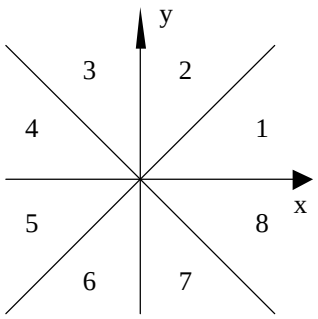
Aufgabe 6

Der Linienalgorithmus beruht sehr stark auf dem in der Vorlesung vorgestellten Verfahren. Leider war letzteres nicht in der Lage, beliebige Linien darstellen zu können, sondern stellte die Bedingung, dass sowohl die Steigung A zwischen 0 und 1 liegt als auch $x_1<x_2$.
Mit etwas Überlegung wird klar, dass die Grenzfälle A=0 und A=1 noch problemlos mit dem Algorithmus rasterisierbar sind, da dann stets d<=0 (es wird immer E als nächster Punkt gewählt) oder d>0 (NE ist die richtige Wahl) gilt, was korrekt umgesetzt wird.
In einer Grafik, wo der Startpunkt der Linie im Koordinatenursprung zentriert ist, erkennt man, dass damit aber erst ein Achtel alle tatsächlich möglichen Fälle abgedeckt wird, nämlich Bereich 1:

Abbildung 1



Recht einfach sind Linien in den Sektoren 3 bis 6 zu zeichnen: Man vertauscht Start- und Endpunkt, wenn der Startpunkt wieder in den Koordinatenursprung gelegt wird, erhält man Linien in den Sektoren 7,8,1 oder 2.

```
// zeichne immer von links nach rechts, vertausche ggf. Eckpunkte
if (x2 < x1)
{
    int temp;

    temp = x2;
    x2 = x1;
    x1 = temp;

    temp = y2;
    y2 = y1;
    y1 = temp;
}
```

Als nächstes folgt die Überlegung, dass sich die Bereiche 7,8 nur dadurch von 1 und 2 unterscheiden, dass der Anstieg negativ ist. Wenn man dies vorher erkennt, so kann man den Absolutbetrag von *dy* bilden und die Steigung entlang der y-Achse abändern:

```
// dx ist immer nicht-negativ (aufgrund des vorhergehendes Tausches von x1,x2)
int dx = x2 - x1;
// dy kann auch negativ sein ...
int dy = y2 - y1;

// berechne Steigung pro Iteration entlang der y-Achse
int ystep = 1;
if (dy < 0)
{
    ystep = -1;
    // ab jetzt ist dy immer nicht-negativ
    dy = -dy;
}
```

Im Sektor 1 kommt der Originalalgorithmus aus der Vorlesung zum Zuge, es wurde nur `y++` in `y+=ystep` abgeändert, um negativen Steigungen gerecht zu werden:

```
// Steigung <= 1 ?
if (dy <= dx)
{
    // Standardalgorithmus aus Vorlesung, lediglich y++ durch y += ystep ersetzt
    int d  = 2*dy - dx;
    int dE = 2*dy;
    int dNE= 2*(dy-dx);
    int x  = x1;
    int y  = y1;

    raster_.setPixel(x,y);

    while(x != x2)
    {
        if (d <= 0)
            d += dE;
        else
        {
            d += dNE;
            y += ystep;
        }
        x++;

        raster_.setPixel(x,y);
    }
}
```

Der Code für Sektor 2 wurde als Kopie des Verfahrens für Sektor 1 erzeugt, allerdings sind alle Vorkommen von x durch y (und umgekehrt) ersetzt worden, selbst auf dx und dy trifft dies zu. Nicht geändert wurde das Zeichnen der Pixel, dort stehen x und y in der ursprünglichen Reihenfolge.

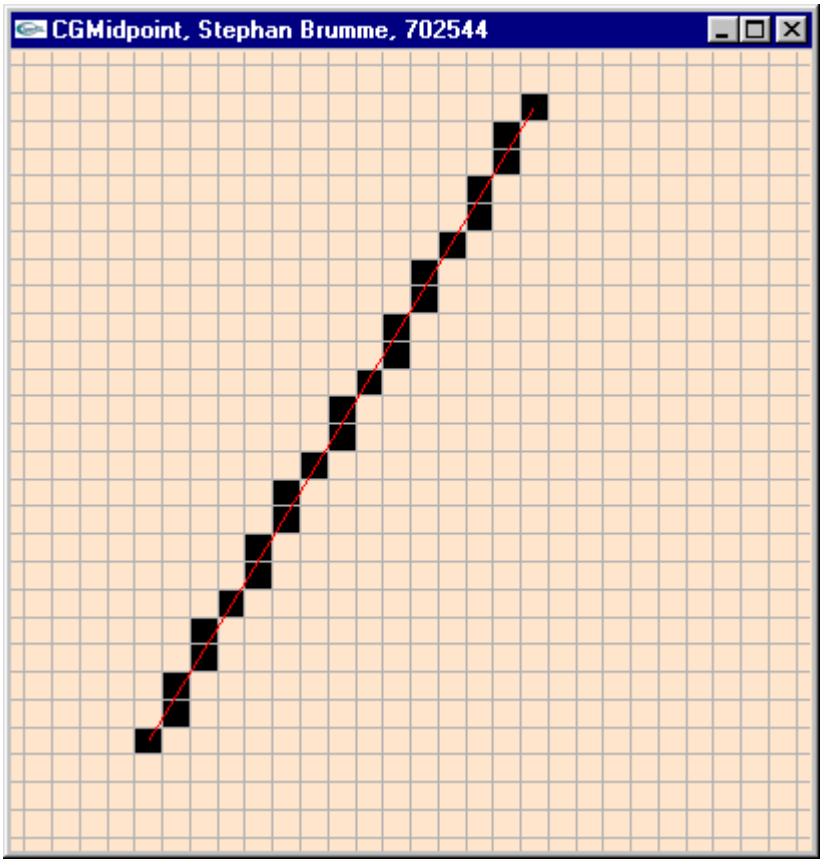
```
else
{
    // alle x durch y ersetzt (und umgekehrt, auch dx und dy !),
    // lediglich setPixel blieb jeweils unverändert
    int d  = 2*dx - dy;
    int dNE= 2*dx;
    int dN = 2*(dx-dy);
    int x  = x1;
    int y  = y1;

    raster_.setPixel(x,y);

    while(y != y2)
    {
        if (d <= 0)
            d += dNE;
        else
        {
            d += dN;
            x++;
        }
        y += ystep;

        raster_.setPixel(x,y);
    }
}
```

Abbildung 2



Das Raster wird in einem Array aus `bool`s abgespeichert, wobei der Startpunkt unten links ist, was auch dem Koordinatensystem von OpenGL entspricht. Leider ist es nicht möglich, auf einfachem Wege zweidimensionale Felder dynamisch zu erzeugen, deshalb behelfe ich mich mit einem eindimensionalen und verwende zur Bestimmung der korrekten Position die Formel

$$pos = x + y \cdot Breite$$

Um extrem große Raster zu verhindern, wird im Konstruktor die Breite bzw. Höhe gegen MAXWIDTH bzw. MAXHEIGHT geprüft. Sowohl beim Setzen als auch beim Auslesen eines Rasterpunktes erfolgt eine kurze Bereichsüberprüfung, die unzulässige Speicherzugriffe verhindert.
Bei der Bildschirmdarstellung mache ich mir zunutze, dass OpenGL Linien standardmäßig mit einer Breite von 1 Pixel zeichnet, unabhängig von der Auflösung oder Fenstergröße. Genau dieses Verhalten wird bei der eigentlichen Rasterpunktanzeige problematisch. Ich umgehe diese Hürde dadurch, dass ich Rechtecke mit einer Breite und Höhe von jeweils 1 zeichne. Mein Raster hat generell die Eigenschaft, dass ein Punkt von *(x-0.5,y-0.5)* bis *(x+0.5,y+0.5)* geht, der geometrische Punkt *(x,y)* liegt demzufolge genau im Zentrum des Rasterpunktes. Entsprechend sind die Randpunkte auf dem Bildschirm nur teilweise zu sehen, die tatsächlich sichtbare Breite bzw. Höhe ist um 1 kleiner als die im Konstruktor vorgegebenen Werte.
Um unschöne Effekte beim Programmstart zu vermeiden, beginnt die Linienzeichnung erst *nach* Festlegung des ersten Endpunktes, d.h. nach Bewegung der Maus bei gedrückter Maustaste. Die rote Hilfslinie, die zur Veranschaulichung der Idealrasterisierung dient, verläuft - gemäß obiger Definition meines Rasters – von (x_1, y_1) bis (x_2, y_2). Etwas komplizierter ist die Bestimmung der Rasterposition der Maus, die aus der Bildschirmposition errechnet werden muss:

$$x_{Raster} = 0{,}5 + \frac{x_{Bildschirm}}{Breite_{Bildschirm} - 1} \cdot Breite_{Raster}$$

$$y_{Raster} = 0{,}5 + \frac{y_{Bildschirm}}{Höhe_{Bildschirm} - 1} \cdot Höhe_{Raster}$$

Im Quellcode muss ich noch zusätzlich auf die unterschiedlichen Datentypen achten (`int` bzw. `float`), sodass folgende Zeilen notwendig sind:

```
// Mausposition in Rasterkoordinaten umrechnen
    x = (int)(0.5 + x / ((float)winWidth_ /(raster_.width()-1)));
    y = (int)(0.5 + y / ((float)winHeight_/(raster_.height()-1)));
```

Quelltext

siehe Anhang
(CGApplication wurde nicht verändert und ist deshalb dort auch nicht aufgeführt)

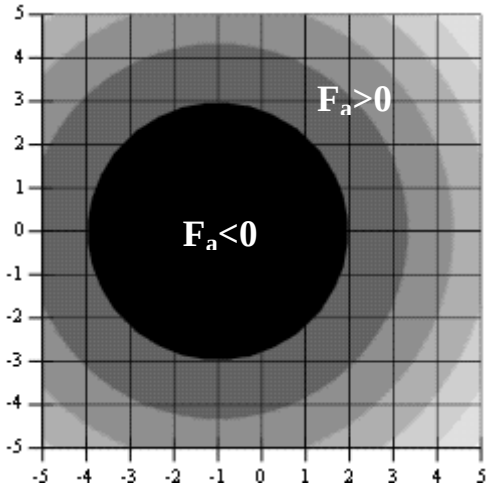
Aufgabe 7

In folgenden Erklärungen werde ich kurz den gebräuchlichen Namen der Funktionen (in ihrer expliziten Darstellung) angeben und den Bereich benennen, für den die implizite Funktion negative Werte ergibt.
Da mir kein Scanner zur Verfügung steht, musste ich gezwungenermaßen die Skizzen im Rechner erstellen und dabei auf explizite Ausdrücke zurückgreifen. Dunkle Flächen stehen für negative Werte, helle für positive; leider war die verwendete Software nicht in der Lage, die Nulllinie korrekt zu zeichnen, ich hole dies durch manuelles Einfügen von Indikatoren der Form $F > 0$ bzw. $F < 0$ nach.

$$F_a(x,y) = (x+1)^2 + y^2 - 9 = 0$$

Diese Gleichung ist die implizite Funktion zur Darstellung eines Kreises, dessen Mittelpunkt bei (-1,0) liegt und der einen Radius von 3 besitzt. Sie nimmt für alle Punkte innerhalb der Kreislinie negative Werte an.

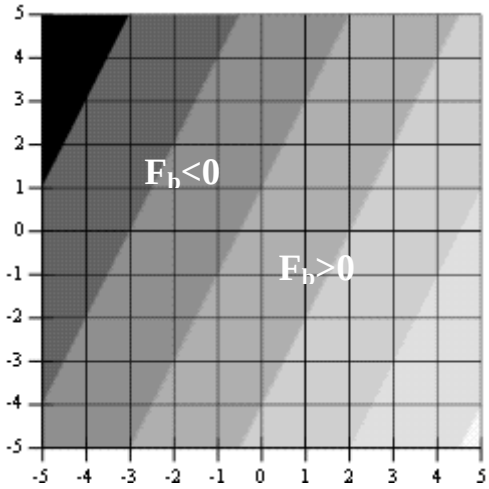
Abbildung 3



$$F_b(x,y) = 2x - y + 1 = 0$$

Die Gerade mit der Steigung 2, die die y-Achse im Punkt (0,1) schneidet, wird durch die angegebene implizite Funktion beschrieben. Alle Punkte unterhalb der Linie bewirken ein negatives Ergebnis.

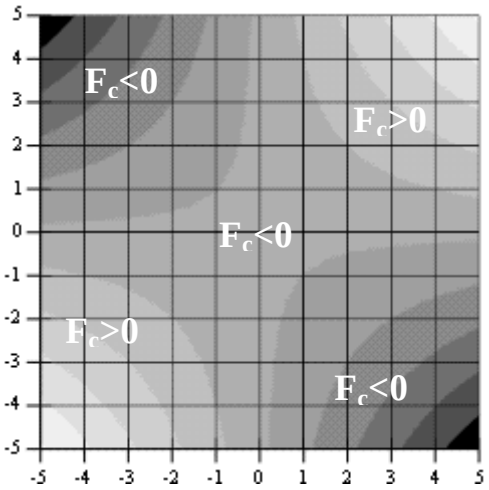
Abbildung 4



$$F_c(x,y) = xy - 4 = 0$$

Eine Hyperbel, die durch die Punkte (-2,-2) und (2,2) geht, kann mittels dieser impliziten Funktionen erzeugt werden. Punkte, die im 2. oder 4. Quadranten liegen, sorgen für negative Resultate.

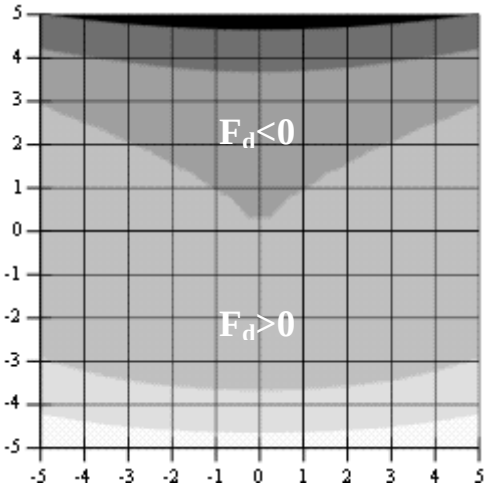
Abbildung 5



$$F_d(x,y) = x^2 - y^3 = 0$$

Die damit beschriebene Potenzfunktion ergibt bei Einsetzen von Punkten, die oberhalb von ihr liegen, negative Werte.

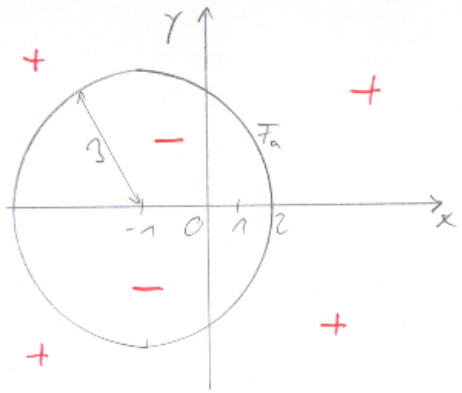
Abbildung 6



Erst in kurz vor Abgabeschluss dieses Aufgabenblattes gelang es mir, einen Scanner aufzutreiben. Jedoch wollte ich die obigen, bereits per Software generierten Bilder nicht entfernen, sodass ich die Originalskizze hier aufführe:

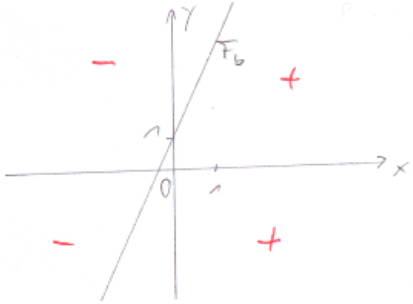
$F_a(x,y) =$

$(x+1)^2+y^2-9=0$



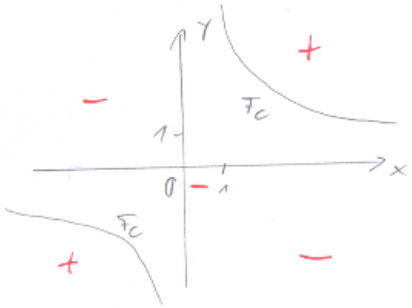
$F_b(x,y) =$

$2x-y+1=0$



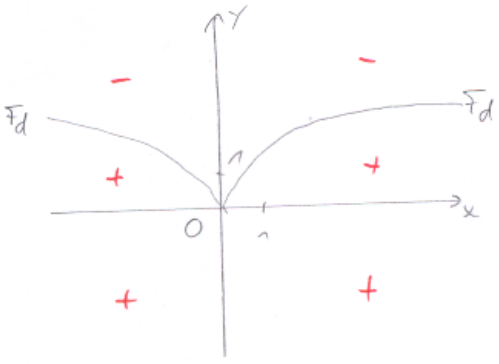
$F_c(x,y) =$

$xy-4=0$



$F_d(x,y) =$

$x^2-y^3=0$



Aufgabe 8

Die grundlegende Vorgehensweise besteht darin, dass die Gerade G_1 bzw. G_2 nach einer Variablen (ich entschied mich für x) umgeformt werden und diese in die Kreisgleichung K eingesetzt wird. Die entstehende quadratische Gleichung wird aufgelöst und beide Ergebnisse y_1 und y_2 dienen über die Liniengleichung zur Bestimmung von x_1 und x_2:

$G_1(x,y)$ und $K(x,y)$:

$$\begin{aligned}
x - y - 17 &= 0 \\
x &= y + 17 \\
x^2 + y^2 - 169 &= 0 \\
(y+17)^2 + y^2 - 169 &= 0 \\
2y^2 + 34y + 120 &= 0 \\
y^2 + 17y + 60 &= 0 \\
y_{1,2} &= -\frac{17}{2} \pm \sqrt{\frac{289}{4} - 60} \\
&= -\frac{17}{2} \pm \frac{7}{2} \\
y_1 &= -5 \\
y_2 &= -12 \\
x_1 &= y_1 + 17 \\
&= 12 \\
x_2 &= y_2 + 17 \\
&= 5
\end{aligned}$$

Eine kurze Überprüfung bestätigt beide Ergebnisse:

$$\begin{aligned}
12^2 + (-5)^2 - 169 &= 0 \\
169 - 169 &= 0 \\
5^2 + (-12)^2 - 169 &= 0 \\
169 - 169 &= 0
\end{aligned}$$

Somit ist G_1 eine Sekante von K mit den Schnittpunkten $S_1(12,-5)$ und $S_2(5,-12)$.

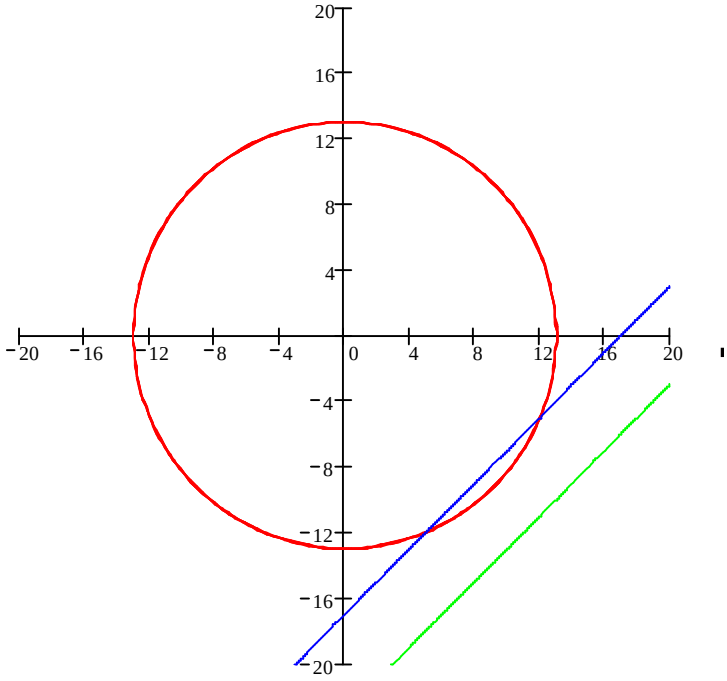
$G_2(x,y)$ und $K(x,y)$:

$$x^2 + y^2 - 169 = 0$$
$$x - y - 23 = 0$$
$$x = y + 23$$
$$(y+23)^2 + y^2 - 169 = 0$$
$$2y^2 + 46y + 360 = 0$$
$$y^2 + 23y + 180 = 0$$
$$y_{1,2} = -\frac{23}{2} \pm \sqrt{\frac{529}{4} - 180}$$

An dieser Stelle muss man abbrechen, da unter der Wurzel ein negativer Ausdruck entsteht, was im Bereich der reellen Zahlen nicht zulässig ist. Die geometrische Interpretation dieses Resultates ist, dass die Gerade eine Passante des Kreises darstellt.

Zeichnet man sowohl den Kreis als auch beide Geraden in einem kartesischen Koordinatensystem auf, so erhält eine sehr schöne Veranschaulichung meiner Ergebnisse:

Abbildung 7



Anhang

Quelltext aus Aufgabe 6

source file "cgraster.h"

```
// Computergraphik I
// Prof. Dr. Juergen Doellner
// Sommersemester 2001
//
// Rahmenprogramm fuer Aufgabenzettel 3

// Stephan Brumme, 702544
// last changes: May 12, 2001

#ifndef CG_RASTER_H
#define CG_RASTER_H

#include "cgapplication.h"

class CGRaster {
public:
    CGRaster(int width, int height);
    ~CGRaster();

    int width() const;
    int height() const;

    void clear();

    bool getPixel(int x, int y) const;
    void setPixel(int x, int y);

    void draw() const;

    enum
    {
        MAXWIDTH  = 100,
        MAXHEIGHT = 100
    };

private:
    // Attribute fuer width, height und
    // den bool-Array
    int m_nWidth;
    int m_nHeight;

    bool* m_arBuffer;
};

#endif // CG_RASTER_H
```

source file "cgraster.cpp"

```
// Computergraphik I
// Prof. Dr. Juergen Doellner
// Sommersemester 2001
//
// Rahmenprogramm fuer Aufgabenzettel 3

// Stephan Brumme, 702544
// last changes: May 12, 2001

#include "cgraster.h"
#include <memory.h>
#include "gl/gl.h"


CGRaster::CGRaster(int width, int height) {
```

```
    // speichern von width und height
    // in Objektvariablen

    // überprüfen, ob Ausmaße vernünftig sind
    if (width  <= 0 || width  > MAXWIDTH)
        width  = MAXWIDTH;
    if (height <= 0 || height > MAXHEIGHT)
        height = MAXHEIGHT;

    m_nWidth = width;
    m_nHeight = height;

    // anlegen des bool-Arrays
    m_arBuffer = new bool[m_nWidth*m_nHeight];

    // loeschen des Arrays
    clear();
}


CGRaster::~CGRaster() {
    // Rasterspeicher freigeben
    delete [] m_arBuffer;
}


int CGRaster::width() const {
    return m_nWidth;
}


int CGRaster::height() const {
    return m_nHeight;
}


void CGRaster::clear() {
    // setzten aller Werte des bool-Arrays auf false
    memset(m_arBuffer, false, sizeof(bool)*m_nWidth*m_nHeight);
}


bool CGRaster::getPixel(int x, int y) const {
    // liegen Koordinaten innerhalb des Rasters ?
    if (x<0 || x>=m_nWidth ||
        y<0 || y>=m_nHeight)
        return false;

    // Punkt zurückgeben
    return m_arBuffer[x+y*m_nWidth];
}


void CGRaster::setPixel(int x, int y) {
    // liegen Koordinaten innerhalb des Rasters ?
    if (x<0 || x>=m_nWidth ||
        y<0 || y>=m_nHeight)
        return;

    // Punkt setzen
    m_arBuffer[x+y*m_nWidth] = true;
}


void CGRaster::draw() const {
    // zeichnen aller Pixel, die gesetzt sind

    // Umgebung sichern
    glPushAttrib(GL_CURRENT_BIT);

    // schwarz
    glColor3f(0.0, 0.0, 0.0);

    // alle Punkte des Rasters durchlaufen
    int x,y;
```

```
    for (x=0; x<m_nWidth; x++)
        for (y=0; y<m_nHeight; y++)
            // ggf. Rechteck mit Seitenlänge 1 zeichnen, im Zentrum liegt (x,y)
            if (getPixel(x,y))
                glRectf(x-0.5, y-0.5, x+0.5, y+0.5);


    // zeichnen der Raster-Linien
    glColor3f(0.7, 0.7, 0.7);
    glBegin(GL_LINES);
    // vertikal
    for (x=0; x<m_nWidth; x++)
    {
        glVertex2f(x+0.5, 0);
        glVertex2f(x+0.5, m_nWidth);
    }

    // horizontal
    for (y=0; y<m_nHeight; y++)
    {
        glVertex2f(0, y+0.5);
        glVertex2f(m_nHeight, y+0.5);
    }
    glEnd();

    glPopMatrix();
}
```

source file "cgmidpoint.h"

```
// Computergraphik I
// Prof. Dr. Juergen Doellner
// Sommersemester 2001
//
// Rahmenprogramm fuer Aufgabenzettel 3

// Stephan Brumme, 702544
// last changes: May 12, 2001

#ifndef CG_BRESENHAM_H
#define CG_BRESENHAM_H

#include "cgapplication.h"
#include "cgraster.h"

class CGMidpoint : public CGApplication {
public:
    CGMidpoint(int width, int height);

    virtual void onInit();
    virtual void onDraw();
    virtual void onSize(unsigned int newWidth,unsigned  int newHeight);

    virtual void onButton(MouseButton button, MouseButtonEvent event, int x, int y);
    virtual void onMove(MouseButton button, int x, int y);

private:
    // zeichenen der Linie in das Raster raster_
    void drawLine(int x1, int y1, int x2, int y2);

    // interne Raster-Klasse
    CGRaster raster_;

    // Fenstergroesse speichern
    int winWidth_;
    int winHeight_;

    // Linienanfangspunkt und Endpunkt
    int xBegin_;
    int yBegin_;
    int xEnd_;
    int yEnd_;
```

```
    bool First_;
};

#endif // CG_BRESENHAM_H
```

source file "cgmidpoint.cpp"

```
// Computergraphik I
// Prof. Dr. Juergen Doellner
// Sommersemester 2001
//
// Rahmenprogramm fuer Aufgabenzettel 3

// Stephan Brumme, 702544
// last changes: May 12, 2001

#include "cgmidpoint.h"
#include <stdio.h>

CGMidpoint::CGMidpoint(int width, int height) : raster_(width,height) {
    // Objektvariablen initialisieren
    First_ = true;
}

void CGMidpoint::onInit() {
    // Hintergrundfarbe weiss
    glClearColor(1, 1, 1, 1);

    // ohne perspektivische Verzerrung, da nur 2D
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0, raster_.width()-1, 0, raster_.height()-1);
}

void CGMidpoint::onDraw() {
    // Colorbuffer loeschen
    glClear(GL_COLOR_BUFFER_BIT);

    // Zeichenen der Raster-Klasse
    raster_.draw();

    // Zeichenen einer roten Kontroll-Linie
    // mit OpenGL

    // erst durchführen, wenn ein Endpunkt feststeht
    if (!First_)
    {
        // Umgebung sichern
        glPushAttrib(GL_CURRENT_BIT);
        // rote Linie
        glColor3f(1.0, 0.0, 0.0);

        // und Hilfslinie zeichnen
        glBegin(GL_LINES);
        glVertex2f(xBegin_, yBegin_);
        glVertex2f(xEnd_  , yEnd_  );
        glEnd();
        // Umgebung wiederherstellen
        glPopAttrib();
    }

    // Backbuffer anzeigen
    swapBuffers();
}

void CGMidpoint::onSize(unsigned int newWidth, unsigned int newHeight) {
    // neue Fenstergröße speichern
    winWidth_ = newWidth;
    winHeight_ = newHeight;
    glViewport(0, 0, newWidth - 1, newHeight - 1);
}
```

```
void CGMidpoint::onButton(MouseButton button, MouseButtonEvent event, int x, int y) {
    // Sichern der x und y Werte
    // Anfang und Ende unterscheiden durch MousButtonEvent:
    // MouseButtonDown = Start
    // MouseButtonUp = End

    // Mausposition in Rasterkoordinaten umrechnen
    x = (int)(0.5 + x / ((float)winWidth_ /(raster_.width()-1)));
    y = (int)(0.5 + y / ((float)winHeight_/(raster_.height()-1)));

    // Linienanfang
    if (event == MouseButtonDown)
    {
        xBegin_ = x;
        yBegin_ = y;
    }

    // Linienende
    if (event == MouseButtonUp)
    {
        xEnd_ = x;
        yEnd_ = y;
        // Endpunkt steht fest, Linie darf gezeichnet werden
        First_ = false;
    }

    // Raster löschen
    raster_.clear();
    // Linie im Raster zeichnen
    if (!First_)
        drawLine(xBegin_, yBegin_, xEnd_, yEnd_);

    // Raster auf dem Bildschirm darstellen
    onDraw();
}

void CGMidpoint::onMove(MouseButton button, int x, int y) {
    // Endpunkt ermitteln
    onButton(button, MouseButtonUp, x, y);
}

void CGMidpoint::drawLine(int x1, int y1, int x2, int y2) {
    // Bresenham Algorithmus
    // Zeichnen mit der Raster-Klasse

    // zeichne immer von links nach rechts, vertausche ggf. Eckpunkte
    if (x2 < x1)
    {
        int temp;

        temp = x2;
        x2 = x1;
        x1 = temp;

        temp = y2;
        y2 = y1;
        y1 = temp;
    }

    // dx ist immer nicht-negativ (aufgrund des vorhergehendes Tausches von x1,x2)
    int dx = x2 - x1;
    // dy kann auch negativ sein ...
    int dy = y2 - y1;

    // berechne Steigung pro Iteration entlang der y-Achse
    int ystep = 1;
    if (dy < 0)
    {
        ystep = -1;
        // ab jetzt ist dy immer nicht-negativ
        dy = -dy;
    }

    // Steigung <= 1 ?
    if (dy <= dx)
```

```
    {
        // Standardalgorithmus aus Vorlesung, lediglich y++ durch y += ystep ersetzt
        int d  = 2*dy - dx;
        int dE = 2*dy;
        int dNE= 2*(dy-dx);
        int x  = x1;
        int y  = y1;

        raster_.setPixel(x,y);

        while(x != x2)
        {
            if (d <= 0)
                d += dE;
            else
            {
                d += dNE;
                y += ystep;
            }
            x++;

            raster_.setPixel(x,y);
        }
    }
    else
    {
        // alle x durch y ersetzt (und umgekehrt, auch dx und dy !),
        // lediglich setPixel blieb jeweils unverändert
        int d  = 2*dx - dy;
        int dNE= 2*dx;
        int dN = 2*(dx-dy);
        int x  = x1;
        int y  = y1;

        raster_.setPixel(x,y);

        while(y != y2)
        {
            if (d <= 0)
                d += dNE;
            else
            {
                d += dN;
                x++;
            }
            y += ystep;

            raster_.setPixel(x,y);
        }
    }
}

int main(int argc, char* argv[]) {
    CGMidpoint bresenham(30, 30);
    bresenham.start(argc, argv, "CGMidpoint, Stephan Brumme, 702544");
    return(0);
}
```