# Programmieraufgabe
# Bresenheim Algorithmus

Björn Rathjen & Patrick Winterstein

May 27, 2014

# 1 Programmablauf

## 1.1 Klassen

Unser Programm besitzt im Wesentlichen 2 Klassen:

### 1.1.1 GrafikDemoProgramm

Die Klasse GrafikDemoProgramm ist für die grafische Benutzeroberfläche und somit für die Darstellung der Strecken zuständig.
Sie beinhaltet einen Klasse namens Testpanel, welche von der Klasse JPanel von Java erbt, um diese für unsere Zwecke zu erweitern.
Für unsere Zwecke fügen wir einen MouseListener hinzu, der für das setzen des Anfangs- und Endpunktes der Strecke benötigt wird. Hierbei überprüfen wir, ob die linke Maustaste gedrückt wird, dann setzen wir den 1.Punkt. Wird eine andere Moustaste gedrückt, setzen wir den 2.Punkt. So können wir beide Punkte gezielt setzen.
Zusätzlich wird die paintComponents(Graphics g) Methode überschrieben. Hierbei wird zuerst die complete Graphics g grau gezeichnet, um dann auf die graue fläche die weiße Strecke zu zeichen. Hierfür wird unsere Methode mypaint aufgerufen.
Diese Methode prüft die verschiedenen Fälle und ruft dann den richtigen Algorithmus auf. Diese Fälle werden anhand der Steigung zwischen den Punkten unterschieden.

### 1.1.2 AlgorithmXp

Die Klasse AlgorithmXp stellt die verschiedenen Abwandlungen des Bresenheim-Algorithmuses zur Verfügung, die zur Behandlung der verschiedenen Fälle benötigt werden.
Die Klasse wird per Konstruktor initialisiert, damit man die "Stärke" der Pixel angeben kann.

## 1.2 Abwandlungen des Algorithmus

Für die folgenden Fälle ist zu beachten, dass Java von links oben nach rechts unten größere Pixelzahlen hat, damit sind die Fälle der Steigungen nicht die selben, wie wenn man sich diese in einem normalen Koordiantenkreuz vorstellt.

### 1.2.1 Fall 1: $m > -1 \,\&\, m < 0$ (Aus der Vorlesung)

Formt man den Algorithmus aus der Vorlesung 1:1 in eine Java-Funktion um, kommt man auf dieses Ergebnis:

```
public void mgm1(Point p, Graphics g){
    int s = 4*p.y - 2*p.x + weight;
    int ne = 4*(p.y-p.x);
    int e = 4*p.y;
    int j = 0;
    for(int i = 0; i < p.x; i+=weight){
        g.fillRect(i, j, weight, weight);
        if(s > 0){
            j += weight;
            s += ne;
        }else{
            s += e;
        }
    }
}
```

Da bei uns der 1. Punkt allerdings nicht zwangsläufig im Ursprung liegt, müssen wir diese Verschiebung mit einberechen. Dafür definieren wir uns 2 Varibalen $x$ und $y$, die wir wie folgt initalisieren:

```
public void mgm1(Point p2, Point p2, Graphics g){
    int x = p2.x - p1.x;
    int y = p2.y - p1.y;
    ...
}
```

Somit ist der Punkt (x,y) der Punkt p2, wenn man p1 in den Urpsrung verschiebt. Im Algorithmus ersetzen wir nun alle $p.x$ mit $x$ und alle $p.y$ mit $y$ und zeichnen den Pixel and der Stelle $(p1.x + i,\ p1.y + j)$. Damit verschieben wir also den gezeichneten Pixel wieder zurück.
Damit kommen wir auf folgenden Gesamtfunktion:

```
public void mgm1(Point p1, Point p2, Graphics g){
    int x = p2.x - p1.x; int y = p2.y - p1.y;
    int s = 4*y - 2*x + weight;
    int ne = 4*(y-x); int e = 4*y;
    int j = 0;
    for(int i = 0; i < x; i+=weight){
        g.fillRect(p1.x + i, p1.y+j, weight, weight);
        if(s > 0){
            j += weight;
            s += ne;
        }else{
            s += e;
        }
    }
}
```

### 1.2.2 Fall 2: $m <= -1$

Im Fall aus der Vorlesungen haben wir definiert, dass wir für jede vertikale Gitterlinie einen Pixel zeichen. Dies macht Sinn, da wir in dem Fall einen Strecke haben, wenn man diese als Hypothenuse eines rechtwinklingen Dreiecks betrachtet, die Kathete, die parellel zu x-Achse ist, länger ist als jene, die parallel zur y-Achse ist.
In diesem Fall allerdings, ist die "y-Kathete" länger als die "x-Kathete". Also legen wir fest: Wir zeichnen für jede horizontale Gitterlinie einen Pixel.
Daraus ergibt sich, das wir in der Schleife x mit y ersetzen müssen, sowie der Schleifenzähler nun j ist. Die Berechnungn für x und y bleibt gleich. Bei den restlichen Berechnungen müssen wir x und y vertauschen.
Daraus ergibt sich folgender Code:

```
public void mkm1(Point p1, Point p2, Graphics g){
    int x = p2.x - p1.x;
    int y = p2.y - p1.y;
    int s = 4*x - 2*y + weight;
    int ne = 4*(x-y);
    int e = 4*x;
    int i = 0;
    for(int j = 0; j < y; j+=weight){
        g.fillRect(p1.x + i, p1.y + j, weight, weight);
        if(s > 0){
            i += weight;
            s += ne;
        }else{
            s += e;
        }
    }
}
```

### 1.2.3 Fall 3: $m >= 1$

Jetzt wurde es für uns etwas schwieriger.
Wir haben wieder den Fall, dass wir für jede horizontale Gitterlinie einen Pixel zeichnen, somit bleibt $j$ der Schleifenzähler.
Nun liegt p1 aufgrund unserer Fallunterscheidung links unter p2. Damit ergibt sich, dass x und y positiv sind. Vorher war y immer negativ, da p1 links über p2 lag. Deswegen ersetzen wir alle y mit -y. Damit nun der Punkt auch noch an der richtigen Stelle gezeichnet wird, müssen wir j von p1.y abziehen.
Damit ergibt sich:

```
public void mg1(Point p1, Point p2, Graphics g){
    System.out.println("mg1");
    int x = p2.x - p1.x;
    int y = p2.y - p1.y;
    int s = 4*x + 2*y + weight;
    int ne = 4*(x+y);
    int e = 4*x;
    int i = 0;
```

```
    for(int j = 0; j < -y; j+=weight){
        g.fillRect(p1.x + i, p1.y - j, weight, weight);
        if(s > 0){
            i += weight;
            s += ne;
        }else{
            s += e;
        }
    }
}
```

#### 1.2.4 Fall 4: $m > 0 \,\&\, m < 1$

Die Umformung von Fall 1 zu Fall 4 ist equivalent zu der von Fall 2 zu Fall 3, deswegen beschreiben wir diese hier nicht nocheinmal.

```
public void mk1(Point p1, Point p2, Graphics g){
    int x = p2.x - p1.x;
    int y = p2.y - p1.y;
    int s = 4*y + 2*x + weight;
    int ne = 4*(x+y);
    int e = 4*y;
    int j = 0;
    for(int i = 0; i < -x; i+=weight){
        g.fillRect(p1.x - i, p1.y + j, weight, weight);
        if(s > 0){
            j += weight;
            s += ne;
        }else{
            s += e;
        }
    }
}
```