# GRAFIKA KOMPUTEROWA

## Wykład 3: podstawowe algorytmy grafiki dwuwymiarowej

Tymon Rubel

Zakład Elektroniki Jądrowej i Medycznej
Instytut Radioelektroniki i Technik Multimedialnych PW

Materiały opracowane w ramach zadania 15 „Modyfikacja międzywydziałowych studiów I stopnia na kierunku Inżynieria Biomedyczna”
projektu „NERW PW. Nauka – Edukacja – Rozwój – Współpraca”, współfinansowanego jest ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego

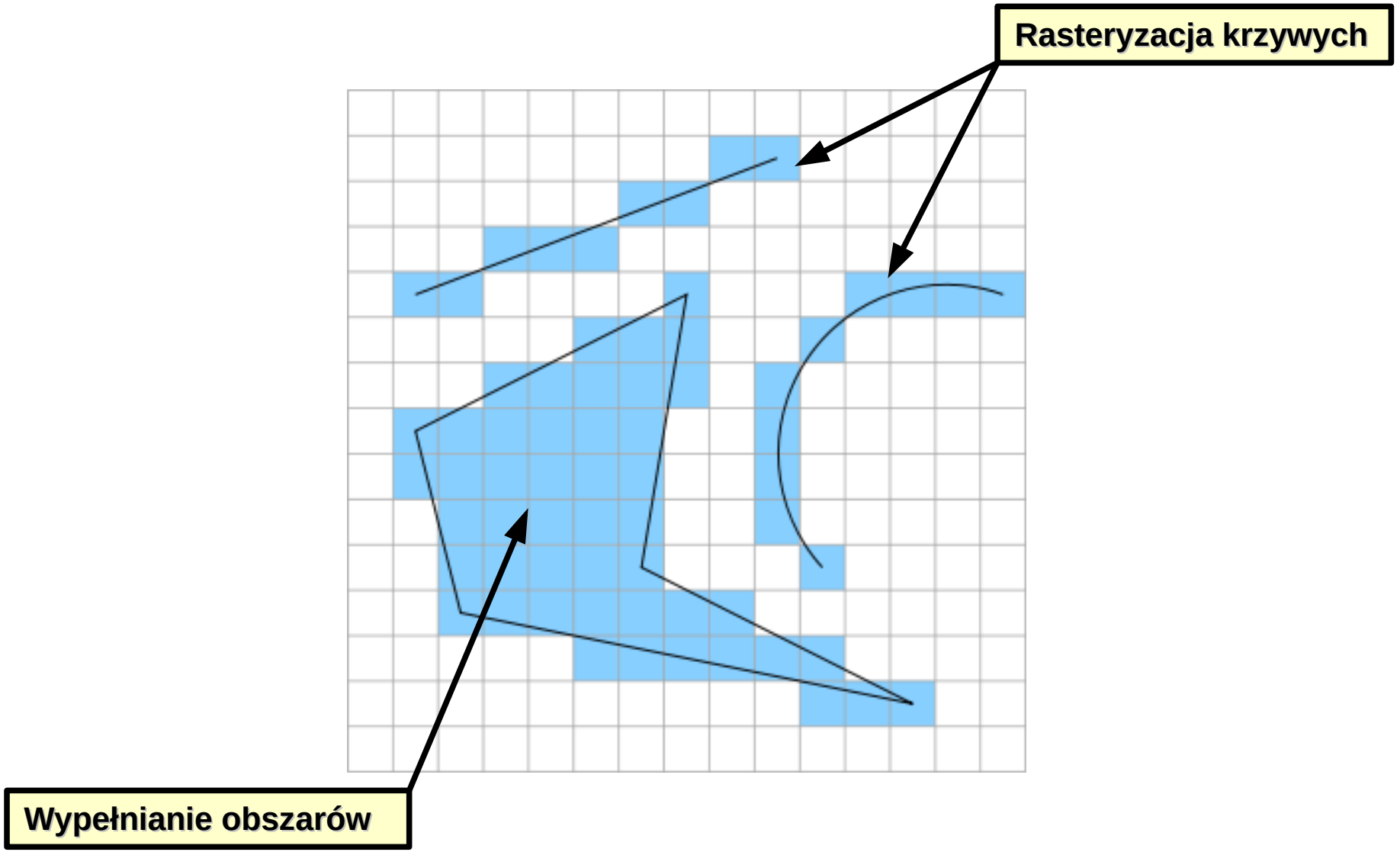
Fundusze Europejskie
Wiedza Edukacja Rozwój

Politechnika Warszawska

Unia Europejska
Europejski Fundusz Społeczny

# Podstawowe algorytmy grafiki dwuwymiarowej

**Tworzenie grafiki dwuwymiarowej wymaga rozwiązania problemów wynikających z konieczności reprezentowania figur geometrycznych za pomocą zbiorów pikseli. Zagadnieniami o podstawowym znaczeniu są tutaj: rasteryzacja i wypełnianie.**



# GRAFIKA KOMPUTEROWA

## Wykład 3: podstawowe algorytmy grafiki dwuwymiarowej (rasteryzacja)

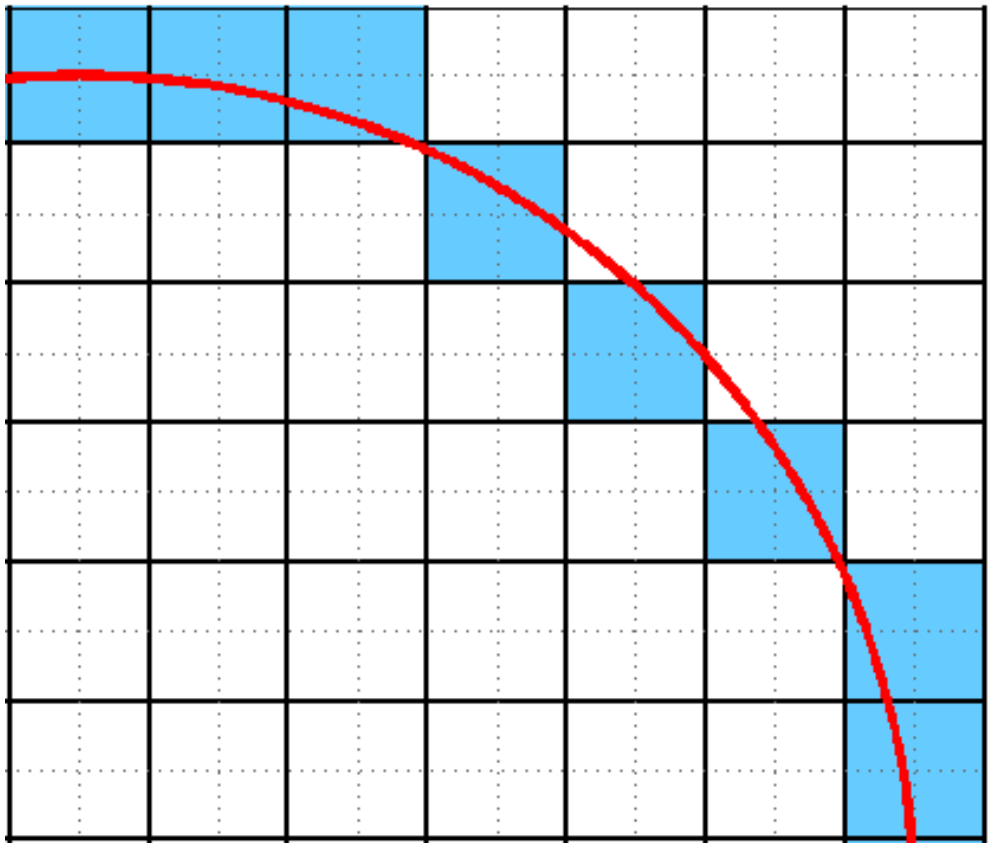
**Tymon Rubel**

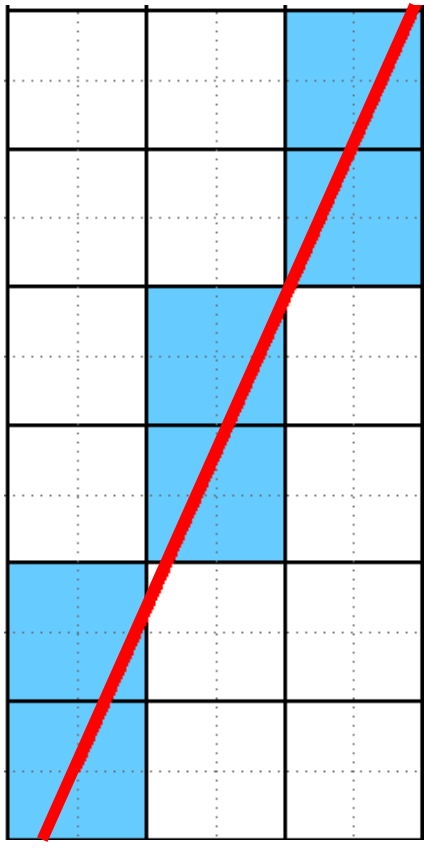
**Zakład Elektroniki Jądrowej i Medycznej**
**Instytut Radioelektroniki i Technik Multimedialnych PW**

# Podstawowe algorytmy grafiki dwuwymiarowej: rasteryzacja

**Rasteryzacja dwuwymiarowa** to proces mający na celu możliwie jak najwierniejsze odwzorowanie krzywej lub figury geometrycznej w przestrzeni **rastra**, czyli na dyskretnej płaszczyźnie złożonej z zadanej liczby pikseli o skończonych wymiarach.

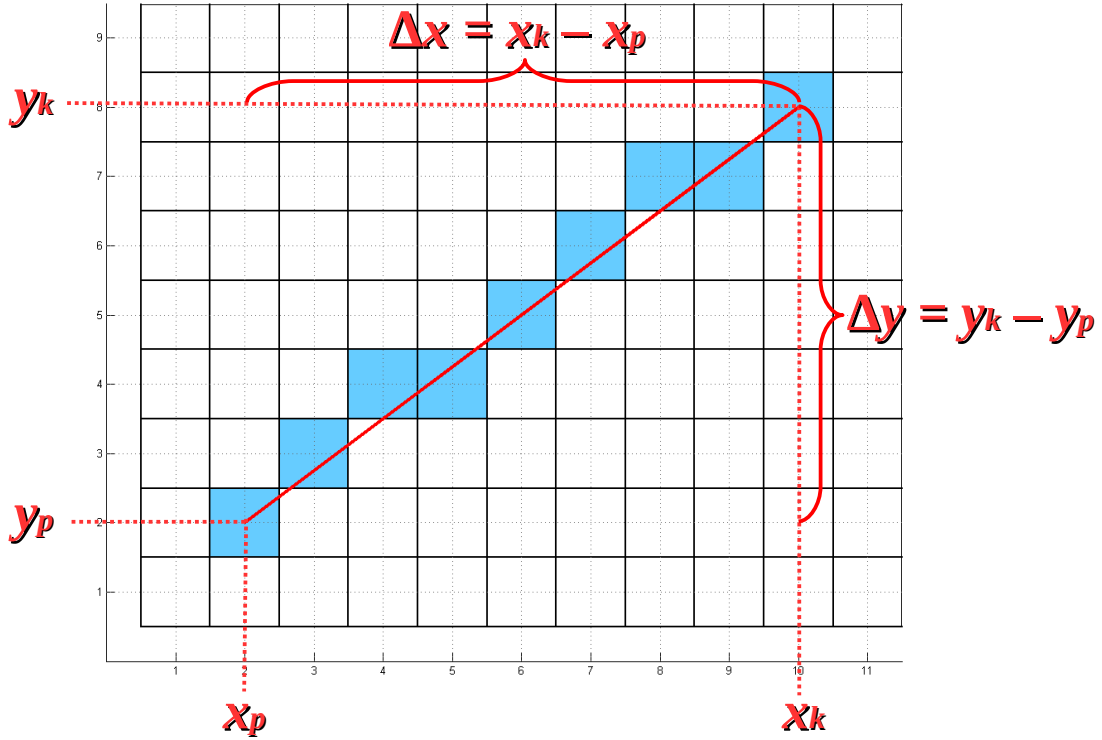
Odwzorowywana krzywa praktycznie nigdy nie przechodzi dokładnie przez węzły siatki, którą wyznaczają środki pikseli. Stąd też konieczność używania algorytmów pozwalających zminimalizować błąd reprezentacji rastrowej.





# Rasteryzacja: rysowanie odcinka

**Zadanie polega na określeniu pozycji pikseli stanowiących wizualizację odcinka o początku w punkcie $P_p\ (x_p, y_p)$ i końcu w punkcie $P_k\ (x_k, y_k)$.**



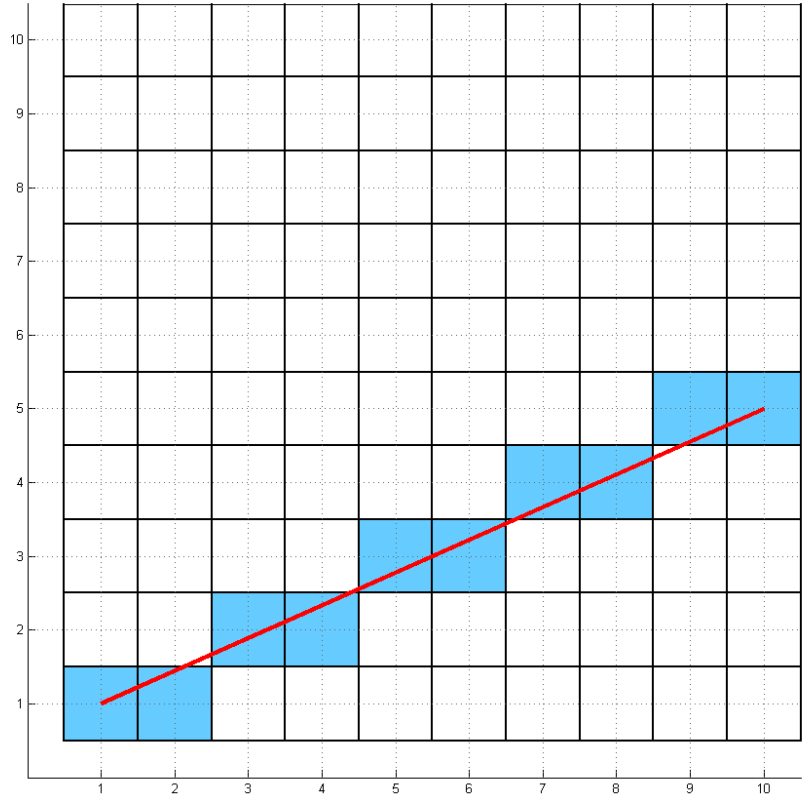

**Ogólny wzór prostej, na której leży odcinek to: $y = f(x) = a \cdot x + b$. Jednak znając współrzędne punktów $P_p$ i $P_k$ można go przedstawić w sposób bardziej zgodny ze sposobem sformułowania zadania:**

$$\left.\begin{aligned} a &= \tan(\alpha) = \frac{y_k - y_p}{x_k - x_p} = \frac{\Delta y}{\Delta x} \\ b &= y_p - a \cdot x_p = y_p - \frac{\Delta y}{\Delta x} \cdot x_p \end{aligned}\right\} \quad y = \frac{\Delta y}{\Delta x} \cdot (x - x_p) + y_p$$

# Rasteryzacja: rysowanie odcinka

**Niezależnie od algorytmu używanego do określania pozycji pikseli należy pamiętać o wyróżnieniu dwóch przypadków, zależnych od nachylenia prostej:**

- **$|a| \leq 1$: w każdej kolumnie (dla każdej liczby całkowitej $x_i \in [x_p, x_k]$) zawsze występuje dokładnie jeden piksel będący elementem odcinka, w każdym wierszu (dla każdej wartości całkowitej $y_i \in [y_p, y_k]$) występuje co najmniej jeden piksel odcinka;**

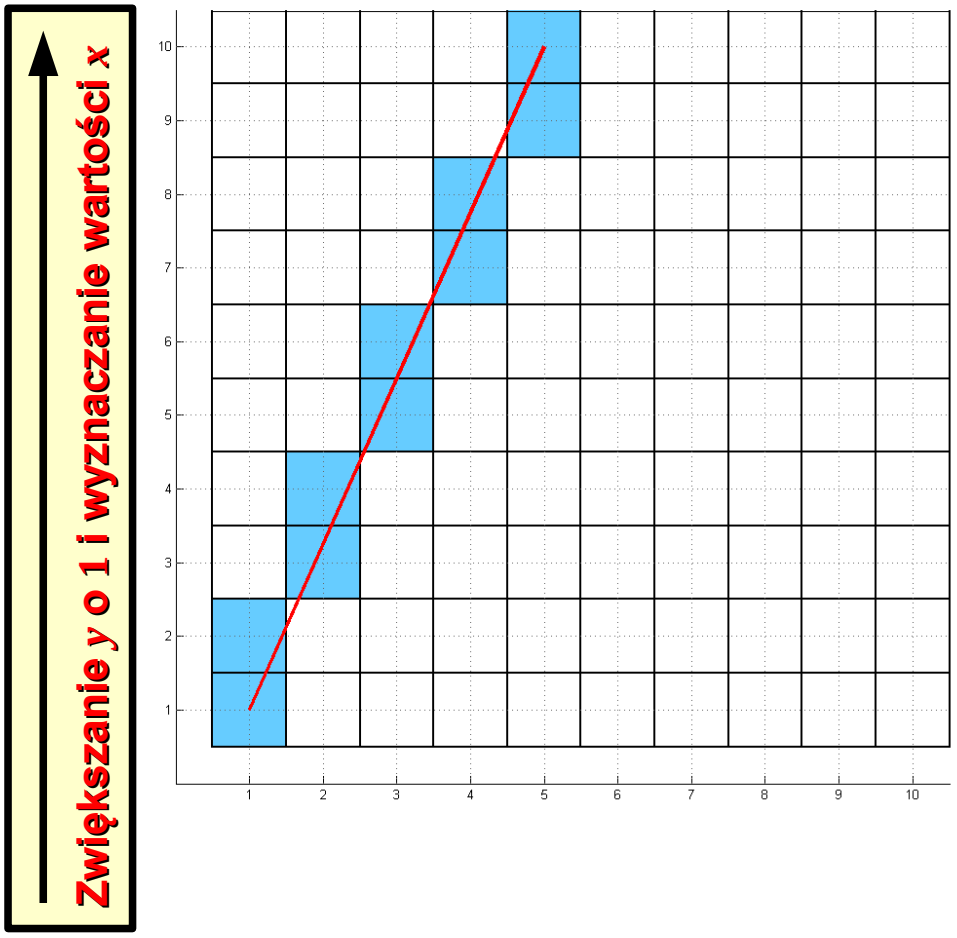



**Zwiększanie $x$ o 1 i wyznaczanie wartości $y$**

# Rasteryzacja: rysowanie odcinka

**Niezależnie od algorytmu używanego do określania pozycji pikseli należy pamiętać o wyróżnieniu dwóch przypadków, zależnych od nachylenia prostej:**

- **$|a| > 1$: w każdym wierszu (dla każdej liczby całkowitej $y_i \in [y_p, y_k]$) zawsze występuje dokładnie jeden piksel będący elementem odcinka, w każdej kolumnie (dla każdej wartości całkowitej $x_i \in [x_p, x_k]$ występuje co najmniej jeden piksel odcinka;**



# Rasteryzacja: rysowanie odcinka (prosty algorytm)

**Najprostszym rozwiązaniem jest użycie algorytmu korzystającego bezpośrednio z wzoru prostej, na której leży odcinek: dla każdej kolumny (lub wiersza, w zależności od nachylenia) wyznaczana jest rzeczywista wartość drugiej współrzędnej punktu leżącego na prostej, która następnie zaokrąglana jest do liczby całkowitej.**

**1. Obliczenie nachylenia i wyrazu wolnego na podstawie punktów początku i końca:**

$$a = \frac{y_k - y_p}{x_k - x_p}$$

$$b = y_p - a \cdot x_p$$

**2. Wyznaczenie współrzędnych pikseli w pętli:**

- **jeżeli** $|a| \leq 1 \rightarrow$ **dla każdego całkowitego** $x_i \in [x_p, x_k]$ **określenie** $y_i$ **jako:**

$$y_i = \text{round}\left(a \cdot x_i + b\right)$$

- **jeżeli** $|a| > 1 \rightarrow$ **dla każdego całkowitego** $y_i \in [y_p, y_k]$ **określenie** $x_i$ **jako:**

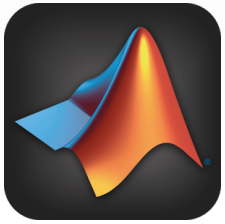
$$x_i = \text{round}\left(\frac{y_i - b}{a}\right)$$

# Rasteryzacja: rysowanie odcinka (prosty algorytm)

```
function odcinek(xp,yp,xk,yk)
    % Funkcja realizująca prosty algorytm
    % rysowania odcinka

    dx = xk - xp;
    dy = yk - yp;

    a = dy / dx;
    b = yp - a*xp;



    if (abs(a) < 1)
        for x = xp : xk
            y = round(a*x + b);
            set_pixel(x,y);
        end
    else
        for y = yp : yk
            if (dx == 0)
                x = xp;
            else
                x = round((y - b) / a);
            end;
            set_pixel(x,y);
        end
    end
```

Opis oznaczeń w kodach źródłowych:

niebieski - słowa kluczowe
czerwony - nazwy funkcji
zielony - komentarze
czarny - pozostałe elementy kodu



```
#include <math.h>

void odcinek(int xp, int yp, int xk, int yk) {
    // Funkcja realizująca prosty algorytm
    // rysowania odcinka

    int dx = xk - xp;
    int dy = yk - yp;

    double a = (double)dy / (double)dx;
    double b = yp - a*xp;

    int x, y;

    if (fabs(a) < 1) {
        for (x = xp; x <= xk; x++) {
            y = (int)round(a*x + b);
            set_pixel(x,y);
        }
    } else {
        for (y = yp; y <= yk; y++) {
            if (dx == 0) {
                x = xp;
            } else {
                x = (int)round((y - b) / a);
            }
            set_pixel(x,y);
        }
    }
}
```

# Rasteryzacja: rysowanie odcinka (prosty algorytm)

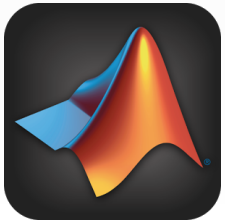
```
function odcinek(xp,yp,xk,yk)
    % Funkcja realizująca prosty algorytm
    % rysowania odcinka

    dx = xk - xp;
    dy = yk - yp;

    a = dy / dx;
    b = yp - a*xp;



    if (abs(a) < 1)
        for x = xp : xk
            y = round(a*x + b);
            set_pixel(x,y);
        end
    else
        for y = y
            if (d
                x
            else
                x
            end;
            set_p
        end
    end
```

```
#include <math.h>

void odcinek(int xp, int yp, int xk, int yk) {
    // Funkcja realizująca prosty algorytm
    // rysowania odcinka

    int dx = xk - xp;
    int dy = yk - yp;

    double a = (double)dy / (double)dx;
    double b = yp - a*xp;

    int x, y;

    if (fabs(a) < 1) {
        for (x = xp; x <= xk; x++) {
            y = (int)round(a*x + b);
            set_pixel(x,y);
        }
    } el
                                  = yk; y++) {
                                  {
                                  )round((y - b) / a);
                                  y);
        }
    }
}
```

Funkcja *set_pixel* (służąca do rysowania piksela) nie jest standardowym elementem ani języka C/C++, ani Matlaba. Należy ją samodzielnie przygotować w sposób zgodny oczekiwaniami stawianymi tworzonemu programowi.

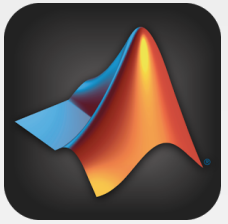
Opis oznaczeń w kodach źródłowych:

niebieski - słowa kluczowe
czerwony - nazwy funkcji
zielony - komentarze
czarny - pozostałe elementy kodu



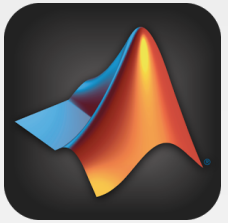
# Rasteryzacja: rysowanie odcinka (prosty algorytm)

```
function odcinek(xp,yp,xk,yk)

    % Funkcja realizująca prosty algorytm
    % rysowania odcinka

    dx = xk - xp;
    dy = yk - yp;

    a = dy/dx;
    b = yp - a*xp;

    if (abs(a)<1)
        for x = xp : xk
            y = round(a*x + b);
            set_pixel(x,y);
        end;
    else
        for y = yp : yk
            if (dx==0)
                x = xp;
            else
                x = round((y - b)/a);
            end;
            set_pixel(x,y);
        end;
    end;
```



**Argumentami wejściowymi funkcji są współrzędne początku $(x_p, y_p)$ i końca $(x_k, y_k)$ rysowanego odcinka.**

# Rasteryzacja: rysowanie odcinka (prosty algorytm)

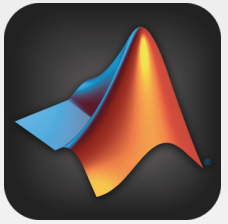


```
function odcinek(xp,yp,xk,yk)

    % Funkcja realizująca prosty algorytm
    % rysowania odcinka

    dx = xk - xp;
    dy = yk - yp;

    a = dy/dx;
    b = yp - a*xp;

    if (abs(a)<1)
        for x = xp : xk
            y = round(a*x + b);
            set_pixel(x,y);
        end;
    else
        for y = yp : yk
            if (dx==0)
                x = xp;
            else
                x = round((y - b)/a);
            end;
            set_pixel(x,y);
        end;
    end;
```

Wyznaczenie parametrów $a$ i $b$ prostej, na której leży odcinek na podstawie współrzędnych jego punktów początkowego i końcowego.

# Rasteryzacja: rysowanie odcinka (prosty algorytm)

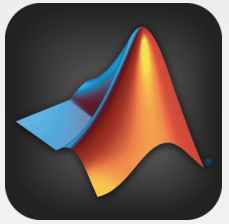
```
function odcinek(xp,yp,xk,yk)

    % Funkcja realizująca prosty algorytm
    % rysowania odcinka

    dx = xk - xp;
    dy = yk - yp;

    a = dy/dx;
    b = yp - a*xp;

    if (abs(a)<1)
        for x = xp : xk
            y = round(a*x + b);
            set_pixel(x,y);
        end;
    else
        for y = yp : yk
            if (dx==0)
                x = xp;
            else
                x = round((y - b)/a);
            end;
            set_pixel(x,y);
        end;
    end;
```



**Przypadek $|a| \leq 1$: w pętli dla $x = x_p, x_p+1, \ldots, x_k$ liczona jest wartość $y$ i wypełniany jest piksel.**

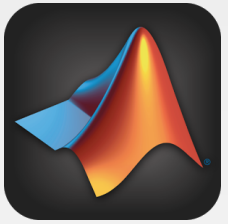
# Rasteryzacja: rysowanie odcinka (prosty algorytm)

```
function odcinek(xp,yp,xk,yk)

    % Funkcja realizująca prosty algorytm
    % rysowania odcinka

    dx = xk - xp;
    dy = yk - yp;

    a = dy/dx;
    b = yp - a*xp;

    if (abs(a)<1)
        for x = xp : xk
            y = round(a*x + b);
            set_pixel(x,y);
        end;
    else
        for y = yp : yk
            if (dx==0)
                x = xp;
            else
                x = round((y - b)/a);
            end;
            set_pixel(x,y);
        end;
    end;
```



**Przypadek $|a| > 1$: w pętli dla $y = y_p, y_p+1, \ldots, y_k$ liczona jest wartość $x$ i wypełniany jest piksel.**

# Rasteryzacja: rysowanie odcinka (prosty algorytm)

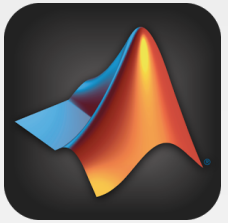


```
function odcinek(xp,yp,xk,yk)

    % Funkcja realizująca prosty algorytm
    % rysowania odcinka

    dx = xk - xp;
    dy = yk - yp;

    a = dy/dx;
    b = yp - a*xp;

    if (abs(a)<1)
        for x = xp : xk
            y = round(a*x + b);
            set_pixel(x,y);
        end;
    else
        for y = yp : yk
            if (dx==0)
                x = xp;
            else
                x = round((y - b)/a);
            end;
            set_pixel(x,y);
        end;
    end;
```

**Przypadek $|a| = \infty$ musi zostać rozpatrzony osobno.**

# Rasteryzacja: rysowanie odcinka (prosty algorytm)

```
function odcinek(xp,yp,xk,yk)

    % Funkcja realizująca prosty algorytm
    % rysowania odcinka

    dx = xk - xp;
    dy = yk - yp;

    a = dy/dx;
    b = yp - a*xp;

    if (abs(a)<1)
        for x = xp : xk
            y = round(a*x + b);
            set_pixel(x,y);
        end;
    else
        for y = yp : yk
            if (dx==0)
                x = xp;
            else
                x = round((y - b)/a);
            end;
            set_pixel(x,y);
        end;
    end;
```



**Algorytm jest bardzo prosty koncepcyjnie, ale nieefektywny:**

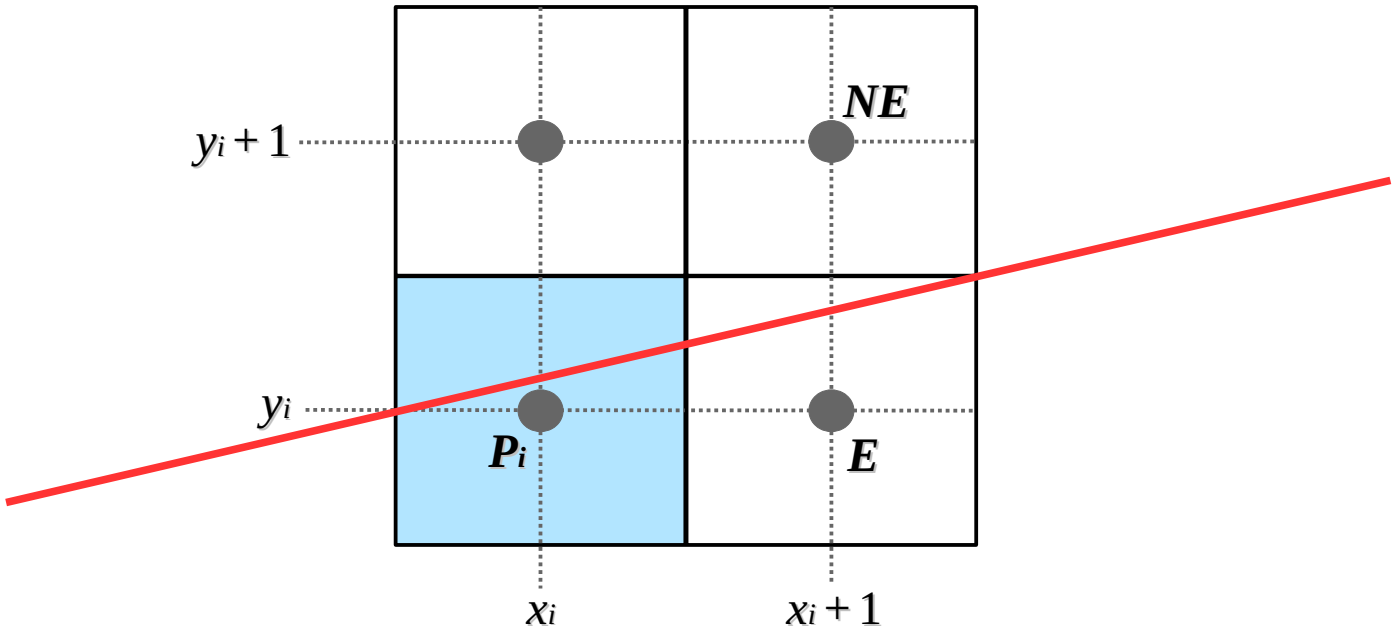
- **korzysta w trakcie obliczeń z liczb rzeczywistych;**
- **wymaga dużej liczby operacji w każdym kroku: po jednym mnożeniu i dodawaniu oraz zaokrąglanie (nie wliczając inkrementacji licznika pętli).**

# Rasteryzacja: rysowanie odcinka (algorytm Bresenhama)

**Algorytm Bresenhama** pozwala rysować odcinki w taki sposób, aby w każdym kroku używane było tylko **całkowitoliczbowe dodawanie i porównanie z zerem**.

Jego podstawą jest obserwacja, że w $i$-tym kroku, po narysowaniu piksela w pozycji $P_i$ o współrzędnych $(x_i, y_i)$, w następnym kroku możliwy jest jedynie wybór pomiędzy dwoma pikselami. **Dla odcinka o nachyleniu od $0^o$ do $45^o$ ($0 \le a \le 1$)** będą to:

- **piksel „wschodni” $E$ o współrzędnych $x_{i+1} = x_i + 1$ oraz $y_{i+1} = y_i$;**
- **piksel „północno-wschodni” $NE$ o współrzędnych $x_{i+1} = x_i + 1$ oraz $y_{i+1} = y_i + 1$.**

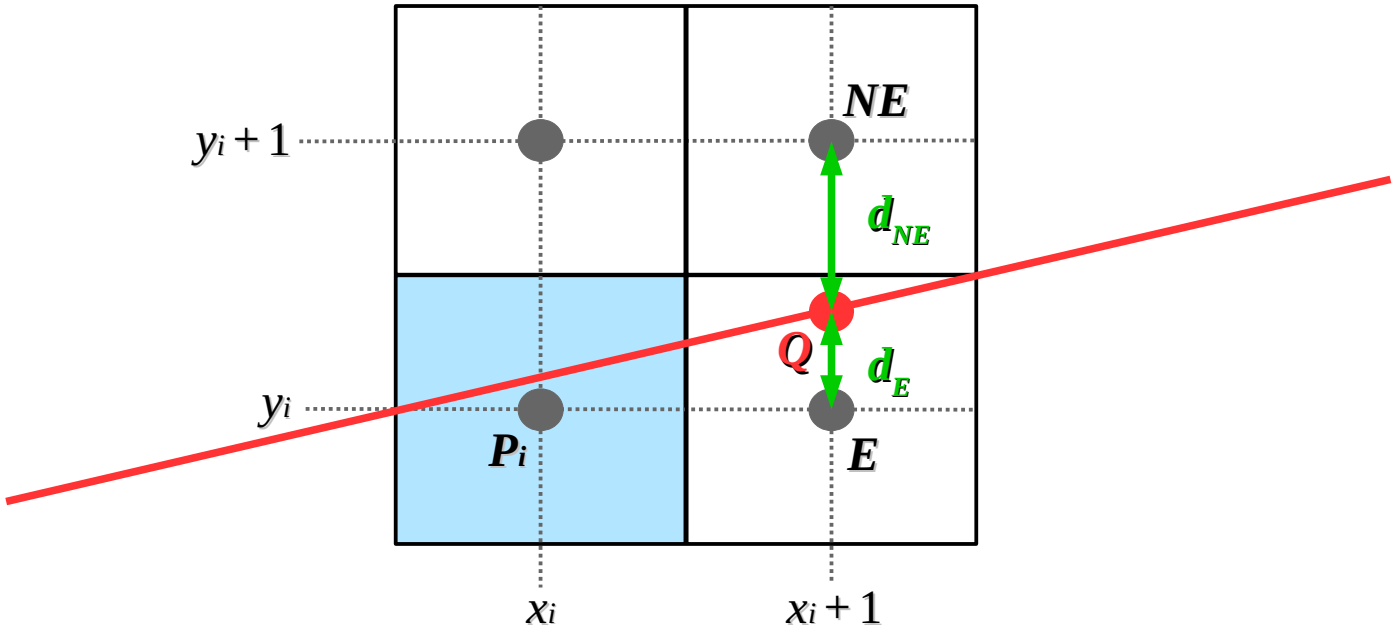


# Rasteryzacja: rysowanie odcinka (algorytm Bresenhama)

**Wybór pomiędzy tymi dwoma pikselami wynika z odległości pomiędzy ich środkami a leżącym na prostej punktem $Q$, a decyzję o tym, który z nich zostanie w następnym kroku wypełniony podejmuje się na podstawie znaku zmiennej $d_{i+1} = \Delta x \cdot (d_E - d_{NE})$:**

- **jeżeli $d_{i+1} < 0$ (czyli $d_E < d_{NE}$), to wypełniony zostanie piksel $E$;**
- **w przeciwnym razie, jeśli $d_{i+1} \geq 0$ (czyli $d_E \geq d_{NE}$), to wypełniony zostanie piksel $NE$.**

**Kryterium to jest równoważne minimalizacji błędu między rzeczywistym położeniem a środkiem piksela po rasteryzacji.**

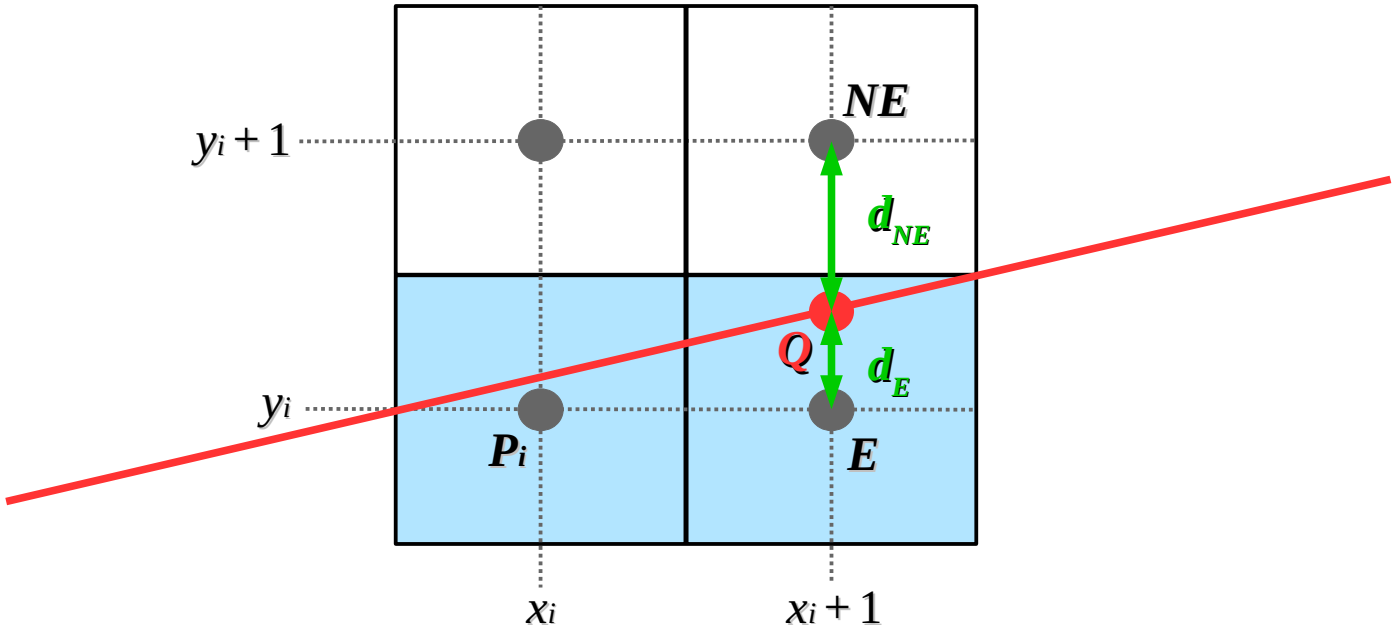


# Rasteryzacja: rysowanie odcinka (algorytm Bresenhama)

**Wybór pomiędzy tymi dwoma pikselami wynika z odległości pomiędzy ich środkami a leżącym na prostej punktem $Q$, a decyzję o tym, który z nich zostanie w następnym kroku wypełniony podejmuje się na podstawie znaku zmiennej $d_{i+1} = \Delta x \cdot (d_E - d_{NE})$:**

- **jeżeli $d_{i+1} < 0$ (czyli $d_E < d_{NE}$), to wypełniony zostanie piksel $E$;**
- **w przeciwnym razie, jeśli $d_{i+1} \geq 0$ (czyli $d_E \geq d_{NE}$), to wypełniony zostanie piksel $NE$.**

**Kryterium to jest równoważne minimalizacji błędu między rzeczywistym położeniem a środkiem piksela po rasteryzacji.**

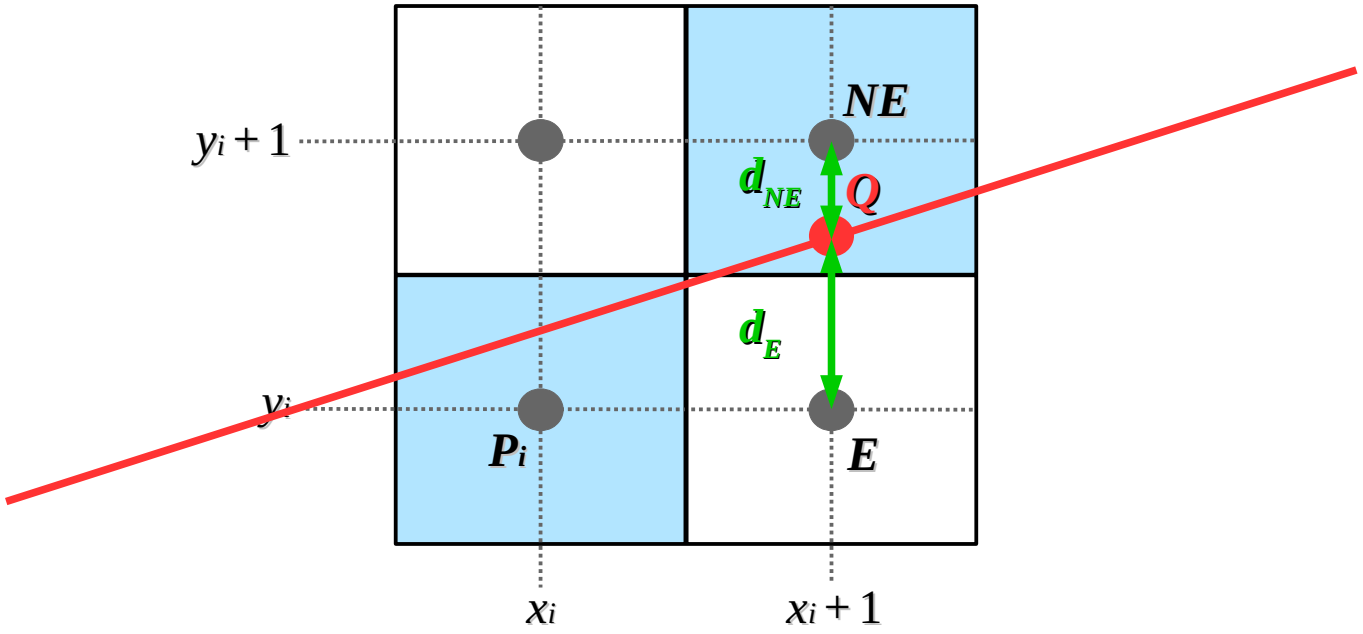


# Rasteryzacja: rysowanie odcinka (algorytm Bresenhama)

**Wybór pomiędzy tymi dwoma pikselami wynika z odległości pomiędzy ich środkami a leżącym na prostej punktem $Q$, a decyzję o tym, który z nich zostanie w następnym kroku wypełniony podejmuje się na podstawie znaku zmiennej $d_{i+1} = \Delta x \cdot (d_E - d_{NE})$:**

- **jeżeli $d_{i+1} < 0$ (czyli $d_E < d_{NE}$), to wypełniony zostanie piksel $E$;**
- **w przeciwnym razie, jeśli $d_{i+1} \geq 0$ (czyli $d_E \geq d_{NE}$), to wypełniony zostanie piksel $NE$.**

**Kryterium to jest równoważne minimalizacji błędu między rzeczywistym położeniem a środkiem piksela po rasteryzacji.**



# Rasteryzacja: rysowanie odcinka (algorytm Bresenhama)

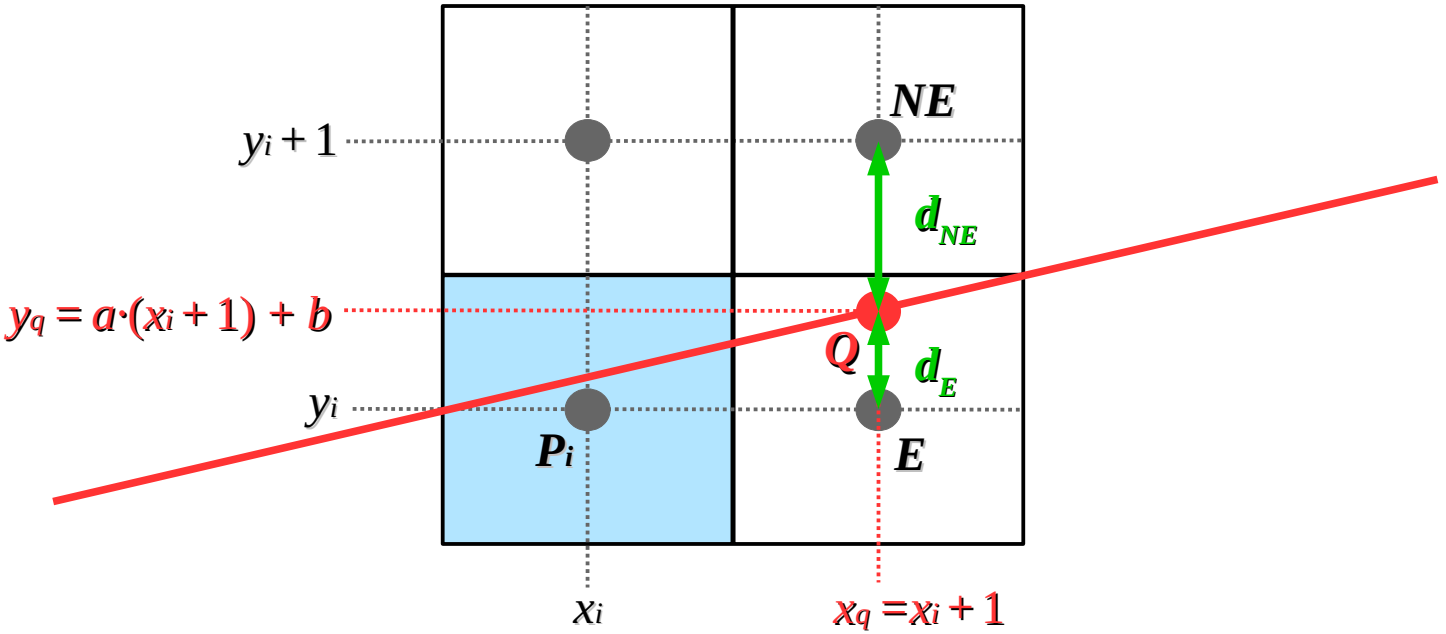
**Wyznaczając odległości:**

$$d_{NE} = y_{i+1} - y_q = y_i + 1 - a\cdot(x_i+1) + b = y_i + 1 - \frac{\Delta y}{\Delta x}\cdot(x_i + 1 - x_p) + y_p$$

$$d_E = y_q - y_i = a\cdot(x_i + 1) + b - y_i = \frac{\Delta y}{\Delta x}\cdot(x_i + 1 - x_p) + y_p - y_i + 1$$

**można określić postać zmiennej decyzyjnej w następnym kroku, zależną jedynie od parametrów odcinka i współrzędnych aktualnego piksela:**

$$d_{i+1} = \Delta x\cdot(d_E - d_{NE}) = 2\cdot\Delta y\cdot(x_i - x_p + 1) + \Delta x\cdot(2\cdot y_i + 2\cdot y_p + 1)$$



# Rasteryzacja: rysowanie odcinka (algorytm Bresenhama)

**Powtarzając takie same obliczenia dla obecnego kroku można wyznaczyć różnicę pomiędzy wartościami zmiennej decyzyjnej w dwóch kolejnych iteracjach:**

$$d_{i+1} - d_i = 2 \cdot \Delta x \cdot \left( y_{i-1} - y_i \right) + 2 \cdot \Delta y$$

**Powyższy wzór pokazuje, że zmienna decyzyjna jest modyfikowana przez dodanie wartości całkowitej złożonej z dwóch członów: jednego stałego ( $2 \cdot \Delta y$ ) i drugiego, zależnego od tego, który z pikseli został wybrany w poprzednim kroku ( $2 \cdot \Delta x \cdot (y_{i-1} - y_i)$ ). Prowadzi to do prostej reguły modyfikacji $d$ w kolejnych iteracji algorytmu:**

- **jeżeli w $i$-tym kroku $d_i < 0$ (w kroku $i-1$ wybrany został piksel $E$, czyli $y_i = y_{i-1}$), to w kroku $i+1$ zmienna $d_{i+1}$ wyniesie:**

$$d_{i+1} = d_i + 2 \cdot \Delta y$$

- **jeżeli w $i$-tym kroku $d_i \geq 0$ (w kroku $i-1$ wybrany został piksel $NE$, czyli $y_i = y_{i-1} + 1$), to w kroku $i+1$ zmienna $d_{i+1}$ wyniesie:**

$$d_{i+1} = d_i + 2 \cdot \left( \Delta y - \Delta x \right)$$

**Do pełnej definicji wymagane jest jeszcze podanie początkowej wartości $d$, którą można obliczyć na podstawie współrzędnych pierwszego punktu odcinka $(x_p, y_p)$:**

$$d_0 = 2 \cdot \Delta y - \Delta x$$

# Rasteryzacja: rysowanie odcinka (algorytm Bresenhama)

**Podsumowując, algorytm Bresenhama rysowania odcinka o nachyleniu od $0^o$ do $45^o$ ($0 \le a \le 1$) można zapisać w przedstawionych poniżej krokach.**

**1. Obliczenie wartości stałych pomocniczych oraz początkowej wartości zmiennej $d$ (wszystkie są liczbami całkowitymi):**

$$p_E = 2 \cdot \Delta y \qquad \Delta x = x_k - x_p$$

$$p_{NE} = 2 \cdot (\Delta y - \Delta x) \qquad \Delta y = y_k - y_p$$

$$d_i = d_0 = 2 \cdot \Delta y - \Delta x$$

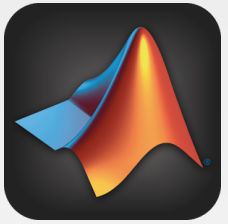
**2. Ustawienie wartości początkowej $y_i = y_p$.**

**3. W pętli dla każdego całkowitego $x_i \in (x_p, x_k)$:**

- **narysowanie piksela dla aktualnych wartości $(x_i, y_i)$.**
- **jeżeli $d_i < 0 \rightarrow$ wybieramy piksel $E$, czyli $y_i$ pozostaje bez zmian, zmienna decyzyjna dla następnego kroku jest obliczana jako $d_{i+1} = d_i + p_E$;**
- **jeżeli $d_i \ge 0 \rightarrow$ wybieramy piksel $NE$, czyli zwiększamy $y_i$ o $1$ oraz modyfikujemy zmienną decyzyjną dla następnego kroku zgodnie z zależnością $d_{i+1} = d_i + p_{NE}$;**

# Rasteryzacja: rysowanie odcinka (algorytm Bresenhama)

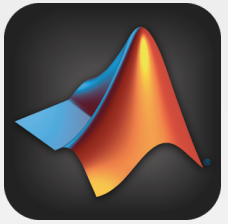
```
function odcinek_bresenham(xp,yp,xk,yk)

    % Funkcja realizująca algorytm Bresenhama rysowania
    % odcinka o nachyleniu (0,45] stopni

    dx = xk - xp;
    dy = yk - yp;

    pE = 2*dy;
    pNE = 2*(dy - dx);
    d = 2*dy - dx;

    y = yp;
    for x = xp : xk
         set_pixel(x,y);

        if (d<0)
            d = d + pE;
        else
            y = y + 1;
            d = d + pNE;
        end;
    end;
```



Taka implementacja algorytmu rysowania **używa jedynie liczb całkowitych** i **wymaga mniejszej liczby działań**: w każdym kroku wykonuje porównanie i jedno lub dwa dodawania (nie licząc inkrementacji $x$).

# Rasteryzacja: rysowanie odcinka (algorytm Bresenhama)

```
function odcinek_bresenham(xp,yp,xk,yk)

    % Funkcja realizująca algorytm Bresenhama rysowania
    % odcinka o nachyleniu (0,45] stopni

    dx = xk - xp;
    dy = yk - yp;

    pE = 2*dy;
    pNE = 2*(dy - dx);
    d = 2*dy - dx;

    y = yp;
    for x = xp : xk
         set_pixel(x,y);

        if (d<0)
            d = d + pE;
        else
            y = y + 1;
            d = d + pNE;
        end;
    end;
```



**Wyznaczenie stałych pomocniczych oraz początkowej wartości zmiennej decyzyjnej $d$.**

# Rasteryzacja: rysowanie odcinka (algorytm Bresenhama)

```
function odcinek_bresenham(xp,yp,xk,yk)

    % Funkcja realizująca algorytm Bresenhama rysowania
    % odcinka o nachyleniu (0,45] stopni

    dx = xk - xp;
    dy = yk - yp;

    pE = 2*dy;
    pNE = 2*(dy - dx);
    d = 2*dy - dx;

    y = yp;
    for x = xp : xk
         set_pixel(x,y);

        if (d<0)
            d = d + pE;
        else
            y = y + 1;
            d = d + pNE;
        end;
    end;
```

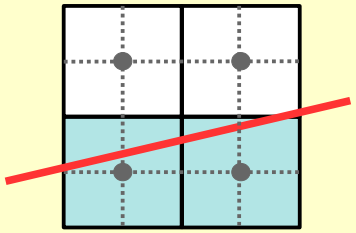
**Główna pętla algorytmu:** $x = x_p, x_p+1, \ldots, x_k$ **rysowany jest obecny piksel i wybierany następny.**

# Rasteryzacja: rysowanie odcinka (algorytm Bresenhama)

```
function odcinek_bresenham(xp,yp,xk,yk)

    % Funkcja realizująca algorytm Bresenhama rysowania
    % odcinka o nachyleniu (0,45] stopni

    dx = xk - xp;
    dy = yk - yp;

    pE = 2*dy;
    pNE = 2*(dy - dx);
    d = 2*dy - dx;

    y = yp;
    for x = xp : xk
         set_pixel(x,y);

        if (d<0)
            d = d + pE;
        else
            y = y + 1;
            d = d + pNE;
        end;
    end;
```

**Przypadek $d < 0$:**

- wybieramy piksel E;
- $y$ bez zmian;
- $d$ aktualizowane o $p_E$.

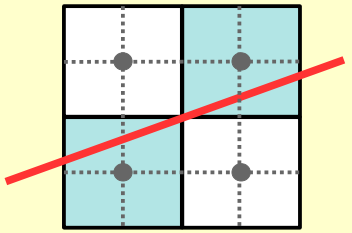


# Rasteryzacja: rysowanie odcinka (algorytm Bresenhama)

```
function odcinek_bresenham(xp,yp,xk,yk)

    % Funkcja realizująca algorytm Bresenhama rysowania
    % odcinka o nachyleniu (0,45] stopni

    dx = xk - xp;
    dy = yk - yp;

    pE = 2*dy;
    pNE = 2*(dy - dx);
    d = 2*dy - dx;

    y = yp;
    for x = xp : xk
         set_pixel(x,y);

        if (d<0)
            d = d + pE;
        else
            y = y + 1;
            d = d + pNE;
        end;
    end;
```

**Przypadek $d \geq 0$:**

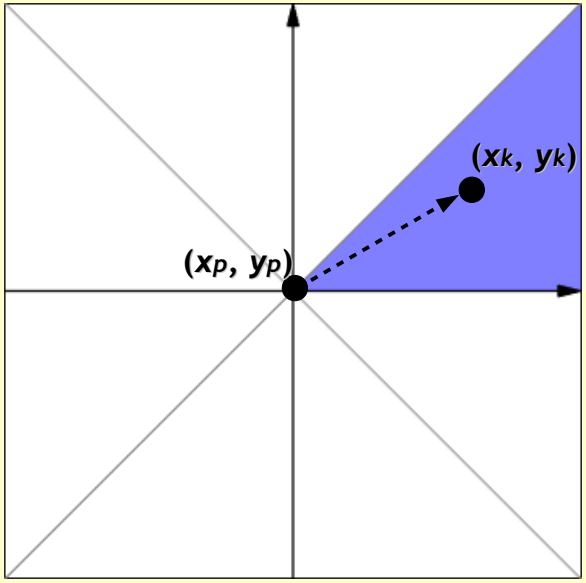
- wybieramy piksel NE;
- $y$ wzrasta o $1$;
- $d$ aktualizowane o $p_{NE}$.



# Rasteryzacja: rysowanie odcinka (algorytm Bresenhama)

```
function odcinek_bresenham(xp,yp,xk,yk)

    % Funkcja realizująca algorytm Bresenhama rysowania
    % odcinka o nachyleniu (0,45] stopni

    dx = xk - xp;
    dy = yk - yp;

    pE = 2*dy;
    pNE = 2*(dy - dx);
    d = 2*dy - dx;

    y = yp;
    for x = xp : xk
         set_pixel(x,y);

        if (d<0)
            d = d + pE;
        else
            y = y + 1;
            d = d + pNE;
        end;
    end;
```

**Wadą jest fakt, że ta implementacja pozwala jedynie rysować odcinki o nachyleniu od $0^o$ do $45^o$ ($0 \le a \le 1$). Ponadto zakłada, że $x_p \le x_k$ oraz $y_p \le y_k$.**

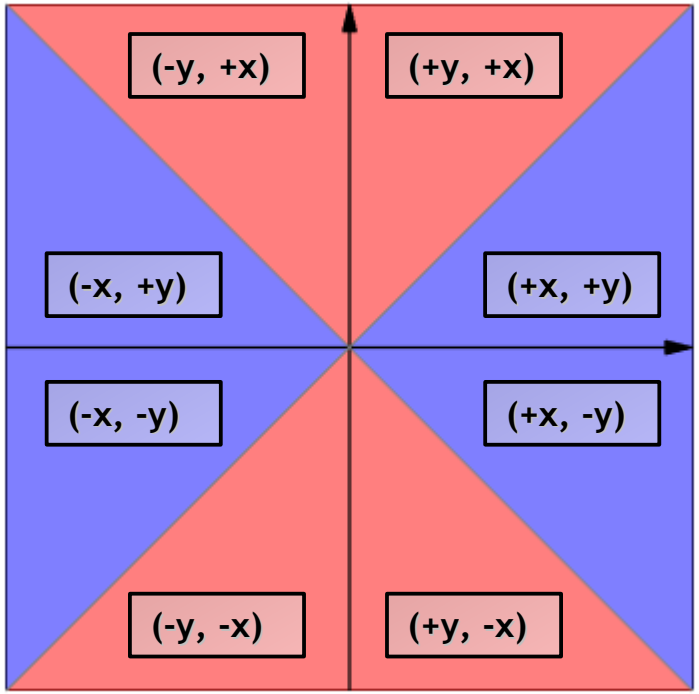


# Rasteryzacja: rysowanie odcinka (algorytm Bresenhama)

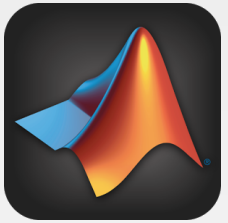
**Uwzględnienie wszystkich możliwych nachyleń wymaga wprowadzenia modyfikacji wynikających z symetrii wobec punktu początkowego odcinka:**

- **wszystkie odcinki o $|a| \leq 1$: zastąpienie $\Delta x$ i $\Delta y$ przez ich wartości bezwzględne oraz dopuszczenie możliwości zmniejszania wartości $x_i$ i $y_i$ o $1$ w kolejnych krokach;**

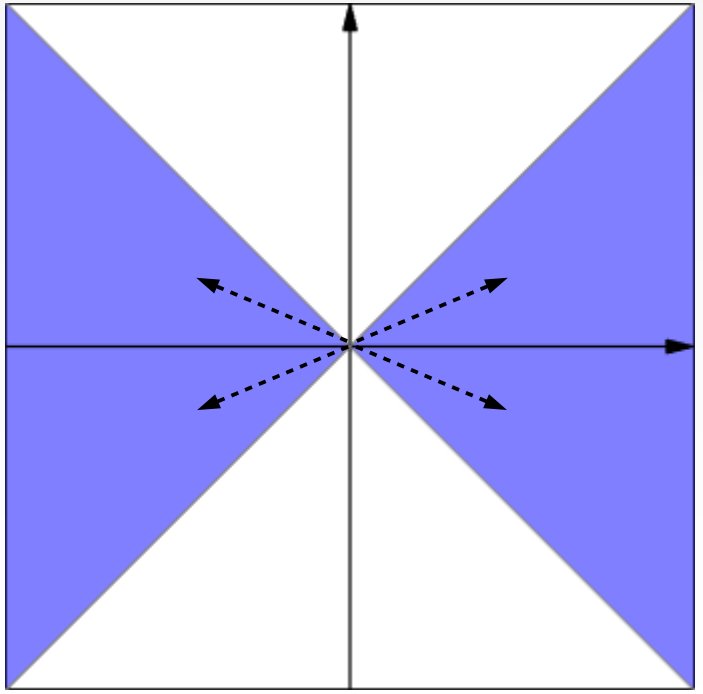
- **wszystkie odcinki o $|a| > 1$: to samo co powyżej oraz dodatkowo zamienienie rolami zmiennych $\Delta x$ i $\Delta y$ oraz $x_i$ i $y_i$ .**



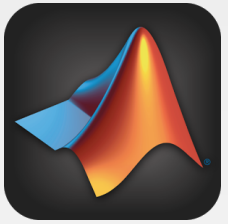
# Rasteryzacja: rysowanie odcinka (algorytm Bresenhama)



```
function odcinek_bresenham2(xp,yp,xk,yk)

    % Funkcja realizująca algorytm Bresenhama rysowania
    % odcinka o nachyleniach |a| <= 1

    dx = xk - xp;
    dy = yk - yp;

    sx = sign(dx);
    sy = sign(dy);

    dx = abs(dx);
    dy = abs(dy);

    pE = 2*dy;
    pNE = 2*(dy - dx);
    d = 2*dy - dx;

    x = xp;
    y = yp;
    for i = 0 : dx
        set_pixel(x,y);

        x = x + sx;

        if (d<0)
            d = d + pE;
        else
            y = y + sy;
            d = d + pNE;
        end;
    end;
```



# Rasteryzacja: rysowanie odcinka (algorytm Bresenhama)



```
function odcinek_bresenham2(xp,yp,xk,yk)

    % Funkcja realizująca algorytm Bresenhama rysowania
    % odcinka o nachyleniach |a| <= 1

    dx = xk - xp;
    dy = yk - yp;

    sx = sign(dx);
    sy = sign(dy);

    dx = abs(dx);
    dy = abs(dy);

    pE = 2*dy;
    pNE = 2*(dy - dx);
    d = 2*dy - dx;

    x = xp;
    y = yp;
    for i = 0 : dx
        set_pixel(x,y);

        x = x + sx;

        if (d<0)
            d = d + pE;
        else
            y = y + sy;
            d = d + pNE;
        end;
    end;
```

$\Delta x$ i $\Delta y$ zastępujemy ich wartościami bezwzględnymi.

# Rasteryzacja: rysowanie odcinka (algorytm Bresenhama)

```
function odcinek_bresenham2(xp,yp,xk,yk)

    % Funkcja realizująca algorytm Bresenhama rysowania
    % odcinka o nachyleniach |a| <= 1

    dx = xk - xp;
    dy = yk - yp;

    sx = sign(dx);
    sy = sign(dy);

    dx = abs(dx);
    dy = abs(dy);

    pE = 2*dy;
    pNE = 2*(dy - dx);
    d = 2*dy - dx;

    x = xp;
    y = yp;
    for i = 0 : dx
        set_pixel(x,y);

        x = x + sx;

        if (d<0)
            d = d + pE;
        else
            y = y + sy;
            d = d + pNE;
        end;
    end;
```

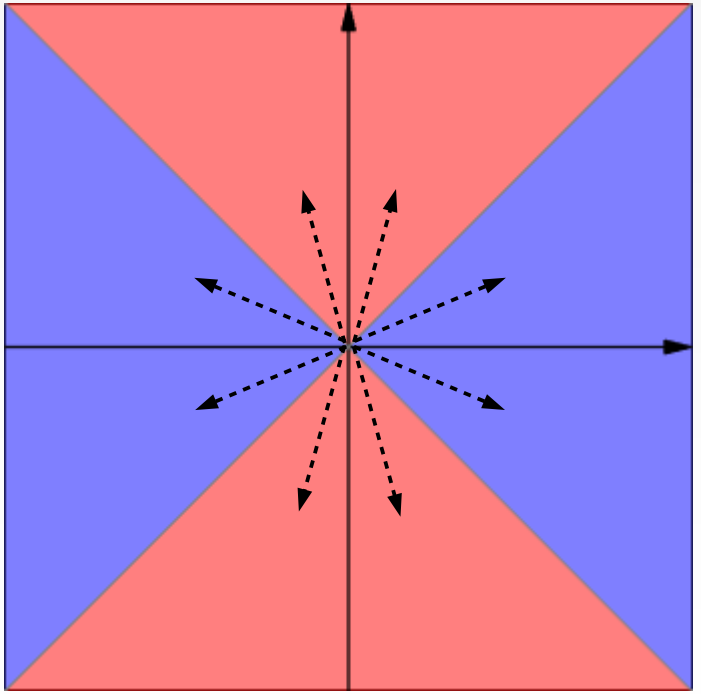
Współrzędne $x$ i $y$ będą w głównej pętli modyfikowane o $1$ lub $-1$, w zależności od znaków $\Delta x$ i $\Delta y$.

Funkcja sign jest równoważne warunkowi:

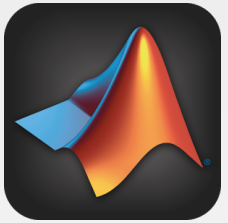
```
if (dx>=0)
    sx=1;
else
    sx=-1;
end;
```

# Rasteryzacja: rysowanie odcinka (algorytm Bresenhama)

```
function odcinek_bresenham3(xp,yp,xk,yk)

    % Funkcja realizująca algorytm Bresenhama rysowania
    % odcinka o dowolnym nachyleniu

    dx = xk - xp;
    dy = yk - yp;

    sx = sign(dx);
    sy = sign(dy);

    dx = abs(dx);
    dy = abs(dy);

    if (abs(dy/dx)<1)
        …
    else
        pN = 2*dx;
        pNE = 2*(dx - dy);
        d = 2*dx - dy;

        x = xp;
        y = yp;
        for i = 0 : dy
            set_pixel(x,y);

            y = y + sy;

            if (d<0)
                d = d + pN;
            else
                x = x + sx;
                d = d + pNE;
            end;
        end;
    end;
```



# Rasteryzacja: rysowanie odcinka (algorytm Bresenhama)



```
function odcinek_bresenham3(xp,yp,xk,yk)

    % Funkcja realizująca algorytm Bresenhama rysowania
    % odcinka o dowolnym nachyleniu

    dx = xk - xp;
    dy = yk - yp;

    sx = sign(dx);
    sy = sign(dy);

    dx = abs(dx);
    dy = abs(dy);

    if (abs(dy/dx)<1)
        …
    else
        pN = 2*dx;
        pNE = 2*(dx - dy);
        d = 2*dx - dy;

        x = xp;
        y = yp;
        for i = 0 : dy
            set_pixel(x,y);

            y = y + sy;

            if (d<0)
                d = d + pN;
            else
                x = x + sx;
                d = d + pNE;
            end;
        end;
    end;
```

Przypadek $|a| \leq 1$: kod pozostaje bez zmian.

# Rasteryzacja: rysowanie odcinka (algorytm Bresenhama)

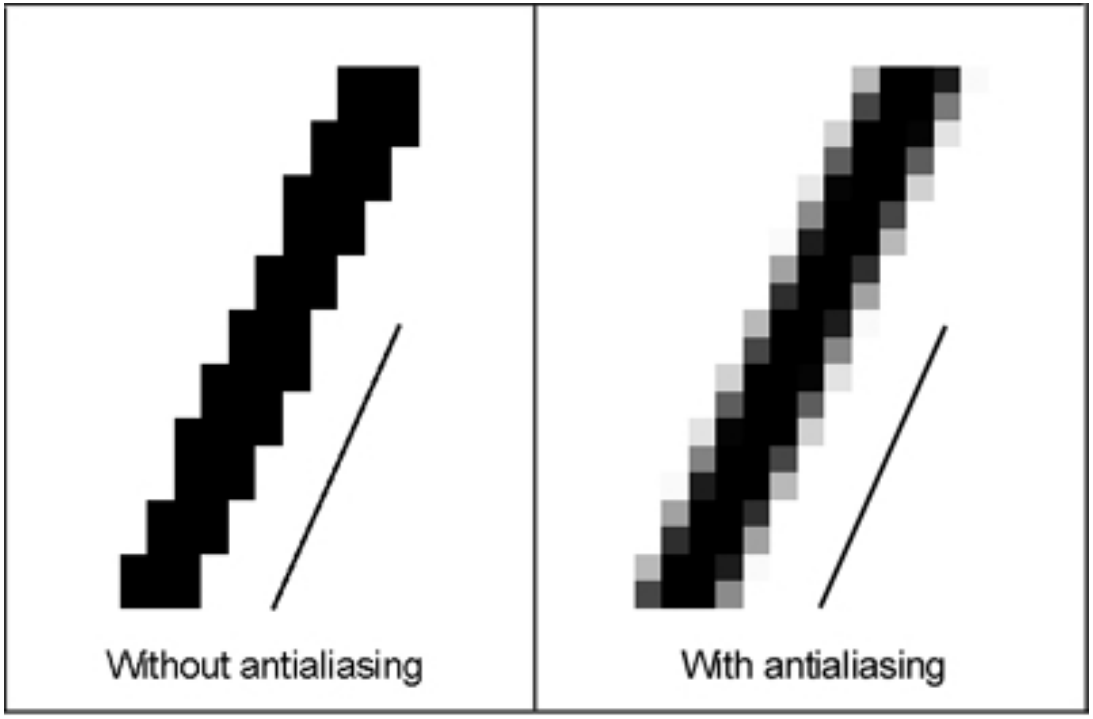
```
function odcinek_bresenham3(xp,yp,xk,yk)

    % Funkcja realizująca algorytm Bresenhama rysowania
    % odcinka o dowolnym nachyleniu

    dx = xk - xp;
    dy = yk - yp;

    sx = sign(dx);
    sy = sign(dy);

    dx = abs(dx);
    dy = abs(dy);

    if (abs(dy/dx)<1)
        …
    else
        pN = 2*dx;
        pNE = 2*(dx - dy);
        d = 2*dx - dy;

        x = xp;
        y = yp;
        for i = 0 : dy
            set_pixel(x,y);

            y = y + sy;

            if (d<0)
                d = d + pN;
            else
                x = x + sx;
                d = d + pNE;
            end;
        end;
    end;
```

**Przypadek $|a| > 1$: zamiana ról $\Delta x$ i $\Delta y$ oraz $xi$ i $yi$ wobec kodu dla przypadku $|a| \leq 1$.**

# Rasteryzacja: rysowanie odcinka (aliasing)

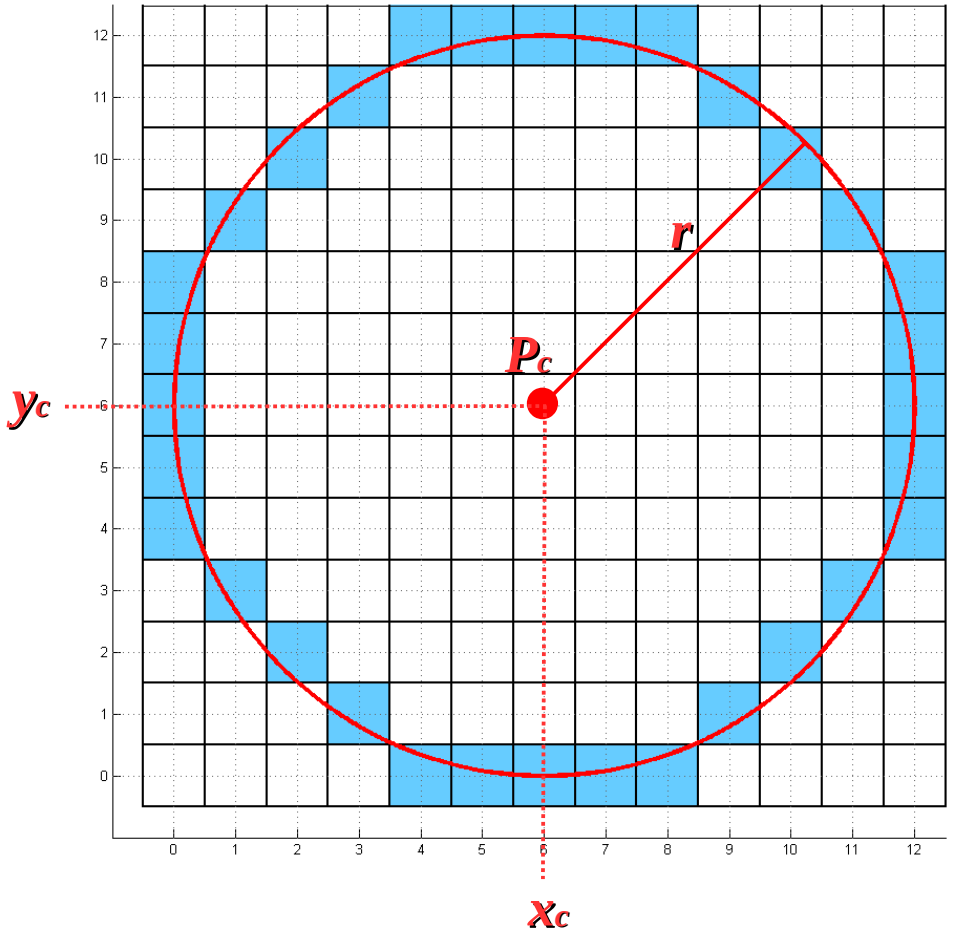
**Naturalną konsekwencją skończonej rozdzielczości rastra jest „schodkowy” wygląd rastrowej reprezentacji odcinka (a w ogólności – każdej krzywej poddanej procesowi rasteryzacji). Ponadto wzór ułożenia pikseli tworzących odcinek będzie się zmieniał wraz ze zmianą jego nachylenia, co dodatkowo może pogorszyć jakość wizualizacji.**

**Jakkolwiek efekt ten (nazywany aliasingiem), staje się mniej zauważalny wraz ze wzrostem rozdzielczości, to jednak zawsze występuje i nie może być w pełni usunięty żadną metodą. Można jednak zmniejszać jego widoczność przy użyciu technik antyaliasingu (wygładzania krawędzi).**



# Rasteryzacja: rysowanie okręgu

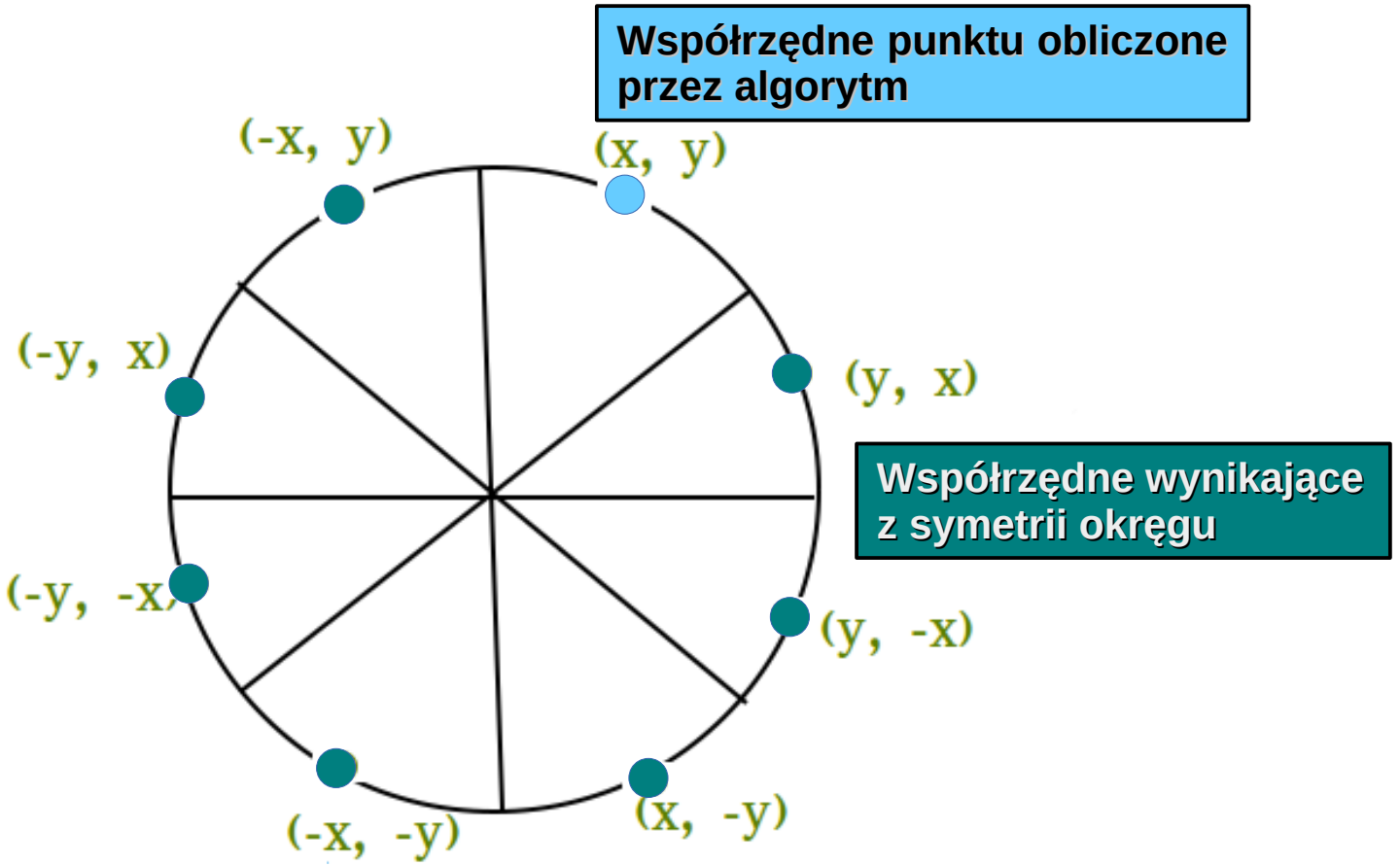
**Zadanie polega na wyznaczeniu pikseli będących wizualizacją okręgu o środku w punkcie $P_c$ o współrzędnych $(x_c, y_c)$ i promieniu $r$.**




**Ogólne równanie okręgu to $(x - x_c)^2 + (y - y_c)^2 = r^2$, ale bez utraty ogólności można ograniczyć opis algorytmu do przypadku okręgu o środku w punkcie $(x_c = 0, y_c = 0)$, dla którego równanie upraszcza się do postaci: $x^2 + y^2 = r^2$.**

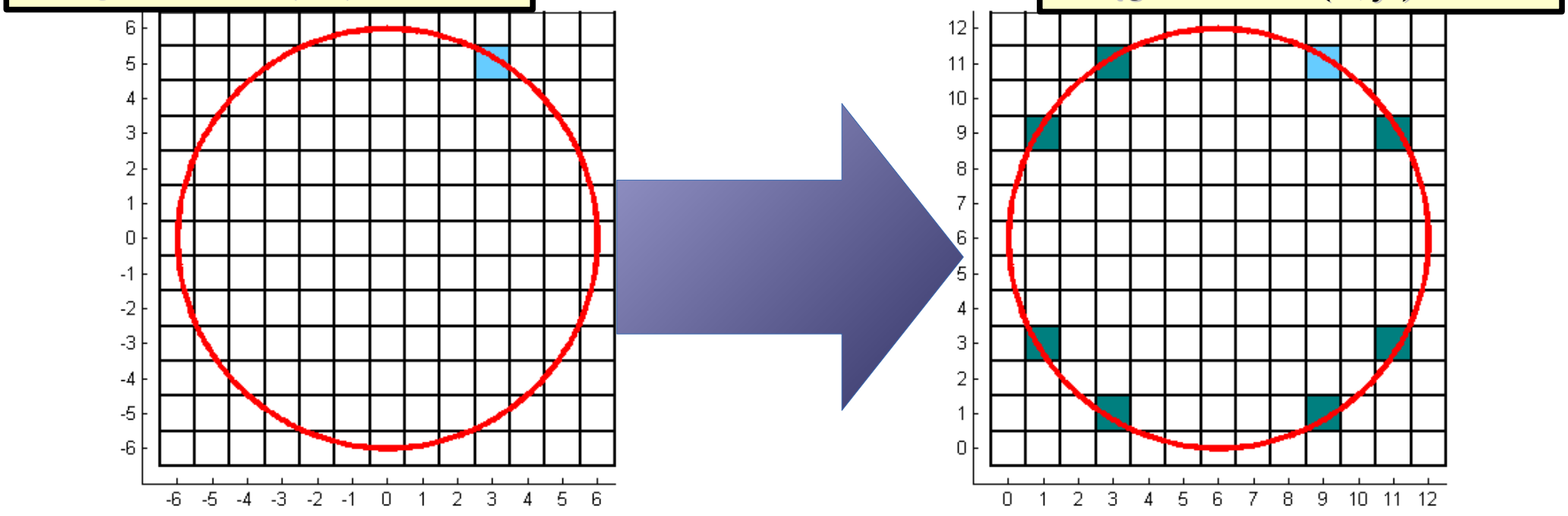
# Rasteryzacja: rysowanie okręgu

**Niezależnie od używanego algorytmu, podczas wyznaczania położeń pikseli okręgu warto korzystać z symetrii. Dzięki temu możliwe jest ograniczenie obliczeń jedynie do punktów z 1/8 okręgu (z jednego oktantu), a współrzędne pozostałych określać w sposób widoczny na poniższym rysunku.**



# Rasteryzacja: rysowanie okręgu

```
function set_circle_point(x,y,xc,yc)

    % Funkcja rysująca 8 pikseli okręgu o środku (xc,yc) na podstawie
    % współrzędnych wyznaczonych dla jednego punktu okręgu o środku (0,0)

    set_pixel( x + xc,  y + yc);
    set_pixel(-x + xc,  y + yc);
    set_pixel( x + xc, -y + yc);
    set_pixel(-x + xc, -y + yc);
    set_pixel( y + xc,  x + yc);
    set_pixel(-y + xc,  x + yc);
    set_pixel( y + xc, -x + yc);
    set_pixel(-y + xc, -x + yc);
```

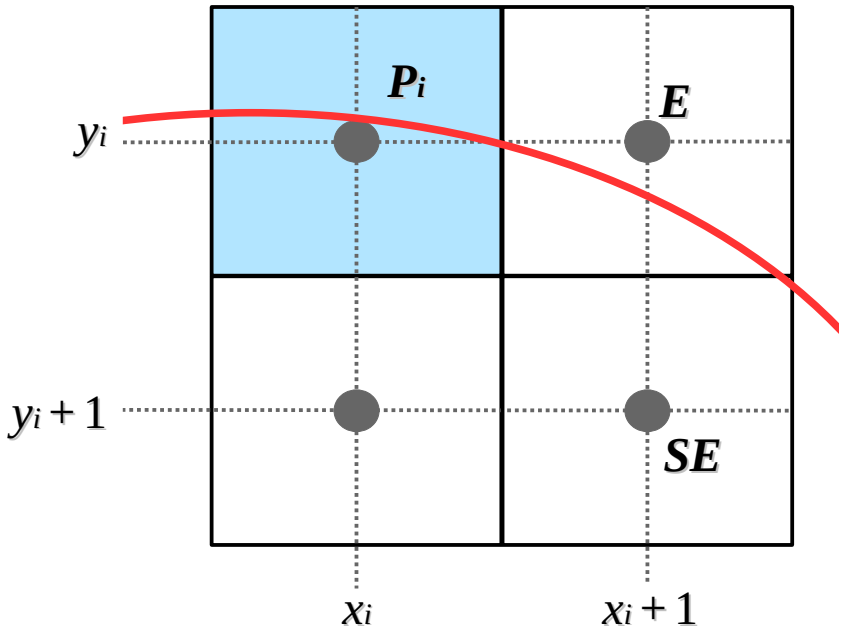


# Rasteryzacja: rysowanie okręgu (algorytm z punktem środkowym)

**Podobnie jak dla odcinka, również w przypadku okręgu wyznaczanie położeń pikseli bezpośrednio z definicji prowadziłoby do nieefektywnego algorytmu. Dlatego też w praktyce stosuje się inne podejścia, np. algorytm z punktem środkowym (*midpoint*):**

**W $i$-tym kroku algorytmu, po ustaleniu piksela pierwszego oktantu w pozycji $P_i$ o współrzędnych $(x_i, y_i)$, w następnym kroku możliwy jest jedynie wybór pomiędzy:**

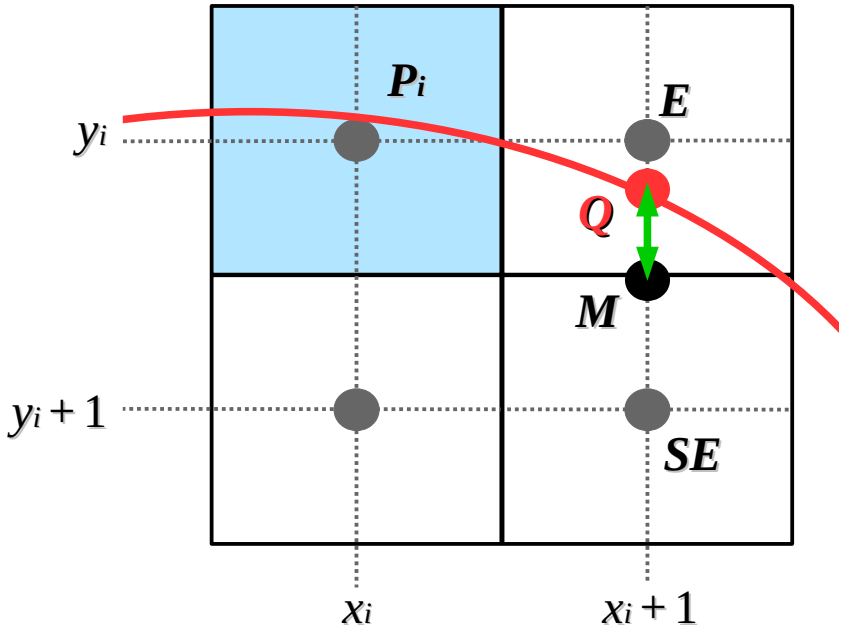
- **pikselem „wschodnim” $E$ o współrzędnych $x_{i+1} = x_i + 1$ oraz $y_{i+1} = y_i$;**
- **pikselem „południowo-wschodnim” $SE$ o współrzędnych $x_{i+1} = x_i + 1$ oraz $y_{i+1} = y_i - 1$.**



# Rasteryzacja: rysowanie okręgu (algorytm z punktem środkowym)

Wyboru pomiędzy tymi dwiema możliwościami dokonuje się analizując położenie względem okręgu punktu środkowego $M$, znajdującego się w połowie odcinka łączącego rozpatrywane piksele. Zależnie od tego czy punkt ten leży wewnątrz czy na zewnątrz okręgu, należy wybrać odpowiednio górny albo dolny piksel:

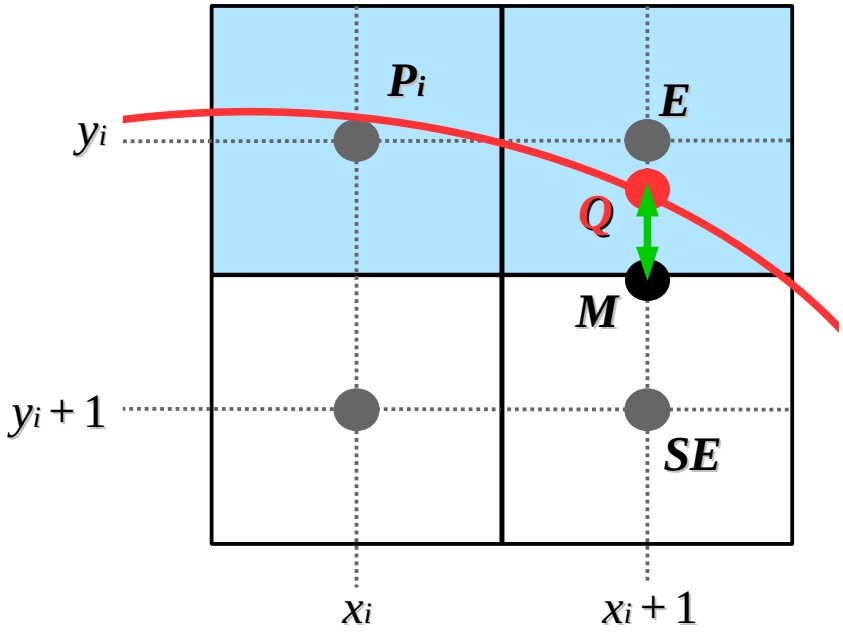
- jeżeli różnica pomiędzy współrzędnymi y punktu $M$ i punktu $Q$ leżącego na okręgu jest mniejsza od $0$, to $M$ leży wewnątrz okręgu i wypełniony zostanie piksel $E$;
- w przeciwnym przypadku, gdy różnica nie jest mniejsza od $0$, to punkt $M$ leży na zewnątrz okręgu i wypełniony zostanie piksel $SE$.



# Rasteryzacja: rysowanie okręgu (algorytm z punktem środkowym)

Wyboru pomiędzy tymi dwiema możliwościami dokonuje się analizując położenie względem okręgu punktu środkowego $M$, znajdującego się w połowie odcinka łączącego rozpatrywane piksele. Zależnie od tego czy punkt ten leży wewnątrz czy na zewnątrz okręgu, należy wybrać odpowiednio górny albo dolny piksel:

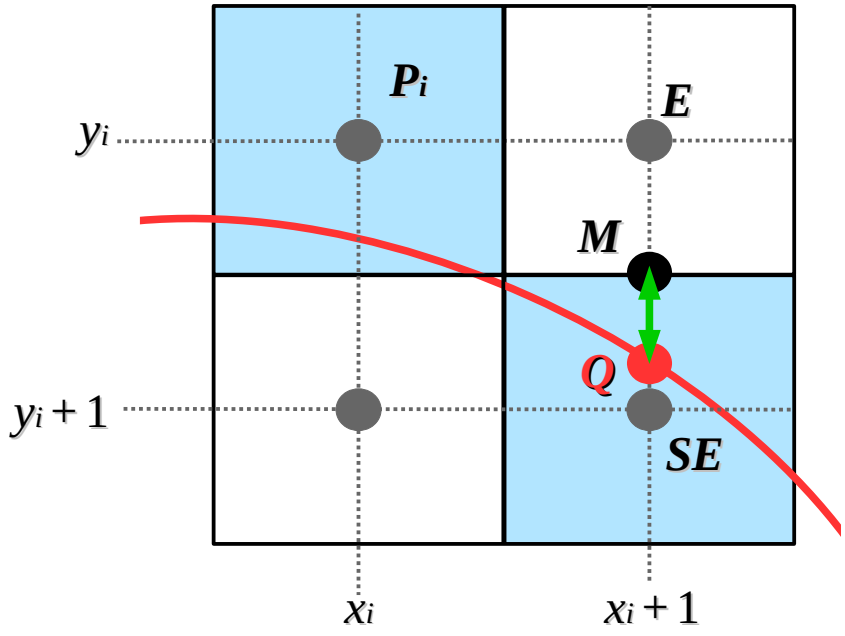
- jeżeli różnica pomiędzy współrzędnymi y punktu $M$ i punktu $Q$ leżącego na okręgu jest mniejsza od $0$, to $M$ leży wewnątrz okręgu i wypełniony zostanie piksel $E$;
- w przeciwnym przypadku, gdy różnica nie jest mniejsza od $0$, to punkt $M$ leży na zewnątrz okręgu i wypełniony zostanie piksel $SE$.



# Rasteryzacja: rysowanie okręgu (algorytm z punktem środkowym)

Wyboru pomiędzy tymi dwiema możliwościami dokonuje się analizując położenie względem okręgu punktu środkowego $M$, znajdującego się w połowie odcinka łączącego rozpatrywane piksele. Zależnie od tego czy punkt ten leży wewnątrz czy na zewnątrz okręgu, należy wybrać odpowiednio górny albo dolny piksel:

- jeżeli różnica pomiędzy współrzędnymi y punktu $M$ i punktu $Q$ leżącego na okręgu jest mniejsza od $0$, to $M$ leży wewnątrz okręgu i wypełniony zostanie piksel $E$;
- w przeciwnym przypadku, gdy różnica nie jest mniejsza od $0$, to punkt $M$ leży na zewnątrz okręgu i wypełniony zostanie piksel $SE$.



# Rasteryzacja: rysowanie okręgu (algorytm z punktem środkowym)

**Ostatecznie, po dokonaniu obliczeń podobnych, do tych pokazanych już wcześniej dla odcinka, algorytm rysowania okręgu można zapisać w poniższy sposób.**

**1. Obliczenie początkowej wartości zmiennej decyzyjnej $d$:**

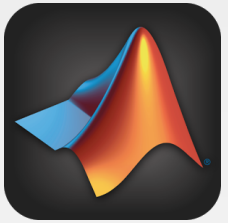
$$d_i = d_0 = 5 - 4 \cdot r$$

**2. Ustawienie współrzędnych $(x_i = 0, y_i = r)$.**

**3. Powtarzanie w pętli dopóki spełniony jest warunek $x_i \leq y_i$:**

- **narysowanie piksela dla aktualnych wartości $(x_i, y_i)$.**
- **jeżeli $d_i < 0 \rightarrow$ wybieramy piksel $E$, czyli $y_i$ pozostaje bez zmian, zmienna decyzyjna dla następnego kroku jest obliczana jako $d_{i+1} = d_i + 8 \cdot x_i + 12$;**
- **jeżeli $d_i \geq 0 \rightarrow$ wybieramy piksel $SE$, czyli zmniejszamy $y_i$ o $1$ oraz modyfikujemy zmienną decyzyjną zgodnie z zależnością $d_{i+1} = d_i + 8 \cdot (x_i - y_i) + 20$;**

# Rasteryzacja: rysowanie okręgu (algorytm z punktem środkowym)



```
function okrag_midpoint(xc,yc,r)

    % Funkcja realizująca algorytm midpoint rysowania
    % okręgu o środku (xc,yc) i promieniu r

    d = 5 - 4 * r;

    x = 0;
    y = r;

    while (x <= y)
        set_circle_point(x,y,xc,yc);

        if (d<0)
            d = d + 8 * x + 12;
        else
            d = d + 8 * (x - y) + 20;
            y = y - 1;
        end;
        x = x + 1;
    end;
```

# GRAFIKA KOMPUTEROWA

## Wykład 3: podstawowe algorytmy grafiki dwuwymiarowej (wypełnianie obszarów)

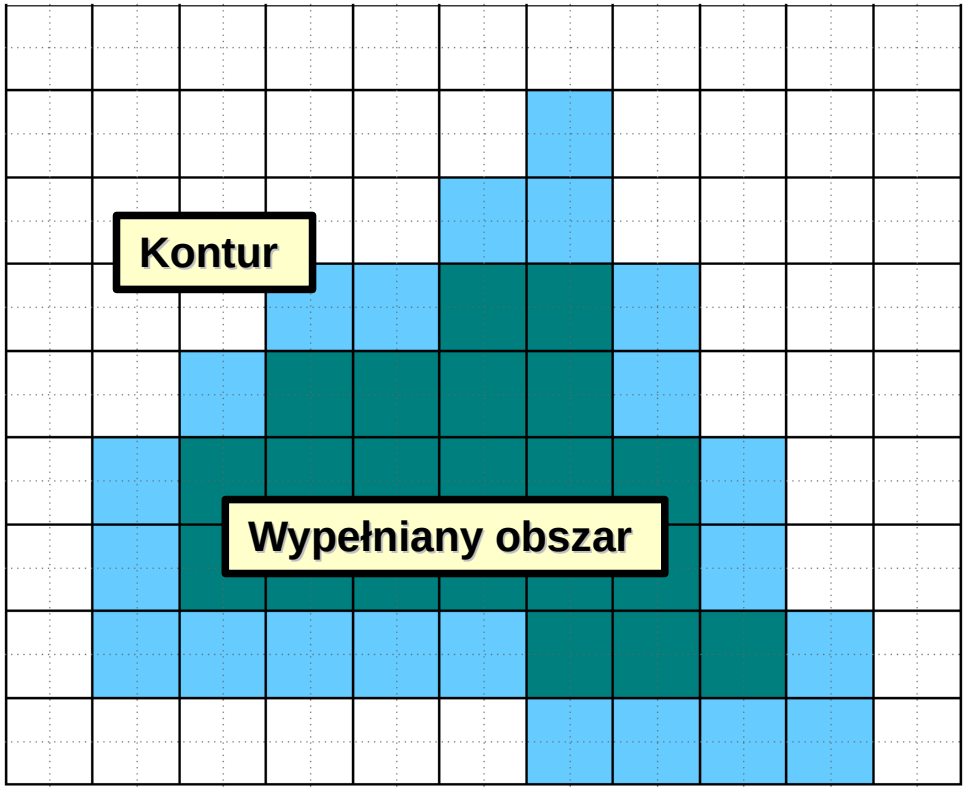
**Tymon Rubel**

**Zakład Elektroniki Jądrowej i Medycznej**
**Instytut Radioelektroniki i Technik Multimedialnych PW**

# Podstawowe algorytmy grafiki dwuwymiarowej: wypełnianie obszarów

**Wypełnianie obszarów** znajdujących się wewnątrz pewnego konturu jest drugim, po rasteryzacji, zagadnieniem o podstawowym znaczeniu dla grafiki dwuwymiarowej.

Dla nieskomplikowanych konturów (np. prostokątnych) problem wypełniania jest prosty do rozwiązania. Jednak w ogólności, dla figur o dowolnym kształcie, zarówno wypukłych, jak i wklęsłych, zadanie to może być nietrywialne.



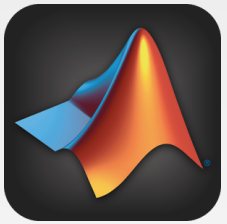

Wybór algorytmu używanego do wypełniania w znacznej mierze zależy od sposobu w jaki definiowany jest kontur obszaru.

# Wypełnianie obszarów: wypełnianie przez spójność

W przypadku **konturu opisanego rastrowo** (czyli poprzez zbiór pikseli o jednakowej barwie) można użyć **algorytmu wypełniania przez spójność**, który zaczyna działanie od zadanego punktu startowego (**ziarna**) wewnątrz obszaru.

**Procedura jest rekurencyjna**: po wypełnieniu ziarna jest ona wywoływana dla pikseli z sąsiedztwa, które stają się nowymi ziarnami. Warunkiem przerwania rekurencji jest sytuacja, w której obecny kolor piksela jest równy kolorowi konturu lub wypełnienia.

```
function flood_fill(x,y,ck,cw)

    %Funkcja wypełniania obszaru przez spójność

    c=get_color(x,y);

    if (compare_color(c,ck)==false && compare_color(c,cw)==false)

        set_pixel(x,y,cw);

        flood_fill(x+1,y  ,ck,cw);
        flood_fill(x-1,y  ,ck,cw);
        flood_fill(x  ,y+1,ck,cw);
        flood_fill(x  ,y-1,ck,cw);
    end;
end;
```

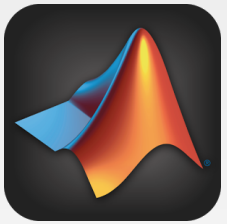


# Wypełnianie obszarów: wypełnianie przez spójność

W przypadku **konturu opisanego rastrowo** (czyli poprzez zbiór pikseli o jednakowej barwie) można użyć **algorytmu wypełniania przez spójność**, który zaczyna działanie od zadanego punktu startowego (**ziarna**) wewnątrz obszaru.

**Procedura jest rekurencyjna**: po wypełnieniu ziarna jest ona wywoływana dla pikseli z sąsiedztwa, które stają się nowymi ziarnami. Warunkiem przerwania rekurencji jest sytuacja, w której obecny kolor piksela jest równy kolorowi konturu lub wypełnienia.

```
function flood_fill(x,y,ck,cw)

    %Funkcja wypełniania obszaru przez spójność

    c=get_color(x,y);

    if (compare_color(c,ck)==false && compare_color(c,cw)==false)

        set_pixel(x,y,cw);

        flood_fill(x+1,y  ,ck,cw);
        flood_fill(x-1,y  ,ck,cw);
        flood_fill(x  ,y+1,ck,cw);
        flood_fill(x  ,y-1,ck,cw);
    end;
end;
```



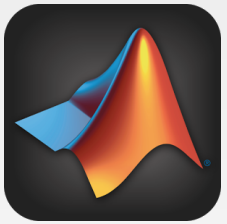
Argumentami funkcji są współrzędne piksela $(x, y)$ oraz kolory konturu $C_k$ i wypełnienia $C_w$.

# Wypełnianie obszarów: wypełnianie przez spójność

W przypadku **konturu opisanego rastrowo** (czyli poprzez zbiór pikseli o jednakowej barwie) można użyć **algorytmu wypełniania przez spójność**, który zaczyna działanie od zadanego punktu startowego (**ziarna**) wewnątrz obszaru.

**Procedura jest rekurencyjna**: po wypełnieniu ziarna jest ona wywoływana dla pikseli z sąsiedztwa, które stają się nowymi ziarnami. Warunkiem przerwania rekurencji jest sytuacja, w której obecny kolor piksela jest równy kolorowi konturu lub wypełnienia.

```
function flood_fill(x,y,ck,cw)

    %Funkcja wypełniania obszaru przez spójność

    c=get_color(x,y);

    if (compare_color(c,ck)==false && compare_color(c,cw)==false)

        set_pixel(x,y,cw);

        flood_fill(x+1,y  ,ck,cw);
        flood_fill(x-1,y  ,ck,cw);
        flood_fill(x  ,y+1,ck,cw);
        flood_fill(x  ,y-1,ck,cw);
    end;
end;
```



Pobranie koloru piksela o współrzędnych $(x, y)$.

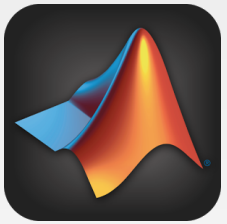
Uwaga: *get_color* nie jest standardową funkcją Matlaba. Ta sama uwaga dotyczy także funkcji *compare_color*.

# Wypełnianie obszarów: wypełnianie przez spójność

W przypadku **konturu opisanego rastrowo** (czyli poprzez zbiór pikseli o jednakowej barwie) można użyć **algorytmu wypełniania przez spójność**, który zaczyna działanie od zadanego punktu startowego (**ziarna**) wewnątrz obszaru.

**Procedura jest rekurencyjna**: po wypełnieniu ziarna jest ona wywoływana dla pikseli z sąsiedztwa, które stają się nowymi ziarnami. Warunkiem przerwania rekurencji jest sytuacja, w której obecny kolor piksela jest równy kolorowi konturu lub wypełnienia.

```
function flood_fill(x,y,ck,cw)

    %Funkcja wypełniania obszaru przez spójność

    c=get_color(x,y);

    if (compare_color(c,ck)==false && compare_color(c,cw)==false)

        set_pixel(x,y,cw);

        flood_fill(x+1,y  ,ck,cw);
        flood_fill(x-1,y  ,ck,cw);
        flood_fill(x  ,y+1,ck,cw);
        flood_fill(x  ,y-1,ck,cw);
    end;
end;
```



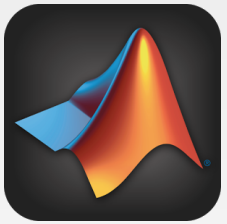
Porównanie koloru z konturem i wypełnieniem.

# Wypełnianie obszarów: wypełnianie przez spójność

W przypadku **konturu opisanego rastrowo** (czyli poprzez zbiór pikseli o jednakowej barwie) można użyć **algorytmu wypełniania przez spójność**, który zaczyna działanie od zadanego punktu startowego (**ziarna**) wewnątrz obszaru.

**Procedura jest rekurencyjna**: po wypełnieniu ziarna jest ona wywoływana dla pikseli z sąsiedztwa, które stają się nowymi ziarnami. Warunkiem przerwania rekurencji jest sytuacja, w której obecny kolor piksela jest równy kolorowi konturu lub wypełnienia.

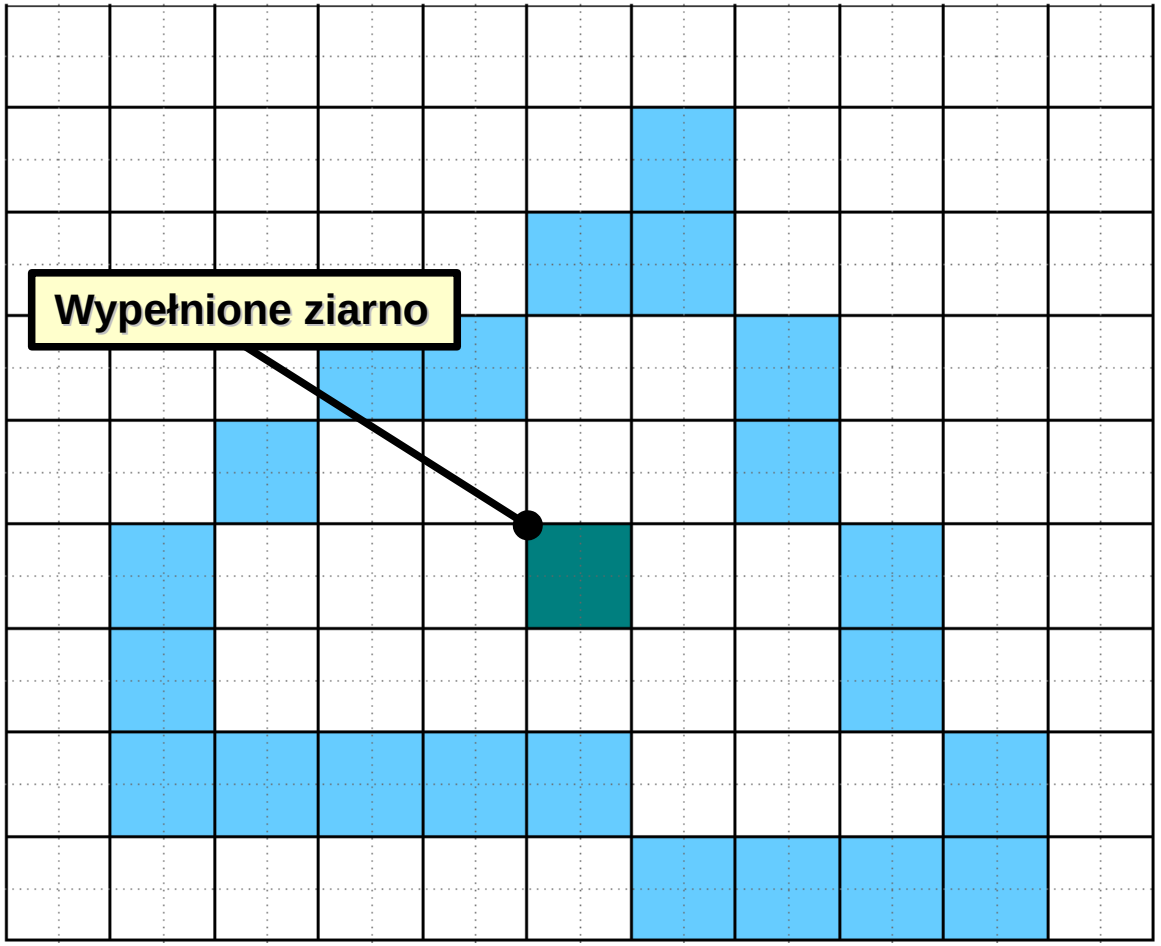
```
function flood_fill(x,y,ck,cw)

    %Funkcja wypełniania obszaru przez spójność

    c=get_color(x,y);

    if (compare_color(c,ck)==false && compare_color(c,cw)==false)

        set_pixel(x,y,cw);

        flood_fill(x+1,y  ,ck,cw);
        flood_fill(x-1,y  ,ck,cw);
        flood_fill(x  ,y+1,ck,cw);
        flood_fill(x  ,y-1,ck,cw);
    end;
end;
```



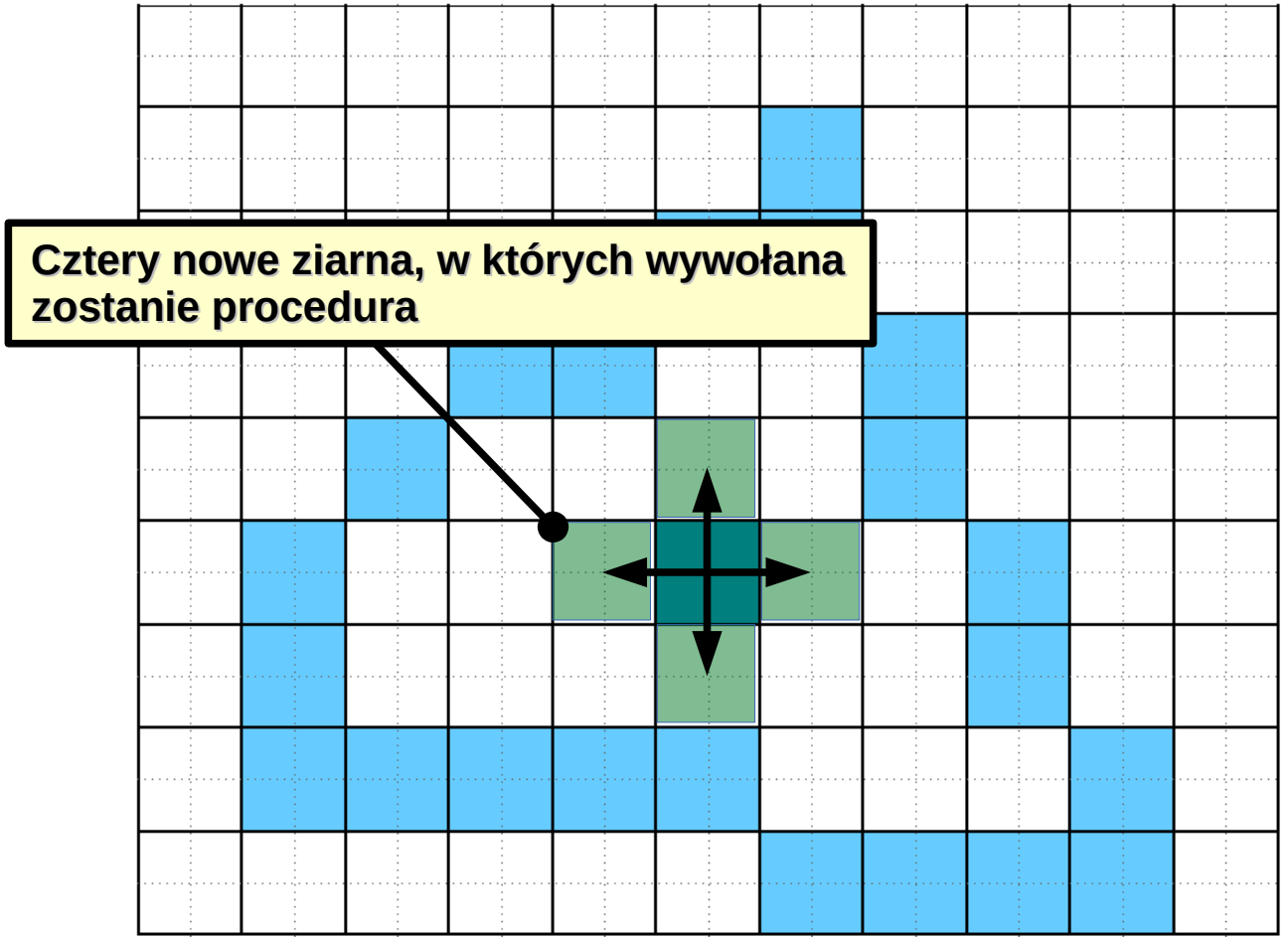
Jeżeli piksel nie został już wypełniony, ani nie jest elementem konturu, to:

- zmiana koloru na $C_w$;
- rekurencyjne wywołanie dla sąsiednich pikseli.

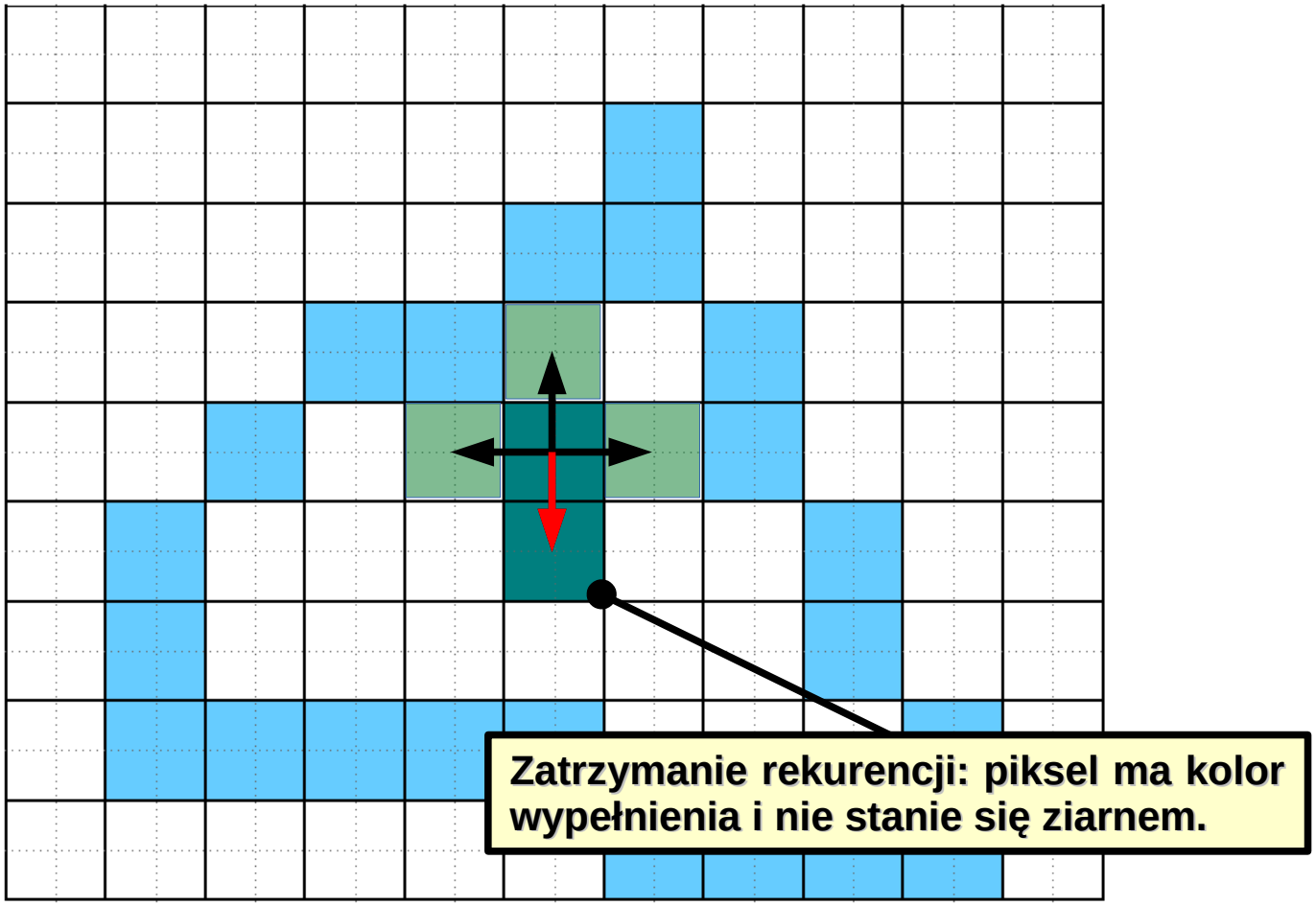
# Wypełnianie obszarów: wypełnianie przez spójność



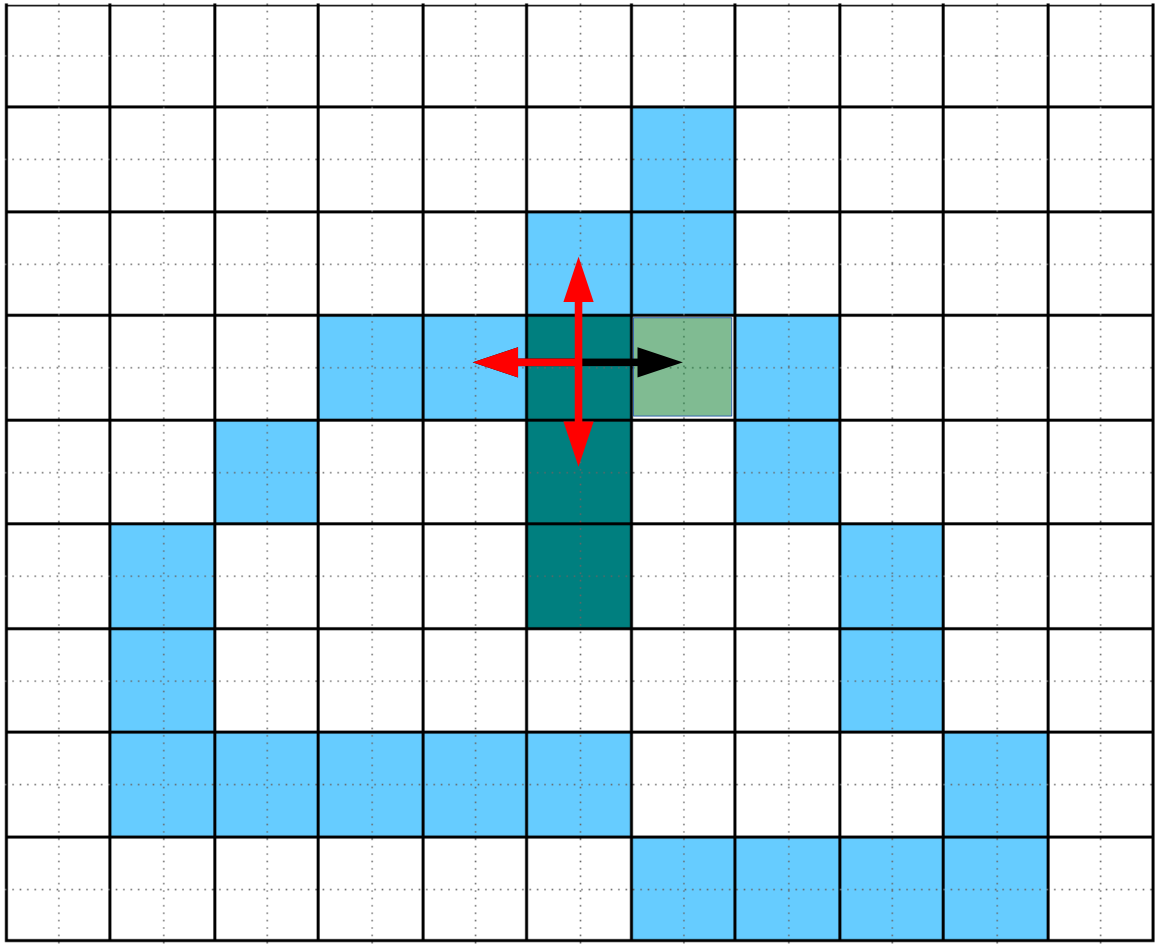
# Wypełnianie obszarów: wypełnianie przez spójność



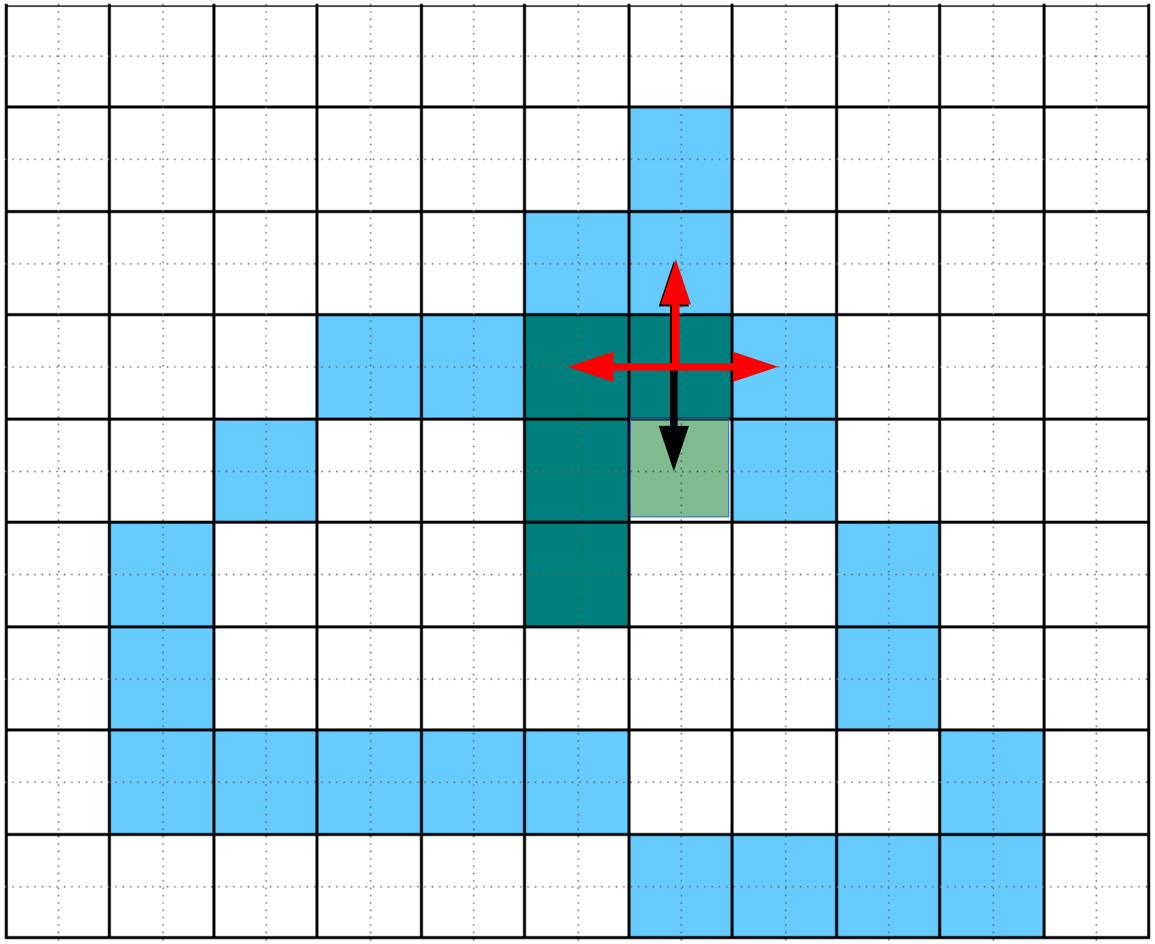
# Wypełnianie obszarów: wypełnianie przez spójność



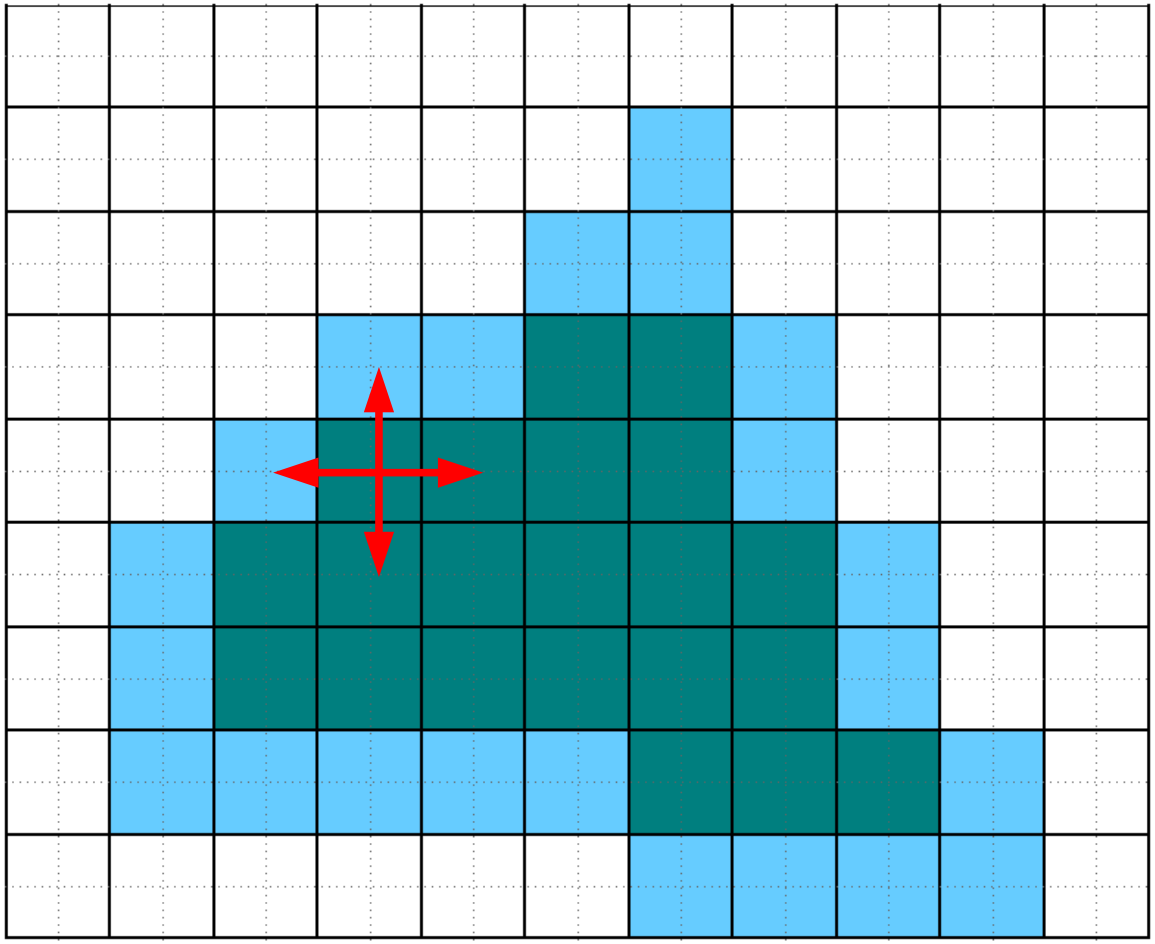
# Wypełnianie obszarów: wypełnianie przez spójność



# Wypełnianie obszarów: wypełnianie przez spójność

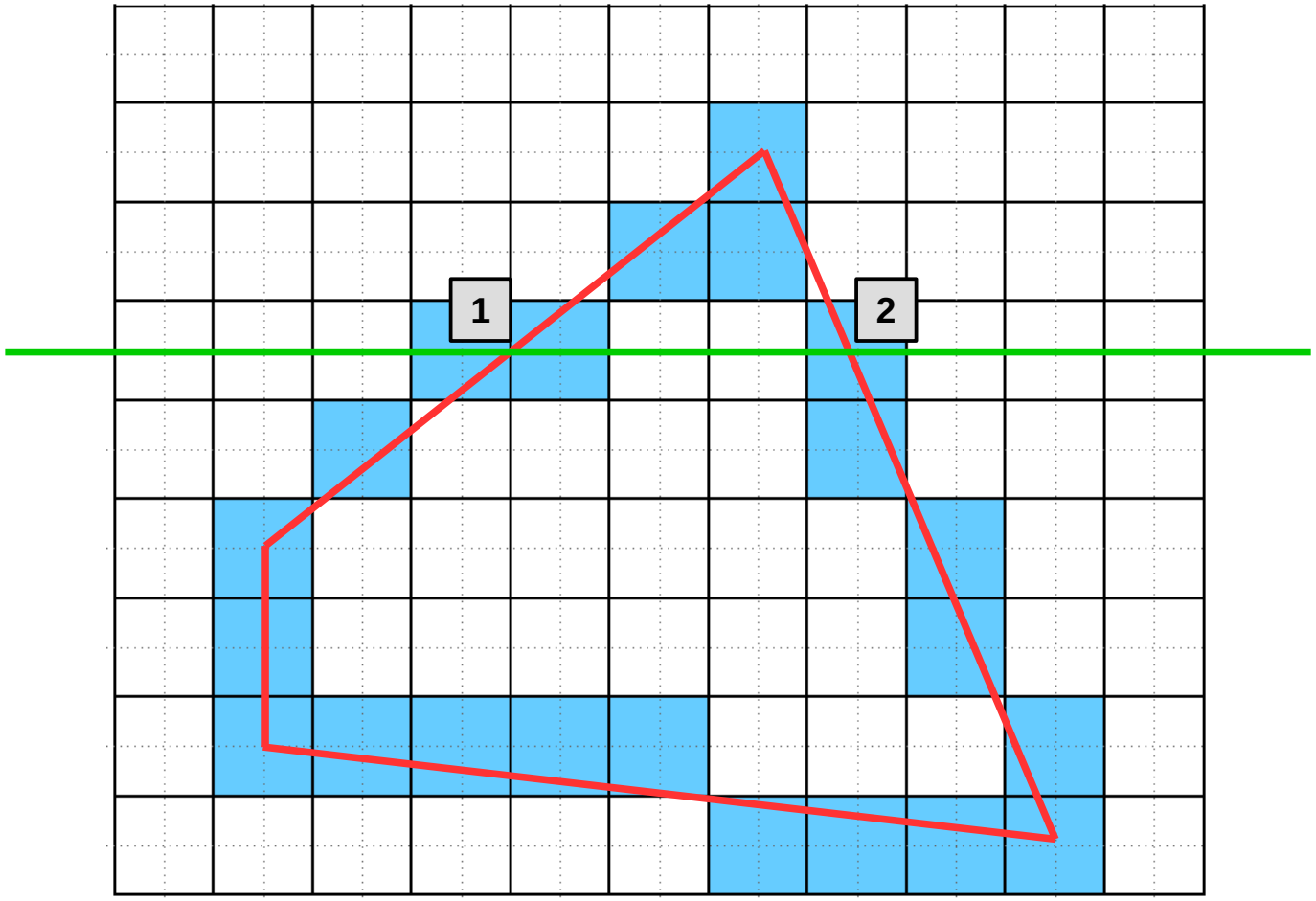


# Wypełnianie obszarów: wypełnianie przez spójność



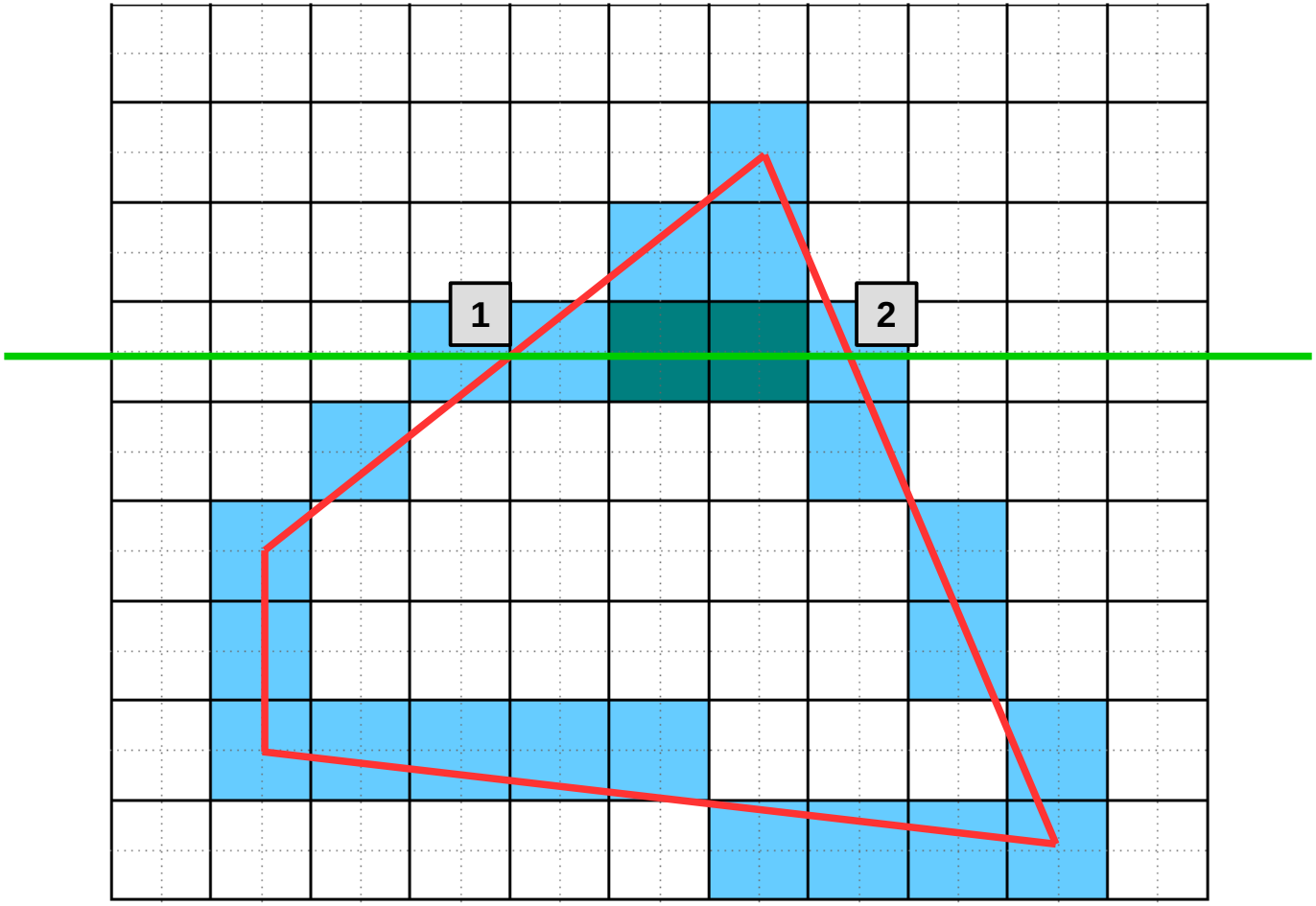
# Wypełnianie obszarów: przeglądanie wierszami z kontrolą parzystości

Przy **wektorowej definicji konturu** (czyli korzystającej z krzywych o znanych wzorach i parametrach, np. odcinków) używa się **algorytmów przeglądania wierszami (*scan-line*) z kontrolą parzystości**. Bazują one na wyznaczaniu i zliczaniu miejsc przecięcia konturu z poziomymi liniami rastra. Nieparzyste punkty przecięcia odpowiadają wejściu do wnętrza wypełnianego obszaru, a parzyste – wyjściu z niego. Tym samym piksele leżące pomiędzy takimi parami powinny zostać wypełnione.



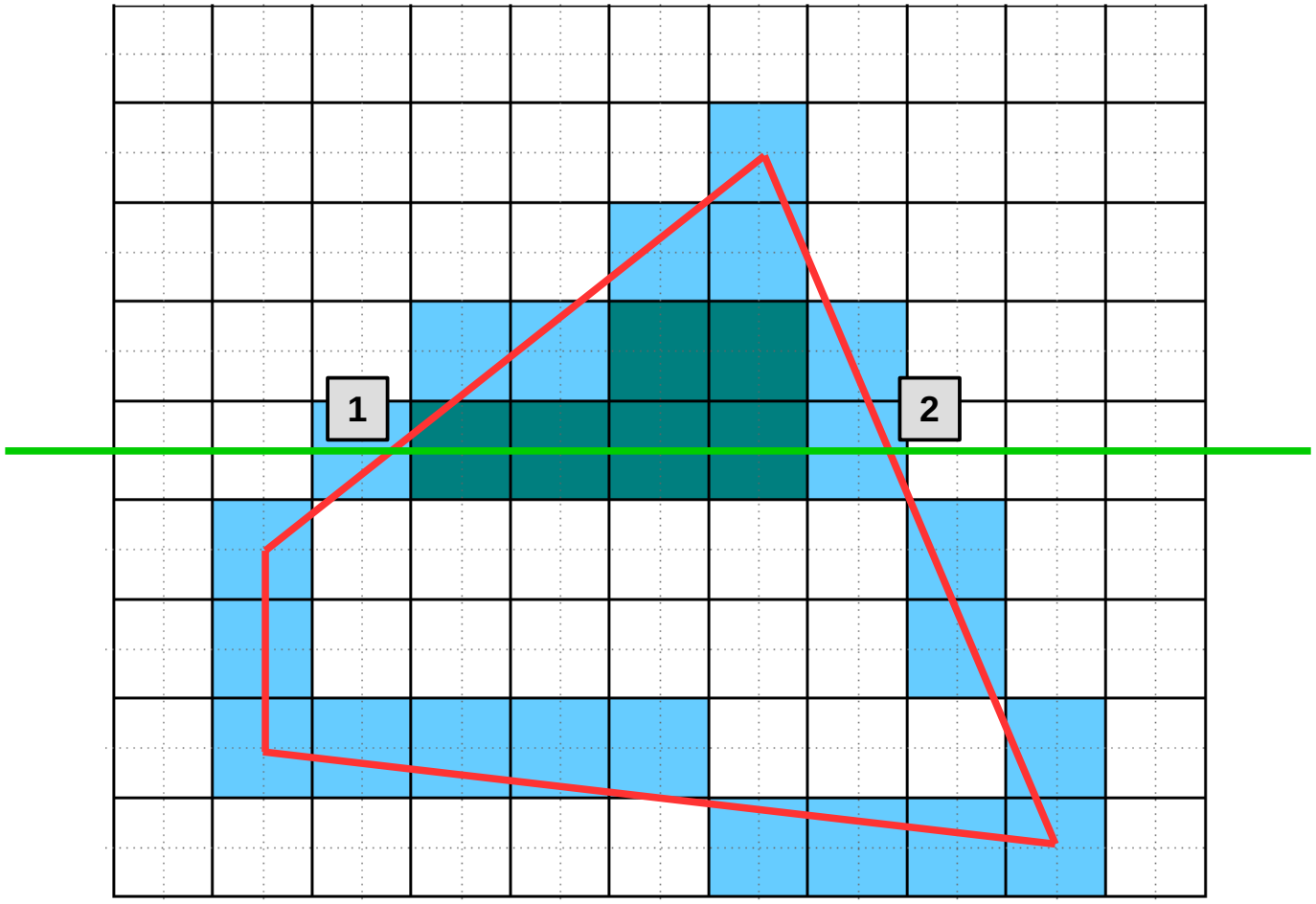
# Wypełnianie obszarów: przeglądanie wierszami z kontrolą parzystości

Przy **wektorowej definicji konturu** (czyli korzystającej z krzywych o znanych wzorach i parametrach, np. odcinków) używa się **algorytmów przeglądania wierszami (*scan-line*) z kontrolą parzystości**. Bazują one na wyznaczaniu i zliczaniu miejsc przecięcia konturu z poziomymi liniami rastra. Nieparzyste punkty przecięcia odpowiadają wejściu do wnętrza wypełnianego obszaru, a parzyste – wyjściu z niego. Tym samym piksele leżące pomiędzy takimi parami powinny zostać wypełnione.



# Wypełnianie obszarów: przeglądanie wierszami z kontrolą parzystości

Przy **wektorowej definicji konturu** (czyli korzystającej z krzywych o znanych wzorach i parametrach, np. odcinków) używa się **algorytmów przeglądania wierszami (*scan-line*) z kontrolą parzystości**. Bazują one na wyznaczaniu i zliczaniu miejsc przecięcia konturu z poziomymi liniami rastra. Nieparzyste punkty przecięcia odpowiadają wejściu do wnętrza wypełnianego obszaru, a parzyste – wyjściu z niego. Tym samym piksele leżące pomiędzy takimi parami powinny zostać wypełnione.



# DZIĘKUJĘ ZA UWAGĘ

Materiały opracowane w ramach zadania 15 „Modyfikacja międzywydziałowych studiów I stopnia na kierunku Inżynieria Biomedyczna”
projektu „NERW PW. Nauka – Edukacja – Rozwój – Współpraca”, współfinansowanego jest ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego

Fundusze Europejskie
Wiedza Edukacja Rozwój

Politechnika Warszawska

Unia Europejska
Europejski Fundusz Społeczny