Programm wird der Vertex in „s“ gespeichert) definiert. Wenn man nun also closeClip auruft, erhält man beim Vergleich zwischen dem ersten und letzten Vertex eine NullPointerException.

Praxis - Warum gibt es 3x3 und 4x4 Matrizen?

Die 3x3 Matrizen verwendet man normalerweise für den 2D Raum, während man die 4x4 Matrizen für den 3D Raum verwendet.

In unserem Fall wird die 3x3 Matrix aber für die 3D Transformationen von Normalvektoren verwendet. Das erkennt man z.B. auch daran das setScale 3 Koordinaten hat.

Aus Forum: „für die normalvektoren sind translationen nicht notwendig, deshalb wäre die 4 zeile/spalte auch überflüssig.“

Praxis - Was ist die 4. Dimension der 4x4 Matrizen?

Die vierte Dimension bei 4x4 Matrizen wird benötigt, um Transformationen mit Matrixmultiplikationen durchführen zu können.

Praxis - Matrizen Addition

In diesem Fall ist die Matrizenaddition assoziativ, kommutativ und besitzt mit der Nullmatrix ein neutrales Element. Im Allgemeinen besitzt die Matrizenaddition diese Eigenschaften jedoch nur, wenn die Einträge Elemente einer algebraischen Struktur sind, die diese Eigenschaften hat.

Praxis - Matrizen Multiplikation

Zu beachten ist, dass Matrizenmultiplikation im Allgemeinen nicht kommutativ ist, d. h. im Allgemeinen gilt $B \cdot A \neq A \cdot B$. Die Matrizenmultiplikation ist aber immer assoziativ:
$(A \cdot B) \cdot C = A \cdot (B \cdot C)$

Praxis - Matrizen Multiplikation

Wie man schon anhand der Datei CG1Matrix4x4.java/ CG1Matrix3x3.java und den Methode CG1Matrix4x4 multiLeft/ CG1Matrix4x4 multiRight bzw. CG1Matrix3x3 multiLeft/ CG1Matrix3x3 multiRight erkenne kann, ist es NICHT egal ob bei einer Matrizen Multiplikation eine Matrize links oder rechts steht.

Wie schon unter „Theorie - Matrizen Multiplikation“ geschrieben ist die **Matrizenmultiplikation NICHT kommutativ und somit gilt nicht AxB = BxA!** Deswegen muss man auch 2 unterschiedliche Methoden machen damit man auf korrekte Ergebnisse kommt.

Warum links und rechts Multiplikationen? Was ist der Unterschied?

Die links Multiplikation wird mit NeueMatrix * BasisMatrix berechnet
Die rechts Multiplikation wird mit BasisMatrix * NeueMatrix berechnet

Wie schon zuvor geschrieben sind Matrizenmultiplikation nicht kommutativ und somit gilt nicht AxB = BxA. Somit kommt man auch bei beiden Berechnungen auf unterschiedliche Ergebnisse!

Der Unterschied liegt darin, dass bei Linksmultiplikation die Rotation sich auf das Weltkoordinatensystem auswirkt und bei Rechtsmultiplikation. auf die Objektkoordinaten.

Praxis – Dot Product vs. Cross Product

Das **Dot Product**, auf Deutsch **Skalarprodukt** (auch inneres Produkt) gennant, wird oft verwendet um **Winkel zwischen zwei Vektoren und die Länge von Vektoren zu bestimmen**. Man berechnet es durch komponentenweises Multiplizieren der Koordinaten der Vektoren und anschließendes Aufsummieren.

$$\begin{pmatrix}1\\2\\3\end{pmatrix} \cdot \begin{pmatrix}-7\\8\\9\end{pmatrix} = 1 \cdot (-7) + 2 \cdot 8 + 3 \cdot 9 = 36$$

Cross Product, auf Deutsch **Kreuzprodukt** (auch äußeres Produkt) gennant, entspricht einem **Vektor, der senkrecht auf der von den beiden Vektoren aufgespannten Ebene steht.** Die Länge dieses Vektors entspricht dem Betrage nach der Fläche des Parallelogramms mit den Seiten $\vec{a}$ und $\vec{b}$.

$$\begin{pmatrix}1\\2\\3\end{pmatrix} \times \begin{pmatrix}-7\\8\\9\end{pmatrix} = \begin{pmatrix}2\cdot 9 - 3\cdot 8\\3\cdot(-7) - 1\cdot 9\\1\cdot 8 - 2\cdot(-7)\end{pmatrix} = \begin{pmatrix}-6\\-30\\22\end{pmatrix}$$

Info am Rande: Kreuz- und Skalarprodukt sind über das Spatprodukt miteinander verknüpft.

Was ist Uniform Scaling?

Unter Uniform Scaling versteht man eine lineare Transformation, die gleichzeitig und gleichstark alle Achsen (im 3D Raum also x, y und z) skaliert. Somit kann man z.B. leicht ein Objekt in allen Achsen um den Faktor 3 größer machen.

Wofür gibt es die homogene Koordinate?

Die homogenen Koordinaten dienen als Skalierungsfaktor für x, y und z.

Aus „Skript“: Damit auch die Translation in Matrixschreibweise angegeben werden kann, verwendet man *homogene Koordinaten*. Jedem Punkt wird eine zusätzliche Koordinate h zugeordnet, wobei die Umrechnung in 2DKoordinaten durch Division der x- und y-Komponente durch h erfolgt. Daher verwendet man meist h=1. Für den Punkt (x,y) schreiben wir daher (x,y,1). Die Transformationsmatrizen werden um eine Zeile und Spalte mit Einheitswerten erweitert.

$$\begin{pmatrix}x'\\y'\\1\end{pmatrix} = \begin{pmatrix}\cos\theta & -\sin\theta & 0\\ \sin\theta & \cos\theta & 0\\ 0 & 0 & 1\end{pmatrix} \cdot \begin{pmatrix}x\\y\\1\end{pmatrix}$$

2D-Rotation

$$\begin{pmatrix}x'\\y'\\1\end{pmatrix} = \begin{pmatrix}s_x & 0 & 0\\ 0 & s_y & 0\\ 0 & 0 & 1\end{pmatrix} \cdot \begin{pmatrix}x\\y\\1\end{pmatrix}$$

2D-Skalierung

$$\begin{pmatrix}x'\\y'\\1\end{pmatrix} = \begin{pmatrix}1 & 0 & t_x\\ 0 & 1 & t_y\\ 0 & 0 & 1\end{pmatrix} \cdot \begin{pmatrix}x\\y\\1\end{pmatrix}$$

2D-Translation

Warum ist es vorteilhaft, alle Transformationen in einheitlicher Matrixschreibweise zu formulieren? Meist werden größere Teile (Objekte, Bilder) als Ganzes transformiert, d.h. auf jeden Punkt dieser Gebilde wird. die gleiche Folge von Transformationen angewendet. Dies entspricht einer sequenziellen Multiplikation eines Punktes P mit Matrizen M1, M2, M3… : P´ = M1·P, P´´ = M2·P´, P´´´ = M3·P´´, … .Nun kann man sich die Assoziativität der Matrizenmultiplikation [also (M1·M2)·M3 = M1·(M2·M3)] zunutze machen und den Rechenaufwand damit massiv reduzieren.

Unterschiede Perspektivische und Parallele Projektion

Bei der Parallelprojektion findet keine Entfernungsverzerrung statt, somit sind weiter weg liegende Objekte gleich groß wie vorne liegende (gleicher Größe).

Bei der perspektivischen Projektion hat man eine Entfernungsverzerrung. Dadurch können Objekte die im Hintergrund sind, viel größer sind als Objekte die im Vordergrund sind, viel kleiner sein/ wirken.

Bei der Parallelprojektion werden Bildpunkt eines beliebigen Punktes im Raum gefunden, indem man die Parallele zur Projektionsrichtung durch diesen Punkt mit der Projektionsebene zum Schnitt bringt.

Geraden werden durch eine Parallelprojektion im Allgemeinen wieder auf Geraden abgebildet. Parallelen zur Projektionsrichtung werden zu Punkte. Die Länge einer Strecke bleibt nur dann erhalten, wenn diese parallel zur Projektionsebene verläuft. In allen anderen Fällen erscheinen Strecken in der Projektion verkürzt. Auch die Größe eines projizierten Winkels stimmt normalerweise nicht mit der Größe des ursprünglichen Winkels überein. Aus diesem Grund wird ein Rechteck im Allgemeinen auf ein Parallelogramm abgebildet, aber nur in Ausnahmefällen auf ein Rechteck. Ähnliches gilt für Kreise, die im Allgemeinen in Ellipsen übergehen.

Bei der perspektivischen Projektion wird ein Punkt in der Ferne fixiert, und die dreidimensionale Wirkung ergibt sich dadurch, dass weiter entfernte Punkte in ihren X- und Y- Koordinatenanteilen näher an diesen *Fluchtpunkt* heranrücken.

Praxis – 2D Viewing Pipeline

Man hat einen Viewport durch den man eine „Szene“ sieht. Dieser Viewport entspricht einem Canvas das man „bewegen“ kann um auch andere Teile der „Szene“ zu sehen. Die Größe des Viewports könnte man auch ändern. Deswegen muss man die Weltkoordinaten in Viewport Koordinaten umwandeln, Objekte die außerhalb des Viewports liegen korrekt clippen (z.B. mithilfe des Sutherland-Hodgman-Algorithmus), sodass man innerhalb des Viewports korrekte Kanten hat.

Aus „Skript“‘: Die Koordinaten, in denen einzelne Objekte konstruiert werden, nennt man **Modellkoordinaten**. Aus diesen Objekten werden Szenen zusammengestellt, diese befinden sich in **Weltkoordinaten**. Diese nennt man nach der Transformation in das Kamerakoordinatensystem die **Viewingkoordinaten**. Die Abbildung eines Fensters der Szene (bei uns sind zum Beispiel die „Screen Coordinates“ zwischen [-1,…. 1 und -1,…. 1]) erfolgt in geräteunabhängiger Weise in **normalisierte Koordinaten**. Schließlich werden diese normalisierten Werte in **gerätespezifische Gerätekoordinaten** abgebildet (zum Beispiel könnte das Canvas 200x300 Pixel haben).

Praxis – 3D Viewing Pipeline

Aus „Skript“: Die Viewing-Pipeline im 3-Dimensionalen ist mit der 2D-Viewing-Pipeline fast ident. Lediglich nach der Definition der Blickrichtung und Orientierung (also der Kamera) erfolgt noch ein zusätzlicher Projektions- Schritt, also die Reduktion der 3D-Daten auf eine Projektionsebene:

Dieser Projektionsschritt kann beliebig aufwändig sein, je nachdem welche der 3D-Viewing-Konzepte dabei einfließen sollen.

<u>Viewing-Koordinaten:</u>
Ähnlich wie beim Fotografieren hat man beim Festlegen der Kamerawerte mehrere Freiheitsgrade:

1. Position der Kamera im Raum
2. Blickrichtung von dieser Position aus
3. Orientierung der Kamera (wo ist oben?)
4. Größe des Bildausschnittes (entspricht der Brennweite bei einem Fotoapparat)

Mit diesen Parametern legt man das *Kamerakoordinatensystem* fest (Viewing-Koordinaten). Normalerweise ist die xy-Ebene dieses Viewing-Koordinatensystems normal auf die Hauptblickrichtung, und man *blickt in die Richtung der negativen z-Achse*.

Infos

Bei einigen Sachen war ich mir nicht ganz sicher ob sie so stimmen. Deswegen habe ich diese Sätze rot geschrieben (zum leichten erkennen das man da „aufpassen“ sollte, da es womöglich nicht stimmt).

Die Teile mit der Überschrift „Theorie“ habe ich aus dem Internet kopiert und teilweiße gekürzt da oft nicht alles brauchbar war. Diese Abschnitte sollten einem ein Hintergrundverständnis für den praktischen Teil geben, sodass man dann weiß um was es eigentlich geht und wie alles funktionieren sollte.

Abschnitte in denen „Praxis“ in der Überschrift steht, wurden von mir geschrieben und auf der LU ausgerichtet. Sie sollen einem helfen beim Programmieren, wie man wo was warum machen könnte,…

Am Ende sind auch Fragen die gerne beim Abgabegespräch gestellt werden mit „Praxis“ bezeichnet. Somit kann man gleich alle Hintergrundinformationen mit „Theorie“ finden und die für die LU ausschließlich relevanten Informationen mit „Praxis“ in der Überschrift.

Da mir leider nicht bewusst war, dass „Cohen-Sutherland-Algorithmus“ ≠ „Sutherland-Hodgman-Algorithmus“ ist, sollte man sich den „Sutherland-Hodgman-Algorithmus“ lieber noch ein Mal im Buch (Seite 331) anschauen, auch wenn er dem „Cohen-Sutherland-Algorithmus“ stark ähnelt.

Da ich lieber die wichtigen Sachen jetzt ausarbeite, lasse ich den „Cohen-Sutherland-Algorithmus“ so stehen wie er ist und werde ihn durch den „Sutherland-Hodgman-Algorithmus“ ersetzen wenn ich alles andere ausgearbeitet habe da das im Moment Vorrang hat!

Ich habe die Ausarbeitung so gut es geht gemacht, aber trotzdem können sich Fehler einschleichen! Falls man welche findet, bitte per E- Mail oder PM an mich weiter leiten damit ich sie ausbessere!

Quellen und weiterführende Links:

- http://de.wikipedia.org/wiki/Rastern_von_Linien
- http://de.wikipedia.org/wiki/Bresenham-Algorithmus
- http://de.wikipedia.org/wiki/Matrix_%28Mathematik%29
- http://de.wikipedia.org/wiki/Cohen-Sutherland-Algorithmus
- http://de.wikipedia.org/wiki/Clipping_%28Computergrafik%29
- http://www.cs.helsinki.fi/group/goa/viewing/leikkaus/lineClip.html
- http://wiki.delphigl.com/index.php/konvex
- http://de.wikipedia.org/wiki/Skalarprodukt
- http://de.wikipedia.org/wiki/Kreuzprodukt
- http://de.wikipedia.org/wiki/Spatprodukt
- http://informatik-forum.at/
- http://en.wikipedia.org/wiki/Scaling_(geometry)
- http://de.wikipedia.org/wiki/Projektion_%28Geometrie%29
- http://www.cg.tuwien.ac.at/courses/CG/repetitorium.pdf
- http://www.cg.tuwien.ac.at/courses/CG/textblaetter.html
- http://www-lehre.informatik.uni-osnabrueck.de/~cg/2000/skript/ (Danke an „rewind“)

Version: 0.6
Zusammenfassung: Martin Tintel (mtintel)