POLITECHNIKA WARSZAWSKA
INSTYTUT RADIOELEKTRONIKI I TECHNIK MULTIMEDIALNYCH

Grafika Komputerowa (GRK)

Laboratorium

Ćwiczenie 3

Podstawowe algorytmy grafiki 2D

Fundusze Europejskie Wiedza Edukacja Rozwój
Politechnika Warszawska
Unia Europejska Europejski Fundusz Społeczny

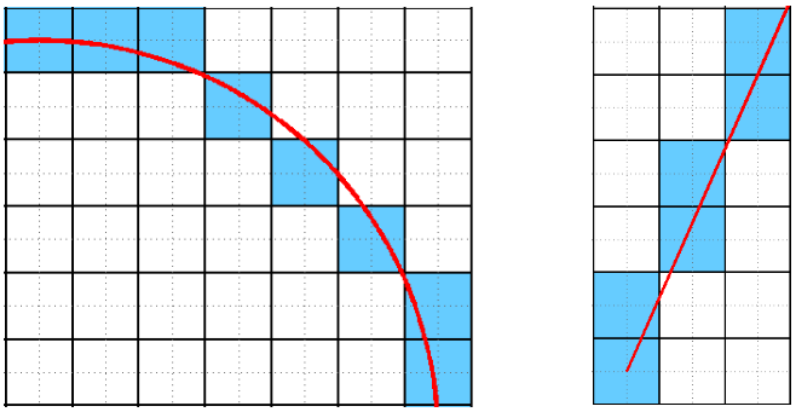
Materiały opracowane w ramach zadania 15 „Modyfikacja międzywydziałowych studiów I stopnia na kierunku Inżynieria Biomedyczna" projektu „NERW PW. Nauka – Edukacja – Rozwój – Współpraca", współfinansowanego jest ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego

1. Wstęp

Celem ćwiczenia jest zapoznanie się z podstawowymi algorytmami dwuwymiarowej grafiki komputerowej. Temat zajęć jest ściśle związany z zagadnieniami omawianymi w trakcie wykładu nr 3 z GRK, stąd też zalecane jest zapoznanie się z jego treścią przed przystąpieniem do zajęć. Dodatkowe informacje są również dostępne w pozycjach umieszczonych w spisie literatury (punkt 7 instrukcji). W części praktycznej ćwiczenia używany będzie program Matlab, będący środowiskiem programistycznym dedykowanym obliczeniom naukowym i inżynierskim (www.mathworks.com/products/matlab.html).

2. Rasteryzacja (algorytm Bresenhama)

Rasteryzacja dwuwymiarowa to proces mający na celu możliwie jak najwierniejsze odwzorowanie krzywej lub konturu 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 (rysunek 1). Stąd też konieczność używania algorytmów pozwalających zminimalizować błąd reprezentacji rastrowej.



Rys. 1. Ilustracja problemu resteryzacji linii i figur geometrycznych

2.1. Rasteryzacja 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 dany jest zależnością $y = ax + b$, jednak znając współrzędne punktów P_p i P_k można go przedstawić w postaci zgodniej ze sposobem sformułowania zadania:

$$y = \frac{(y_k - y_p)(x - x_p)}{(x_k - x_p)} + y_p \,. \tag{2.1}$$

Wprowadzenie oznaczeń $dx = x_k - x_p$ oraz $dy = y_k - y_p$ prowadzi do prostszej zależności:

$$y = \frac{dy}{dx}(x - x_p) + y_p \quad , \tag{2.2}$$

w której stosunek $a = dy/dx$ jest współczynnikiem kierunkowym prostej.

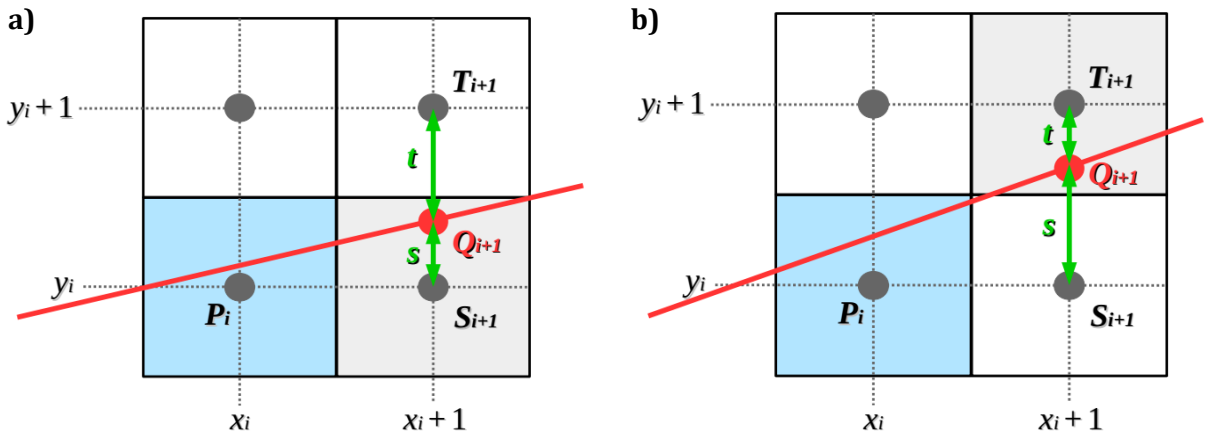
Najprostszym rozwiązaniem problemu rasteryzacji odcinka wydaje się użycie algorytmu korzystającego bezpośrednio z wzoru prostej (równanie 2.2). W algorytmie tym dla każdej kolumny rastra w zakresie od x_p do x_k (lub wiersza od y_p do y_k, w zależności od nachylenia odcinka) wyznaczana jest rzeczywista wartość drugiej współrzędnej punktu leżącego na prostej, która następnie zaokrąglana jest do najbliższej liczby całkowitej. Niestety rozwiązanie to, jakkolwiek prawidłowe, jest jednak nieefektywne: algorytm korzysta z obliczeń na liczba rzeczywistych (zmiennoprzecinkowych) i wymaga relatywnie dużej liczby operacji w każdym z kroków (mnożenia, dodawania i zaokrąglania).

Algorytm Bresenhama służy do wydajnego kreślenia odcinków (a także innych krzywych) na ekranie komputera, a jego zaletami są prostota i efektywność. Korzysta on jedynie z arytmetyki całkowitoliczbowej i w głównej pętli wykonuje tylko dwie operacje: porównanie z zerem i dodawanie.

Na początek rozpatrzmy najprostszy przypadek, w którym przyjmujemy następujące założenia:

- kąt nachylenia (kąt pomiędzy odcinkiem a osią OX) jest zawarty w przedziale od 0 do 45 stopni, co oznacza, że współczynnik kierunkowy spełnia warunek $0 \leq a \leq 1$;
- współrzędne punktu początkowego $P_p(x_p, y_p)$ i punktu końcowego $P_k(x_k, y_k)$ spełniają warunki: $x_k \geq x_p$ oraz $y_k \geq y_p$.

Idea algorytmu Bresenhama bazuje na takim wyborze kolejnych pikseli do wypełniania, aby minimalizować błąd pomiędzy rzeczywistym przebiegiem odcinka a punktami dyskretnej siatki, na której rozmieszczone są środki pikseli (rysunek 2).



Rys. 2. Wybór piksela do wypełnienia w algorytmie Bresenhama

Zakładamy, że w i-tym kroku algorytmu znajdujemy się w punkcie $P_i(x_i, y_i)$, ustalonym w poprzednim kroku $(i-1)$. Przy przyjętych na początku warunkach, w kolejnym kroku $(i+1)$ możemy wybrać wyłącznie jeden z dwóch pikseli, o środkach w punktach:

- $S_{i+1}(x_i + 1, y_i)$, dla którego współrzędna x-owa jest większa o 1 od wartości dla obecnego punktu, zaś współrzędna y-owa pozostaje bez zmian (rysunek 2a);

- $T_{i+1}(x_i + 1, y_i + 1)$, dla którego zarówno współrzędna *x*-owa, jak i *y*-owa są większe o 1 w stosunku do obecnego punktu (rysunek 2b).

Możemy zauważyć, że piksel rysowany w kolejnym kroku algorytmu na pewno będzie miał współrzędną *x* większą o jeden, a wybór dotyczy jedynie współrzędnej *y*. Zauważmy także, że dla kroku *i*+1 rzeczywista prosta przechodziłaby przez punkt $Q_{i+1}(x_q, y_q)$, którego współrzędna *x*-owa będzie wynosić $x_i + 1$, zaś drugą współrzędną można wyznaczyć z równania (2.2). Tym samym spełnienie kryterium minimalizacji błędu rasteryzacji sprowadza się do wypełnienia w następnym kroku (*i*+1) tego piksela, którego środek leży najbliżej punktu Q_{i+1}. W praktycznej realizacji algorytmu wybór pomiędzy pikselami uzależnia się od znaku zmiennej decyzyjnej d_i, o wartości proporcjonalnej do różnicy odległości punktów S_{i+1} i T_{i+1} od punktu Q_{i+1} (na rysunku 2 odległości te oznaczone jako *s* oraz *t*):

- jeżeli $d_i < 0$ (czyli $s < t$), to w kroku *i*+1 wypełniony zostanie piksel o środku w punkcie S_{i+1};
- w przeciwnym razie, jeśli $d_i \geq 0$ (czyli $s \geq t$), to wypełniony zostanie piksel o środku T_{i+1}.

Wyprowadzenie prostej postaci zmiennej decyzyjnej wymaga przeprowadzania obliczeń przedstawionych w dodatku do niniejszej instrukcji (punkt 6). Po ich wykonaniu, algorytm rysowania odcinka o nachyleniu w zakresie od 0º do 45º ($0 \leq a \leq 1$) można zapisać w poniższych krokach:

1. Obliczenie początkowej wartości zmiennej d_i oraz stałych pomocniczych p_1 i p_2:

$$d_i = d_0 = 2dy - dx\,, \tag{2.3}$$

$$p_1 = 2dy\,, \tag{2.4}$$

$$p_2 = 2(dy - dx)\,. \tag{2.5}$$

2. Ustawienie wartości współrzędnych piksela na początek odcinka: $x_i = x_p$ oraz $y_i = y_p$.

3. W pętli (powtarzanej tyle razy, ile wynosi zwiększona o 1 różnica współrzędnych *x* początku i końca odcinka) rysowany jest piksel w punkcie $P_i(x_i, y_i)$ oraz wybierane jest położenie piksela dla następnej iteracji, czemu towarzyszy aktualizacja zmiennej decyzyjnej:

   - $d_i < 0 \rightarrow$ wybieramy jest piksel S_{i+1}: x_i zwiększane o 1, y_i pozostaje bez zmian, zmienna decyzyjna dla następnej iteracji jest obliczana jako $d_{i+1} = d_i + p_1$;
   - $d_i \geq 0 \rightarrow$ wybieramy piksel T: x_i oraz y_i zwiększane o 1, zmienna decyzyjna dla następnej iteracji wyznaczana zgodnie z zależnością $d_{i+1} = d_i + p_2$.

Listing 1 zawiera przykładowy kod prostej funkcji realizującej algorytm Bresenhama w środowisku Matlab. Jest on ograniczony do rysowania odcinków o nachyleniu od 0º do 45º, jednak można go zmodyfikować tak, aby uwzględniła również inne nachylenia. Rysowanie wszystkich odcinków współczynniku kierunkowym $|a| \leq 1$ wymaga zastąpienia zmiennych dx i dy przez ich wartości bezwzględne, a także dopuszczenia możliwości zmniejszania w kolejnych krokach wartości współrzędnych x_i i/lub y_i o 1 (gdy dx i/lub dy są ujemne). Natomiast w przypadku odcinków o $|a| > 1$ konieczne jest dodatkowo zamienienie rolami zmiennych dx z dy oraz x_i z y_i.

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

% wyznaczenie wartości zmiennych pomocniczych
dx = xk - xp;
dy = yk - yp;
p1 = 2 * dy;
p2 = 2 * (dy - dx);

% sprawdzenie warunku nachylenia odcinka
a = dy/dx;
if (a < 0 || a > 1)
    disp('Błąd: nieprawidłowe nachylenie odcinka');
    return;
end

% ustalenie początkowej wartości zmiennej decyzyjnej
d = 2 * dy - dx;

% ustalenie pozycji pierwszego piksela na punkt początkowy odcinka
x = xp;
y = yp;

% główna pętla algorytmu
for i = 1 : dx+1

    % postawienie piksela w lokalizacji (x,y)
    rectangle('Position', [x-0.5, y-0.5, 1, 1], 'FaceColor', 'b');

    % aktualizacja (x,y) i modyfikacja zmiennej decyzyjnej
    if (d < 0)
        % wybór piksela poniżej teoretycznego przebiegu odcinka
        d = d + p1;
        x = x + 1;
    else
        % wybór piksela powyżej teoretycznego przebiegu
        d = d + p2;
        x = x + 1;
        y = y + 1;
    end
end
```

Listing 1. Funkcja implementująca w środowisku Matlab algorytm Bresenhama dla odcinka o nachyleniu od 0º do 45º

Przeanalizujmy teraz dokładnie zaprezentowany powyżej kod. Pierwsza linijka definiuje funkcję o nazwie *bresenham_line*, której argumentami wejściowymi są współrzędne punktów początku (zmienne *xp* oraz *yp*) oraz końca (*xk*, *yk*) rysowanego odcinka. Funkcja ta nie zwraca żadnej wartości.

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

W kolejnych linijkach (po komentarzu, którym w Maltabie jest każda linia rozpoczynająca się znakiem %) definiowane są zmienne pomocnicze: *dx* oraz *dy* stanowią różnicę pomiędzy współrzędnymi końców odcinka, zaś *p1* oraz *p2* będą używane do aktualizacji zmiennej decyzyjnej w sposób zależny od jej znaku.

```
dx = xk - xp;
dy = yk - yp;
p1 = 2 * dy;
p2 = 2 * (dy - dx);
```

Na początku założyliśmy, że algorytm będzie działał dla odcinków o kącie nachylenia od 0 do 45 stopni, co jest równoważne warunkowi $0 \leq a \leq 1$. Warunek ten sprawdzamy za pomocą instrukcji *if*, po uprzednim obliczeniu współczynnika kierunkowego $a = dy/dx$. Jeśli nie jest on spełniony, to musimy poinformować o błędzie (funkcja *disp*) i przerwać wykonywanie funkcji (instrukcja *return*).

```
a = dy/dx;
if (a < 0 || a > 1)
    disp('Błąd: nieprawidłowe nachylenie odcinka');
    return;
end
```

Jeśli odcinek zdefiniowany był prawidłowo, przechodzimy do wykonywania zasadniczej części algorytmu Bresenhama, zaczynając od wyznaczenia początkowej wartość zmiennej decyzyjnej (równanie 2.3).

```
d = 2 * dy - dx;
```

W dalszej kolejności ustalamy położenie, w którym aktualnie się znajdujemy. Na tym etapie jest ono równe punktowi początkowemu odcinka, stąd też to jego współrzędne wpisywane są do zmiennych *x* oraz *y*.

```
x = xp;
y = yp;
```

Po inicjalizacji zmiennych rozpoczyna się główna pętla *for*, która wykona się tyle razy, ile wynosi różnica między *x*-ową współrzędną początku i końca odcinka (plus 1, aby uwzględnić także punkt końcowy). Do momentu ukończenia pętli *for* algorytm powtarza swoje działanie rysując kolejne punkty.

```
for i = 1 : dx+1
    % ciało pętli
    …
end
```

Na początku każdej iteracji pętli rysowany jest piksel w aktualnym położeniu. Komenda *rectangle* stworzy kwadrat o środku w punkcie o współrzędnych (x, y) i długości boku równej 1. Kolor piksela definiowany jest przez parametr *FaceColor* (wartość '*b*' oznacza kolor niebieski).

```
rectangle('Position', [x-0.5, y-0.5, 1, 1], 'FaceColor', 'b');
```

Następnie dokonywany jest wybór kolejnego piksela na podstawie obecnej wartości zmiennej decyzyjnej *d*. Jeśli jest ona mniejsza od zera, to jako kolejny wybieramy punkt, którego współrzędna *x* jest większa o jeden, zaś *y* pozostaje bez zmian. Jednocześnie aktualizujemy zmienną decyzyjną dodając do niej zmienną pomocniczą *p1*. W przeciwnym wypadku (wartość *d* niemniejsza od zera) wybieramy kolejny punkt o obu współrzędnych większych o jeden, zaś zmienną decyzyjną modyfikujemy dodając zmienną pomocniczą *p2*.

```
if (d < 0)
    d = d + p1;
    x = x + 1;
else
    d = d + p2;
    x = x + 1;
    y = y + 1;
end
```

Opisana powyżej funkcja implementująca algorytm Bresenhama może zostać uruchomiona z linii komend środowiska Matlab, bądź wywołana w ramach dowolnej funkcji lub skryptu. Listing 2 zawiera przykład skryptu wywołującego funkcję *bresenham_line* (wynik jego działania przedstawiono na rysunku 3).

```
% współrzędne punktów początkowego (xp, yp) i końcowego (xk, yk)
xp = 3; yp = 3;
xk = 8; yk = 6;

% określenie obszaru i sposobu wyświetlania
axis([0 10 0 10]);
grid on;
hold on;

% wywołanie funkcji rasteryzacji odcinka
bresenham_line(xp, xk, yp, yk);

% narysowanie idealnego przebiegu odcinka (przed rasteryzacją)
plot([xp xk], [yp yk],'r','LineWidth',2);
```

Listing 2. Skrypt wywołujący funkcję *bresenham_line* dla odcinka o początku w punkcie (3,3) i końcu w punkcie (8,6)

Początkowe linie skryptu definiują zmienne przechowujące współrzędne początku i końca odcinka, które następnie są przekazywane jako argumenty funkcji dokonującej rasteryzacji. Kolejna grupa poleceń:

```
axis([0 10 0 10]);
grid on;
hold on;
```

odpowiada za zdefiniowanie zakresu osi (komenda *axis*, zakres od 0 do 10 w obu kierunkach), włączenie siatki (komenda *grid on*) oraz „zatrzymanie" wykresu (komenda *hold on*, która powoduje, że wywoływane

dalej instrukcje odpowiedzialne za rysowanie nie będą otwierać nowych okien z wykresami).

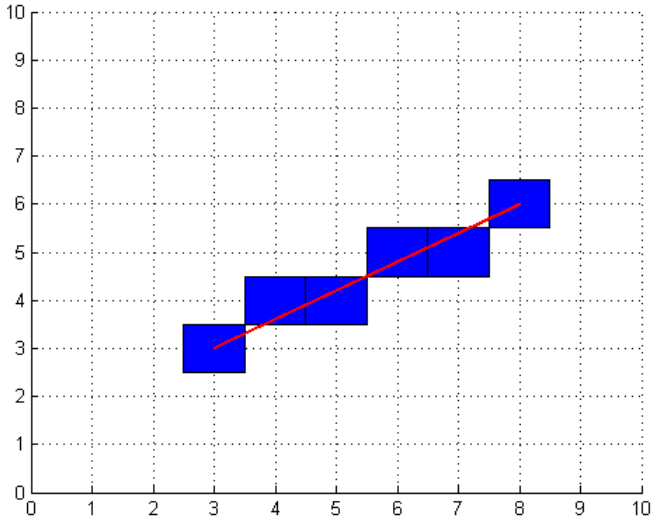
Po określeniu zakresu i sposobu wyświetlania wywoływana jest funkcja *bresenham_line*, która wyświetli efekt resteryzacji odcinka o zadanych wcześniej parametrach.

```
bresenham_line(xp, xk, yp, yk);
```

Ostatnim poleceniem skryptu jest narysowanie przebiegu idealnego odcinka za pomocą komendy *plot*.

```
plot([xp xk], [yp yk],'r','LineWidth',2);
```

W przypadku wywołania dla dwóch par współrzędnych (korzystamy z wcześniej zdefiniowanych współrzędnych początku i końca odcinka) komenda *plot* narysuje odcinek łączący dwa punkty. Dodatkowo określono grubość linii (parametr *LineWidth*) oraz jej kolor (parametr *'r'* oznacza kolor niebieski).



Rys. 3. Wynik rasteryzacji odcinka o punkcie początkowym (3,3) i punkcie końcowym (8,6)

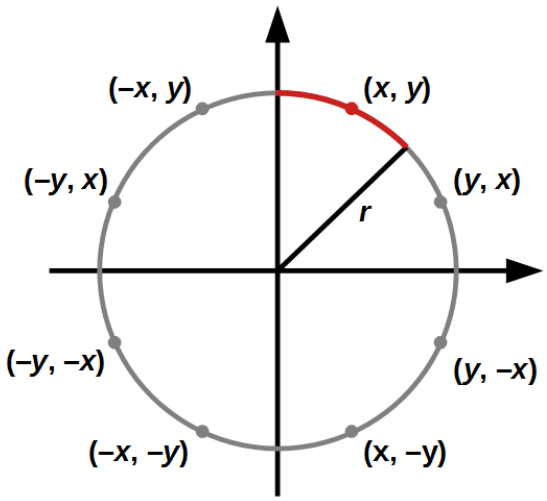
2.2. Rasteryzacja okręgu

Zadanie polega na wyznaczeniu pikseli stanowiących wizualizację okręgu o środku w punkcie $P_c(x_c, y_c)$ i promieniu *r*. Zbiór punktów leżących na okręgu dany jest zależnością:

$$(x - x_c)^2 + (y - y_c)^2 = r^2\,, \tag{2.6}$$

jednak bez utraty ogólności można ograniczyć dalszy opis do okręgu o środku w punkcie (0, 0).

Podobnie jak to miało miejsce dla odcinka, również w przypadku okręgu wyznaczanie pikseli bezpośrednio z definicji prowadziłoby do nieefektywnego algorytmu. Dlatego też w praktyce zwykle stosuje się podejście stanowiące uogólnienie algorytmu Bresenhama, omówionego w punkcie 2.1. Algorytm realizuje się dla 1/8 okręgu, czyli dla jednego oktantu, który na rysunku 4 został zaznaczony kolorem czerwonym. Współrzędne pozostałych pikseli określa się korzystając z symetrii wobec środka okręgu. Tym samym na podstawie znajomości położenia pojedynczego punktu w jednym oktancie można narysować w sumie 8 pikseli o współrzędnych: (x, y), $(x, -y)$, $(-x, y)$, $(-x, -y)$, (y, x), $(y, -x)$, $(-y, x)$, $(-y, -x)$.



Rys. 4. Ilustracja sposobu wykorzystania symetrii podczas rasteryzacji okręgu

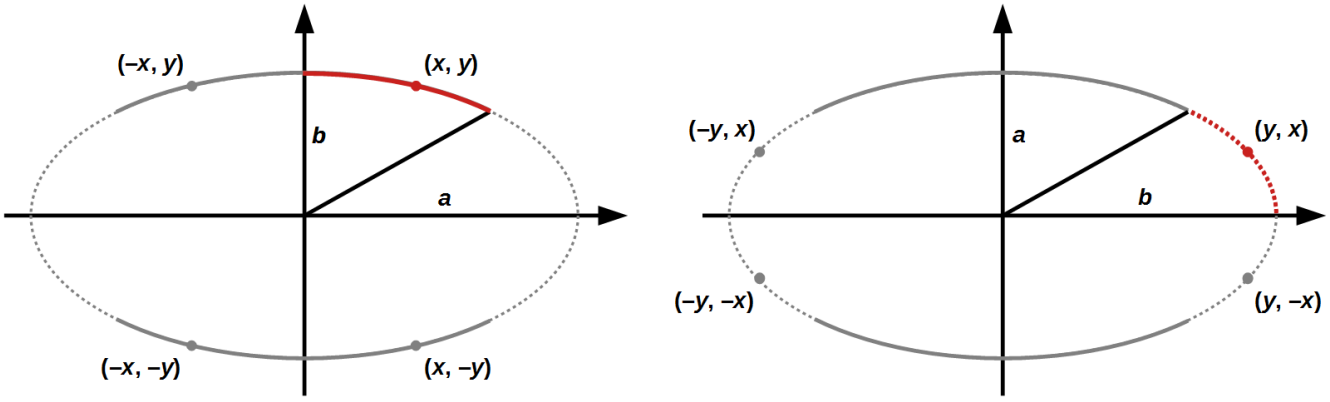
Zarówno ogólny schemat działania algorytmu, jak i jego implementacja w środowisku Matlab są bardzo podobne do tych przedstawionych wcześniej dla odcinka (różnice dotyczą jedynie sposobu wyznaczania wartości zmiennej decyzyjnej). Dlatego też listing 3 ogranicza się do prezentacji pseudokodu algorytmu.

```
ustal wartość początkową zmiennej decyzyjnej d = 5 – 4r;

ustal aktualne współrzędne (x,y) na punkt początkowy (0,r);

dopóki (x <= y):

   narysuj aktualny punkt (x,y) oraz jego 7 symetrii

   jeśli (zmienna decyzyjna < 0):
         zaktualizuj zmienną decyzyjną d = d + 8x + 12;
         zwiększ o jeden wartość x;
   w przeciwnym wypadku:
         zaktualizuj zmienną decyzyjną d = d + 8(x - y) + 20;
         zwiększ o jeden wartość x;
         zmniejsz o jeden wartość y;
```

Listing 3. Pseudokod algorytmu Bresenhama rasteryzacji okręgu

2.3. Rasteryzacja elipsy

Rozumowanie przeprowadzone dla okręgu (punkt 2.2) można zastosować również do rasteryzacji elipsy. W tym przypadku algorytm wykonuje się dwukrotnie, za każdym razem wyznaczając punkty dla 1/8 elipsy (czerwone fragmenty na rysunku 5). W pierwszym powtórzeniu działanie algorytm dotyczy czterech części elipsy, zaznaczonych linią ciągłą na rysunku 5. Na podstawie obliczonych współrzędnych (x, y) jednego punktu określa się położenia punktów w pozostałych częściach, korzystając przy tym z symetrii: $(x, -y)$, $(-x, -y)$ oraz $(-x, y)$. Aby dokonać rasteryzacji dalszych czterech części (linia przerywana na rysunku 5) należy wykorzystać ten sam algorytm, jednak zamieniając ze sobą rolami osie elipsy a i b oraz współrzędne x i y przy rysowaniu pikseli. Pseudokod całego algorytmu został przedstawiony na listingu 4.



Rys. 5. Ilustracja sposobu wykorzystania symetrii podczas rasteryzacji elipsy

```
ustal wartość początkową zmiennej decyzyjnej d = 4b² - 4a²b + a²;

ustal aktualne współrzędne (x,y) na punkt początkowy (0,b);

dopóki ( x² <= (a4 / (a² + b²)) )

    narysuj aktualny punkt (x,y) oraz jego 3 symetrie;

    jeśli (zmienna decyzyjna < 0):
          zaktualizuj zmienną decyzyjną d = d + 8b²x + 12b²;
          zwiększ o jeden wartość x;
    w przeciwnym wypadku:
          zaktualizuj zmienną decyzyjną d = d + 8b²x + 12b² - 8a²y + 8a²;
          zwiększ o jeden wartość x;
          zmniejsz o jeden wartość y;

zamień a z b
powtórz algorytm zmieniając x na y przy rysowaniu pikseli
```

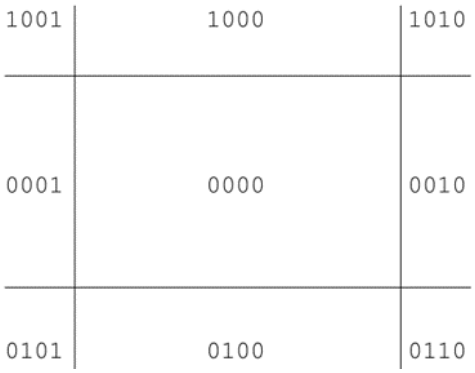
Listing 4. Pseudokod algorytmu Bresenhama rasteryzacji elipsy

3. Obcinanie odcinków (algorytm Cohena – Sutherlanda)

Obcinanie (ang. *clipping*) czy to odcinków, czy całych figur polega na ograniczeniu danego obiektu do pewnego widocznego fragmentu. Algorytm zaproponowany przez Cohena i Sutherlanda jest algorytmem szybkim i efektywnym. Pozwala na łatwe odrzucenie obiektów lub ich zaakceptowanie w przypadkach, gdy znajdują się on odpowiednio: całkowicie poza obszarem widzialnym lub całkowicie w obszarze widzialnym.

Zakładamy, że dysponujemy prostokątnym oknem wyznaczającym obszar widzialny. Okno zdefiniowane jest przez lewy dolny i prawy górny róg. Dla potrzeb procedury eliminacji niewidocznych fragmentów linii przedłużamy odcinki tworzące okno – w ten sposób proste, na których leżą odcinki tworzące okno dzielą płaszczyznę na dziewięć obszarów (rysunek 6). Następnie każdemu z obszarów przyporządkowujemy czterobitowy kod według następującej zasady (pozycje bitów są liczone od prawej do lewej):

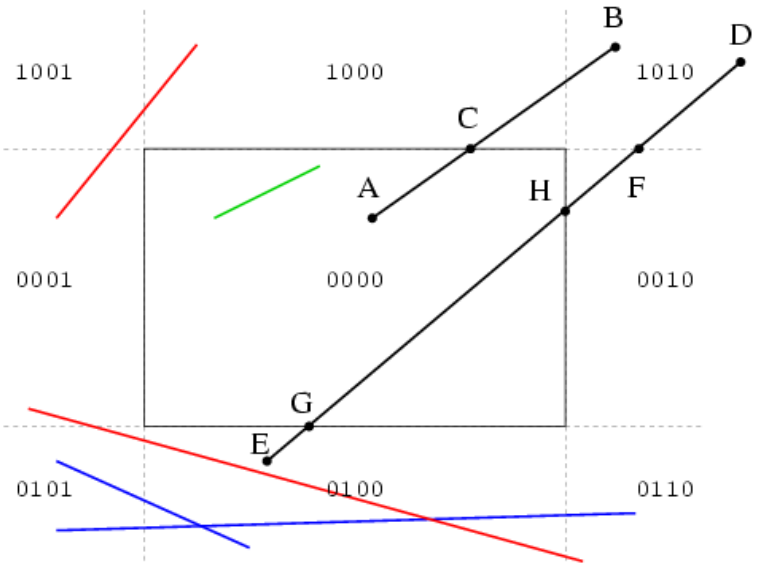
- bit nr 0 ma wartość 1, jeśli obszar leży na lewo od obszaru widzialnego (punkt w tym obszarze ma współrzędną *x*-owa jest mniejsza od współrzędnej *x*-owej lewego dolnego rogu);
- bit nr 1 ma wartość 1, jeśli obszar leży na prawo od obszaru widzialnego (punkt w tym obszarze ma współrzędną *x*-owa jest większa od współrzędnej *x*-owej prawego górnego rogu);
- bit nr 2 ma wartość 1, jeśli obszar leży poniżej obszaru widzialnego (punkt w tym obszarze ma współrzędną *y*-owa jest mniejsza od współrzędnej *y*-owej lewego dolnego rogu);
- bit nr 3 ma wartość 1, jeśli obszar leży powyżej obszaru widzialnego (punkt w tym obszarze ma współrzędną *y*-owa jest większa od współrzędnej *y*-owej prawego górnego rogu).



Rys. 6. Podział płaszczyzny na dziewięć obszarów i odpowiadające im czterobitowe kody

Odpowiednie kody przypisuje się końcom odcinków w zależności od ich położenia. Warto zauważyć, że jeżeli suma logiczna (OR) kodów obu końców odcinka daje wynik zerowy (0000), wówczas odcinek jest w całości akceptowany, gdyż na pewno leży wewnątrz obszaru widzialnego (oba końce muszą mieć kod 0000). Jeśli natomiast iloczyn logiczny (AND) kodów obu końców odcinka daje wynik niezerowy, (na którymkolwiek z bitów występuje jedynka) wówczas odcinek jest w całości odrzucany, gdyż na pewno leży całkowicie poza obszarem widzialnym (np. 0101 & 0110 = 0100). Dla każdego innego przypadku musimy rozpocząć procedurę obcinania. Warto też zauważyć, że może dojść do sytuacji, w której, mimo że odcinek

leży poza obszarem okna widzialności, to wynik iloczynu logicznego i tak będzie zerowy. Uniemożliwi to natychmiastowe odrzucenie odcinka – trzeba będzie go obcinać do momentu, w którym test iloczynu logicznego zakończy się pomyślnie i odcinek zostanie odrzucony.



Rys. 7. Przykłady przycinania odcinków: kolor zielony – odcinki od razu zaakceptowane; niebieski – odcinki od razu odrzucone; czarny – odcinki przycinane aż do momentu, w którym znajdą się w obszarze widocznym (po spełnieniu warunku sumy logicznej); czerwony – odcinki niejednoznaczne, dla których musimy rozpocząć przycinanie, a ich odrzucenie nastąpi dopiero w jednej z iteracji algorytmu (gdy spełnimy warunek iloczynu logicznego dwóch końców)

W trakcie procedury obcinania wybieramy jeden z końców odcinka i sprawdzamy, czy na którymś bicie jego kodu występuje jedynka (w przeciwnym razie przechodzimy do sprawdzania drugiego końca). Jeżeli wykryjemy jedynkę, to najpierw obcinamy odcinek wyznaczając punkt przecięcia z odpowiednią prostą (określoną na podstawie tego, na którym bicie kodu występowała jedynka), a potem przystępujemy do sprawdzania drugiego końca. Jeśli drugi koniec ma w kodzie wartości niezerowe, to ponownie obcinamy odcinek, tym razem z drugiej strony. Następnie aktualizujemy kody nowo wyznaczonych końców odcinka i powtarzamy całą procedurę od nowa: przeprowadzamy testy sumy i iloczynu, i jeśli odcinek nadal nie jest zaakceptowany lub odrzucony, to obcinamy go ponownie.

Prześledźmy teraz algorytm Cohena-Sutherlanda na przykładzie przycinania odcinka AB z rysunku 7, przy okazji analizując fragmenty kodu odpowiedzialne są za realizacje poszczególnych jego etapów. Ze względu na fakt, że dla algorytmu Bresenhama dokłasnie omawialiśmy wszystkie polecenia, tym razem pominiemy pewne elementy (np. instrukcje odpowiedzialne za rysowanie, gdyż ich składania jest taka, jak poprzednio), a skupimy się tylko na kluczowych fragmentach. Pełny kod omawianej tu funkcji *cohen_sutherland* zawarty jest w archiwum dostępnym na stronie przedmiotu.

W naszym przykładzie założymy, że punkt początkowy A ma współrzędne (x_p, y_p), zaś punkt końcowy B ma współrzędne (x_k, y_k). Obszar widzialny jest natomiast zdefiniowany przez współrzędne lewego dolnego (wx_1, wy_1) i prawego górnego (wx_2, wy_2) rogu.

Funkcja *cohen_sutherland* przyjmuje jako argumenty wejściowe współrzędne końców odcinka (zmienne *xp* i *yp* oraz *xk* i *yk*), a także rogów obszaru widzialnego (zmienne *wx1* i *wy1* oraz *wx2* i *wy2*). Po zakończeniu działania zwracane są współrzędne końców odcinka po operacji przycinania (*px1* i *py1* oraz *px2* i *py2*).

```
function [px1,py1,px2,py2]=cohen_sutherland(wx1,wy1,wx2,wy2,xp,yp,xk,yk)
```

Rozpoczynamy od zakodowania końców odcinaka AB. W tym celu korzystamy z funkcji *koduj_koniec*, wywołując ją raz dla współrzędnych punktu A (początek odcinka), a następnie dla współrzędnych punktu B (koniec odcinka). Za każdym razem funkcji przekazywane są także współrzędne obszaru widzialnego.

```
kod1 = koduj_koniec(xp, yp, wx1, wy1, wx2, wy2);
kod2 = koduj_koniec(xk, yk, wx1, wy1, wx2, wy2);
```

Jak łatwo się domyślić, funkcja ta zwraca czterobitowy kod ustalony na podstawie współrzędnych punktu i okna widzialnego. Kody przypisywane są zmiennym *kod1* oraz *kod2*, odpowiednio dla punktów A oraz B. W naszym przypadku *kod1* = 0000, natomiast *kod2* = 1010.

Następnie przystępujemy do części zasadniczej algorytmu, która umieszczona jest w pętli *while.* Jako warunek pętli wpisana jest na stałe wartość jeden, czyli warunek ten zawsze jest spełniony. Oznacza to, że pętla będzie wykonywać się cały czas, chyba że wewnątrz pętli wywołamy komendę *return*, która przerwie działanie całej funkcji.

```
while (1)
    % ciało pętli
    …
end
```

Zgodnie z teorią, którą omawialiśmy wcześniej, wewnątrz pętli najpierw dokonujemy sprawdzenia sumy logicznej (w Matlabie reprezentowanej przez operator |) kodów obu końców odcinka.

```
if (~any(kod1 | kod2))
    px1 = xp;
    px2 = xk;
    py1 = yp;
    py2 = yk;
    % aktualizacja rysunku
    plot([px1 px2], [py1 py2], 'g-', 'LineWidth',3);
    disp('Sytuacja koncowa.');
    return;
end
```

Sprawdzenie to umieszczone jest jako warunek instrukcji *if*. Jeśli w wyniku sumy logicznej *(kod1 | kod2)* nie będzie żadnych jedynek (do sprawdzenia tego warunku służy funkcja *any*, której wynik jest zanegowany operatorem ~), wówczas wejdziemy do wnętrza instrukcji warunkowej. Do zmiennych pomocniczych *px1, py1, px2, py2* (określających ostateczne współrzędne odcinka po całej operacji obcinania) przypisujemy współrzędne początku i końca odcinka, rysujemy odcinek wyświetlając stosowną informację i przerywamy działanie funkcji komendą *return*. W naszym przypadku 0000 | 1010 = 1010, czyli wynik niezerowy. Nie wejdziemy zatem do wnętrza instrukcji warunkowej, lecz pójdziemy dalej.

W kolejnym kroku sprawdzamy iloczyn logiczny (operator & w Matlabie) kodów obu końców odcinka.

```
if (any(kod1 & kod2))
    px1 = xp;
    px2 = xk;
    py1 = yp;
    py2 = yk;
    % aktualizacja rysunku
    plot([px1 px2], [py1 py2], 'r-', 'LineWidth',3);
    disp('Sytuacja koncowa.');
    return;
end
```

Jeśli wynik *any(kod1 & kod2)* będzie prawdziwy (odcinek leży poza obszarem widzialnym), to wykonane zostanie ciało instrukcji warunkowej *if*. Ponownie zmiennym pomocniczym przypiszemy współrzędne początku i końca odcinka, narysujemy odcinek (dla odróżnienia innym kolorem, bo będzie on odrzucony), wyświetlimy komunikat i przerwiemy działanie funkcji. W naszym przykładzie 0000 & 1010 = 0000, a zatem brak jakiekolwiek jedynki spowoduje, że nie wejdziemy do wnętrza tej instrukcji warunkowej.

W kolejnym kroku rozpoczynamy zasadnicze przycinanie. Weryfikujemy kod jednego z końców odcinka – jeśli gdziekolwiek wystąpi w nim jedynka, będzie to oznaczało konieczność przycinania (bo leży on poza obszarem widzialnym). Jeśli jednak kod będzie zerowy, to punkt leży wewnątrz okna widoku i nie należy go modyfikować, co oznacza, że przycinanie będzie dotyczyć drugiego końca.

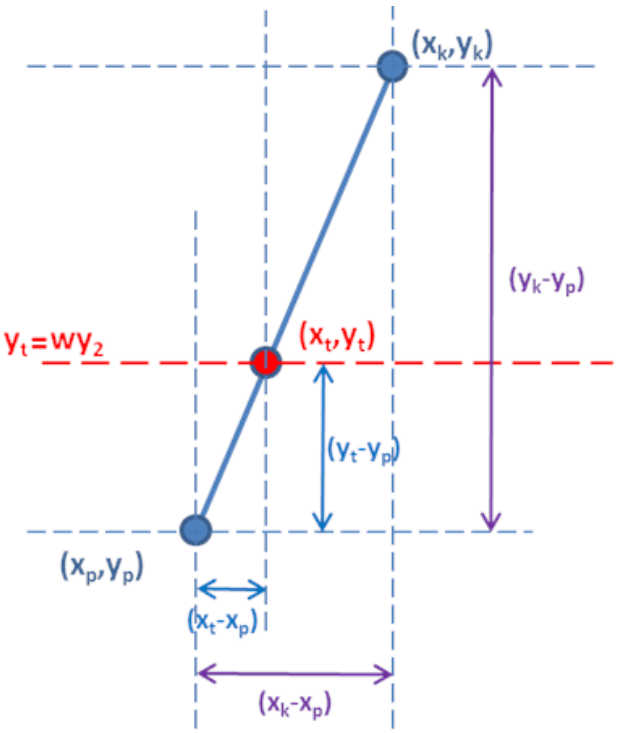
```
if (any(kod1))
    kod = kod1;
else
    kod = kod2;
end
```

W naszym przykładzie najpierw sprawdzamy kod punktu A. Ponieważ jest to 0000 (brak jakiekolwiek jedynki), więc przechodzimy do punktu B, a w zasadzie poprzez instrukcje *else* wybieramy ten punkt automatycznie, przepisując jego czterobitowy kod na zmienną pomocniczą *kod*.

Następnie szukamy, w którym miejscu w zmiennej *kod* występuje jedynka. W zależności od lokalizacji jedynki w kodzie ustalamy, do której krawędzi obszaru widzialnego będziemy przycinać nasz odcinek.

```
if (kod(1) == 1)
    % przycinanie przez górną krawędź
    yt = wy2;
    t = (yt-yp) / (yk-yp);
    xt = xp + t*(xk-xp);
elseif (kod(2) == 1)
    % przycinanie przez dolną krawędź
elseif (kod(3) == 1)
    % przycinanie przez prawą krawędź
elseif (kod(4) == 1)
    % przycinanie przez lewą krawędź
end
```

Dla naszego punktu B kod wynosi 1010. Sprawdzamy zatem pierwszą cyfrę: *kod(1) == 1*. Operacja ta jest warunkiem dla instrukcji warunkowej *if*, która dla naszego przypadku kończy się sukcesem. Realizujemy zatem przycinanie do górnej krawędzi, które sprowadza się do wyznaczenia punktu przecięcia odcinka z prostą $y = wy_2$. Dokładny sposób wyznaczenia współrzędnych punktu przecięcia C(x_t, y_t) pokazano na rysunku 8, natomiast implementacja tych obliczeń znajduje się na listingu w ciele instrukcji warunkowej *if (kod(1) == 1)*.



$$\frac{x_t - x_p}{y_t - y_p} = \frac{x_k - x_p}{y_k - y_p}$$

$$\begin{cases} x_t = x_p + \dfrac{wy_2 - y_p}{y_k - y_p}(x_k - x_p) \\ y_t = wy_2 \end{cases}$$

Rys. 8. Sytuacja przycinania odcinka do górnej krawędzi okna

Warto w tym miejscu zwrócić uwagę na małą niezgodność, jaka występuje pomiędzy teoretycznym opisem a przyjętą praktyką realizacji algorytmu. Gdy opisywaliśmy sposób oznaczania dziewięciu obszarów za pomocą czterobitowych kodów, traktowaliśmy kod jako liczbę w zapisie dwójkowym. Oznacza to, że mówiąc o bicie zerowym mieliśmy na myśli wartość na pierwszej pozycji od prawej strony (101**0** – zaznaczono bit zerowy). W praktycznej realizacji naszego algorytmu kod przechowywany jest w postaci czteroelementowego wektora. Sprawdzenie pierwszej cyfry kodu poprzez odwołanie *kod(1)* powoduje odniesienie się do pierwszego elementu wektora, czyli do wartości na pierwszej pozycji od lewej strony (**1**010 – zaznaczono *kod(1)*). W rzeczywistości nie ma większego znaczenia, od której strony analizujemy nasz czterobitowy kod. Ważne jest tylko by mieć świadomość, na której aktualnie znajdujemy się pozycji w kodzie i do której krawędzi ona się odnosi. Zatem jeśli dla naszego punktu B *kod(1) ==1,*to w rzeczywistości odwołujemy się do 3 bitu, który mówi nam o wystawaniu odcinka powyżej górnej krawędzi ekranu.

Po określeniu współrzędnych (x_t, y_t) musimy dokonać aktualizacji – wyznaczone miejsce przecięcia staje się nowym punktem końcowym odcinka, któremu należy przypisać nową wartość czterobitowego kodu.

```
if (kod == kod1)
      …
else
    x2 = xt;
    y2 = yt;
    kod2 = koduj_koniec(x2, y2, wx1, wy1, wx2, wy2);
    plot(x2, y2, 'Marker', 'o', 'MarkerEdgeColor', 'r','MarkerFaceColor', 'r');
    disp('Przycieto odcinek.');
end
```

Sprawdzamy zatem ponownie instrukcją warunkową *if*, który koniec przed chwilą przycinaliśmy. W naszym przypadku na zmienną pomocniczą *kod* wpisaliśmy wcześnie kod końca odcinak, czyli *kod2,* a zatem wejdziemy do instrukcji warunkowej po słowie kluczowym *else*. Na współrzędne końca odcinka (*x2, y2*) przepiszemy wyliczone współrzędne punktu przecięcia C(x_t, y_t) oraz ponownie wyliczymy aktualną wartość *kod2* za pomocą funkcji *koduj_koniec*. Następnie zaznaczamy na rysunku punkt przycięcia, informując użytkownika o wykonanej procedurze przycinania (polecenia *plot* i *disp*). Zrealizowaliśmy w ten sposób przycięcie odcinak AB do odcinka AC.

Pamiętamy, że algorytm działa w pętli *while,* a zatem wracamy teraz do początku i ponownie sprawdzamy sumę logiczną kodów obu końców odcinka AC. Tym razem zaktualizowany kod dla końca w punkcie C wynosi 0000, a zatem suma logiczna 0000 | 0000 daje nam wynik zerowy 0000. Oznacza to, że odcinek leży całkowicie wewnątrz obszaru widzialnego, więc nie musimy już nic przycinać i możemy zakończyć działanie algorytmu. W naszym kodzie wynik przyrównania do zera jest prawdziwy, a zatem wejdziemy do wnętrza omawianej już wcześniej instrukcji warunkowej.

```
if (~any(kod1 | kod2))
    px1 = xp;
    px2 = xk;
    py1 = yp;
    py2 = yk;
    % aktualizacja rysunku
    plot([px1 px2], [py1 py2], 'g-', 'LineWidth',3);
    disp('Sytuacja koncowa.');
    return;
end
```

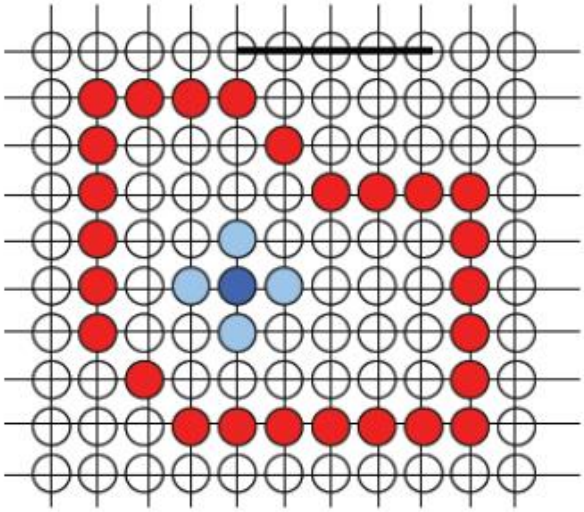
Do zmiennych *px1, py1. px2, py2* (przechowujących finalne współrzędne odcinka po zakończeniu obcinania) przypisujemy aktualne współrzędne początku i końca odcinka (czyli odcinka AC), rysujemy odcinek wyświetlając stosowną informację i przerywamy działanie funkcji komendą *return*.

4. Wypełnianie obszarów

Obok rasteryzacji odcinków i krzywych, wypełnianie obszarów jest kolejnym ważnym problemem w grafice dwuwymiarowej. Dla nieskomplikowanych figur zagadnienie to wydaje się stosunkowo proste, jednakże w ogólnym przypadku, dla obszarów o dowolnym kształcie, zarówno wypukłych, jak i wklęsłych, zadanie to jest już nietrywialne.

Algorytmy wypełniania różnią się w zależności od sposobu opisu konturów obszaru. Opis taki może być albo rastrowy (obszar zdefiniowany jest przez kontur w postaci zbioru punktów na płaszczyźnie rastra), albo wektorowy (obszar ograniczony odcinkami, zaś definicja obszaru zakłada podanie współrzędnych tych odcinków). W zależności od sposobu opisu wyróżniamy dwie podstawowe grupy algorytmów: bazujące na wypełnianiu przez spójność oraz bazujące na kontroli parzystości.

Algorytmy wypełniające obszary przez spójność są typowe dla rastrowego opisu konturów obszaru. Zakłada się w nich znajomość punktu znajdującego się wewnątrz obszaru. W kolejnych krokach algorytm dołącza kolejne piksele w sąsiedztwie 4-spójnym (lub 8-spójnym), analizując ich kolor. Warunkiem zatrzymania algorytmu jest sytuacja, w której kolor przyległego piksela jest równy kolorowi konturu. Ideę wypełniania obszarów przez spójność pokazano na rysunku 9.



Rys. 9. Wypełnianie przez spójność. Po lewej stronie pokazano fragment rastra. Kolorem czerwonym zaznaczone są piksele stanowiące kontur obiektu. Kolor ciemnoniebieski to punkt startowy wewnątrz obszaru (ziarno), zaś kolorem jasnoniebieski zaznaczono jego 4-spójne sąsiedztwo (czyli czterech najbliższych sąsiadów)

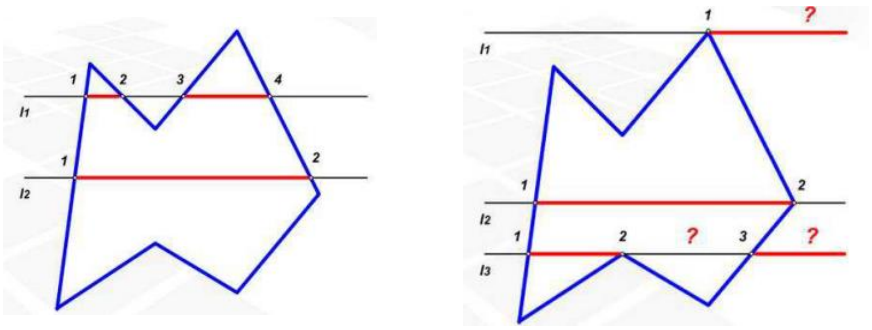
Przykładową realizacją wypełniania przez spójność jest algorytm wypełniania przez sianie (nazywany inaczej algorytmem wypełniania przez zalewanie lub „pożarem prerii"). Algorytm ten realizowany jest w postaci rekurencyjnej. W każdym kroku procedury wypełniania analizujemy piksel, w którym aktualnie się znajdujemy i jeśli nie ma on koloru krawędzi ani wypełnienia, to malujemy go na odpowiedni kolor, a następnie rekurencyjnie wywołujemy procedurę dla jego czterech sąsiadów. W przeciwnym razie nie wykonujemy żadnej czynności. Pseudokod procedury realizującej algorytm wypełniania przez sianie pokazano na listingu 5.

```
procedura wypełniania (współrzędne piksela, kolor konturu, kolor tła)

    pobierz kolor aktualnego piksela;

    jeśli (kolor różny od koloru krawędzi i różny od koloru wypełnienia)

          ustaw kolor piksela na kolor wypełnienia;

          wywołaj procedurę wypełniania dla piksela poniżej;
          wywołaj procedurę wypełniania dla piksela powyżej;
          wywołaj procedurę wypełniania dla piksela na lewo;
          wywołaj procedurę wypełniania dla piksela na prawo;
```

Listing 5. Pseudokod algorytmu wypełniania przez sianie

Rekurencyjny algorytm wypełniania przez sianie jest mało efektywny: może spowodować przepełnienie stosu, a kolor pikseli badany jest nadmiarowo (kilkukrotnie sprawdzamy te same piksele). W algorytmie można ominąć stosowanie rekurencji poprzez użycie struktury listy. Typowym zaś pomysłem stosowanym do przyspieszania jest operowanie nie na pojedynczych pikselach, lecz na tzw. segmentach, czyli ciągach poziomych pikseli ograniczonych po obu stronach konturem.

W przypadku algorytmów z kontrolą parzystości, charakterystycznych dla wektorowego opisu konturów, jedna z typowych realizacji nosi nazwę algorytmu „YX". Jego działanie polega na kontroli parzystości przecięć wypełnianego konturu z poziomymi liniami rastra. Pewien problem, z którym należy się zmierzyć, stanowią punkty lokalnych ekstremów brzegowych (wierzchołków). Ideę algorytmu „YX" z kontrolą parzystości pokazano na rysunku 10.



Rys. 10. Idea wypełniania przez kontrolę parzystości. Po lewej stronie pokazano sytuację idealną – kolejne pary przecięć wyznaczają początek i koniec poziomych odcinków linii rastra znajdujących wewnątrz wypełnianego obszaru. Po prawej stronie zilustrowano sytuację sporną w momencie natrafiania na wierzchołki [Dariusz Sawicki: Grafika komputerowa i wizualizacja, Politechnika Warszawska]

W praktycznej realizacji algorytmu „YX" dla każdej krawędzi wielokąta oblicza się wszystkie punkty przecięcia konturu z poziomymi liniami rastra. Otrzymane punkty przechowywane są w liście, którą następnie sortuje się przy zachowaniu stosownych reguł. Następnie elementy listy grupowane są w pary, odpowiadające poziomym odcinkom kolejnych linii rastra leżących wewnątrz konturu. Dla wszystkich takich odcinków zmieniany jest kolor na stosowny kolor wypełnienia (w procedurze tej należy jednak przeanalizować i uwzględnić punkty wierzchołkowe).

5. Zadania do wykonania

UWAGA: Zamieszczona poniżej lista ma na celu zademonstrowanie zakresu tematycznego i stopnia trudności zdań realizowanych w trakcie laboratorium. Dokładne treści zadań zostaną podane przez osobę prowadzącą zajęcia i mogą w pewnym stopniu różnić się od tutaj przedstawionych.

Zad. 1. Plik **bresenham_line.m** zawiera kod funkcji implementującej algorytm Bresenhama dla odcinków o nachyleniu od 0º do 45º (opis w punkcie 2.1 instrukcji). Do jej uruchomia z odpowiednimi parametrami służy skrypt zapisany w pliku **bresenham_line_skrypt.m**. Zadanie polega na modyfikacji funkcji i skryptu w taki sposób, aby możliwe było wyświetlenie odcinka o początku w punkcie (1, 1) i końcu w punkcie (4, 6), czyli o kącie nachylenia z zakresu 45-90 stopni. **Podpowiedź:** w rozwiązaniu można posłużyć się symetrią wobec początku odcinka lub skorzystać z uwag zamieszczonych pod koniec teoretycznego opisu algorytmu w punkcie 2.1.

W sprawozdaniu zamieścić:

- *kod zmodyfikowanej funkcji;*
- *rysunek demonstrujący jej działanie.*

Zad. 2. Plik **bresenham_circle.m** zawiera szkielet funkcji służącej do resteryzacji okręgu. Do jej wywołania służy skrypt **bresenham_circle_skrypt.m**. Celem zadania jest uzupełnienie funkcji **bresenham_circle** na podstawie opisu algorytmu w punkcie 2.2 instrukcji oraz pseudokodu z listingu 3. **Podpowiedź:** ze względu na znaczne podobieństwo algorytmów warto wzorować się na kodzie funkcji **bresenham_line** z zadania 1.

W sprawozdaniu zamieścić:

- *kod funkcji;*
- *rysunki pokazujące jej działanie dla trzech okręgów o różnych promieniach. Do sprawozdania należy także wpisać informacje o testowanych okręgach.*

Zad. 3. W pliku **cohen_sutherland.m** znajduje się funkcja służąca do przycinania odcinków algorytmem Cohena-Sutherlanda (szczegółowy opis w punkcie 3 instrukcji). Do jej wywołania można użyć skryptu **cohen_sutherland_skrypt.m**. W obecnej wersji funkcja przycina tylko względem górnej granicy obszaru widocznego i ma niedokończoną procedurę kodowania końców odcinka. Zadanie polega na takiej modyfikacji kodu, aby realizował on przycinanie względem wszystkich czterech brzegów okna widoku.

W sprawozdaniu zamieścić:

- *kod zmodyfikowanej funkcji;*
- *rysunki pokazujące działanie algorytmu dla różnych odcinków – proszę przetestować 4 różne odcinki obrazujące przycinanie przez poszczególne boki okna (po jednym na każdy brzeg okna) oraz przynajmniej 2 odcinki przechodzące przez okno widoku i wymagające przycinania z obydwu stron. Informacje o testowanych odcinkach wpisać do sprawozdania.*

Zad. 4. Uzupełnić rekurencyjną funkcję **sianie.m** realizującą algorytm wypełniania obszaru przez sianie (opis w punkcie 4, pseudokod na listingu 5). Następnie zmodyfikować skrypt **sianie_skrypt.m** w taki sposób, aby możliwe było wypełnienie okręgu z zadania 2. W tym celu trzeba najpierw uruchomić skrypt z zadania 2 (**bresenham_circle_skrypt.m**), aby otrzymać obraz okręgu po rasteryzacji, a następnie – mając wciąż otwarty rysunek – należy uruchomić odpowiednio zmodyfikowany skrypt **sianie_skrypt.m**.

W sprawozdaniu zamieścić:

- *kody zmodyfikowanych funkcji i skryptu;*
- *rysunki pokazujące działanie algorytmu wypełniania dla dwóch różnych okręgów.*

6. Dodatek

W *i*-tym kroku algorytmu Bresenhama, po wypełnieniu piksela o środku w punkcie $P_i(x_i, y_i)$, ustalana jest pozycja piksela do narysowania w kolejnym kroku (*i*+1). Dla odcinka nachyleniu od 0º do 45º wybór jest ograniczony do dwóch pikseli, o środkach w punktach:

- $S_{i+1}(x_i + 1, y_i)$, dla którego współrzędna *x*-owa jest większa o 1 od wartości dla obecnego punktu, zaś współrzędna *y*-owa pozostaje bez zmian;
- $T_{i+1}(x_i + 1, y_i + 1)$, dla którego zarówno współrzędna *x*-owa, jak i *y*-owa są większe o 1 w stosunku do obecnego punktu.

Oznacza to, że piksel rysowany w kolejnym kroku algorytmu na pewno będzie miał współrzędną *x* większą o jeden, a niepewność dotyczy jedynie współrzędnej *y*. Wiadomo również, że dla kroku *i*+1 rzeczywista prosta przechodziłaby przez punkt $Q_{i+1}(x_q, y_q)$. Współrzędna *x*-owa tego punktu wynosić będzie $x_i + 1$, zaś współrzędną *y*-ową możemy wyznaczyć z równania prostej przechodzącej przez punkty P_p oraz P_k:

$$y_q = \frac{dy}{dx}(x_q - x_p) + y_p \tag{6.1}$$

$$y_q = \frac{dy}{dx}(x_i + 1 - x_p) + y_p \tag{6.2}$$

Łatwo zauważyć, że optymalnym wyborem w kroku *i*+1 będzie piksel, dla którego popełnimy mniejszy błąd wobec punktu Q_{i+1}. Oznaczmy zatem przez *t* oraz *s* odległości punktu Q_{i+1} odpowiednio od punkt T_{i+1} oraz S_{i+1}. Łatwo możemy wyznaczyć te odległości korzystając z (6.2):

$$t = d(T_{i+1}, Q_{i+1}) = (y_i + 1) - y_q = y_i + 1 - \frac{dy}{dx}\left(x_i + 1 - x_p\right) - y_p \tag{6.3}$$

$$s = d(Q_{i+1}, S_{i+1}) = y_q - y_i = \frac{dy}{dx}\left(x_i + 1 - x_p\right) + y_p - y_i \tag{6.4}$$

Ponieważ zależy nam na porównaniu odległości *s* oraz *t* odejmijmy je od siebie:

$$s - t = \left[\frac{dy}{dx}\left(x_i + 1 - x_p\right) + y_p - y_i\right] - \left[y_i + 1 - \frac{dy}{dx}\left(x_i + 1 - x_p\right) - y_p\right]$$

$$s - t = 2\frac{dy}{dx}\left(x_i + 1 - x_p\right) + 2y_p - 2y_i - 1 \tag{6.5}$$

Oznaczmy sobie dodatkowo naszą różnicę jako d_{i+1} i przemnóżmy przez *dx*:

$$d_{i+1} = (s - t)dx = 2dy\left(x_i + 1 - x_p\right) + 2y_p dx - 2y_i dx - dx \tag{6.6}$$

Uzyskaliśmy w ten sposób dość złożoną zależność, którą można określić mianem zmiennej decyzyjnej – jeśli różnica będzie mniejsza od zera wówczas na pewno odległość pomiędzy punktem S_{i+1}, a punktem Q_{i+1} jest mniejsza niż odległość pomiędzy punktem T_{i+1}, a punktem Q_{i+1}. Oznacza to, że w *i-tym* kroku algorytmu na podstawie zmiennej decyzyjnej d_{i+1} wybierzemy punkt S_{i+1}. W przeciwnym wypadku, czyli w sytuacji, w której nasza różnica (zmienna decyzyjna) byłaby dodatnia (lub zerowa) wybierzemy punkt T_{i+1}. Na pierwszy rzut oka może się wydawać, że wyznaczona w (6.6) wartość zmiennej decyzyjnej niej jest zbyt „przyjazna" i prosta do wyliczenia. Zwróćmy jednak uwagę, że różnicę d_{i+1} wyznaczamy w *i-tym* kroku algorytmu w celu ustalenia, w jakim punkcie będziemy znajdować się w kolejnym *i+1* kroku. Korzystając z wyprowadzonej zależności (6.6), zapiszmy sobie wartość zmiennej decyzyjne d_i, czyli wartości różnicy,

która wyliczana była we wcześniejszym *i*-1 kroku algorytmu w celu ustalenia punktu, w którym znaleźliśmy się w *i-tym* kroku (czyli aktualnego punktu P_i):

$$d_i = 2dy(x_i - x_p) + 2y_p dx - 2y_{i-1}dx - dx \tag{6.7}$$

Policzmy sobie teraz różnicę pomiędzy d_{i+1} a d_i, czyli wartość o jaką zmienia się zmienna decyzyjna przy przejściu do kolejnego kroku algorytmu:

$$d_{i+1} - d_i = \left[2dy(x_i + 1 - x_p) + 2y_p dx - 2y_i dx - dx\right] - \left[2dy(x_i - x_p) + 2y_p dx - 2y_{i-1}dx - dx\right]$$

$$d_{i+1} - d_i = 2dyx_i + 2dy - 2dyx_p + 2y_p dx - 2y_i dx - dx - 2dyx_i + 2dyx_p - 2y_p dx + 2y_{i-1}dx + dx$$

$$d_{i+1} - d_i = -2y_i dx + 2y_{i-1}dx + 2dy = 2dx(y_{i-1} - y_i) + 2dy \tag{6.8}$$

Widzimy zatem, że w każdym kolejnym kroku algorytmu zmienna decyzyjna ulega modyfikacji o pewną stałą wartość, która zależy bezpośrednio od relacji współrzędnych *y-owych* w kolejnych krokach algorytmu.

Załóżmy, że w poprzedzającym *i*-1 kroku, gdy decydowaliśmy o wyborze punktu $P_i(x_i, y_i)$ zmienna decyzyjna d_i miała wartość ujemną. Wówczas relacja pomiędzy współrzędną *y-ową* punktu, w którym znajdowaliśmy się w kroku *i*-1 (symbolicznie oznaczoną przez y_{i-1}) a współrzędną *y-ową* punktu P_i (czyli współrzędną y_i) jest następująca: $y_i = y_{i-1}$. Wówczas na podstawi (6.8) możemy zapisać:

$$d_{i+1} - d_i = 2dx(y_{i-1} - y_i) + 2dy = 2dy \tag{6.9}$$

Wyrażenie (6.9) możemy zaś przekształcić do postaci:

$$d_{i+1} = d_i + 2dy \tag{6.10}$$

Jeśli założymy natomiast, że w kroku *i*-1, gdy decydowaliśmy o wyborze punktu $P_i(x_i, y_i)$ zmienna decyzyjna d_i miała wartość dodatnią, wówczas relacja pomiędzy współrzędną *y*-ową punktu, w którym znajdowaliśmy się w kroku *i*-1 (oznaczoną przez y_{i-1}) a współrzędną *y-ową* punktu P_i (oznaczoną y_i) jest następująca: $y_i = y_{i-1} + 1$. Wówczas na podstawie (6.8) możemy zapisać:

$$d_{i+1} - d_i = 2dx(y_{i-1} - y_i) + 2dy = 2dy - 2dx \tag{6.11}$$

Wyrażenie (6.11) możemy zaś przekształcić do postaci:

$$d_{i+1} = d_i + 2(dy - dx) \tag{6.12}$$

W ten sposób przy pomocy dwóch prostych wyrażeń (6.10) oraz (6.12) mamy receptę na modyfikację zmiennej decyzyjnej (w zależności od jej bieżącej wartości i aktualnie wybranego punktu) przy przejściu do kolejnego kroku algorytmu. Innymi słowy w każdym *i*-tym kroku algorytmu Bresenhama, znajdując się w określonym punkcie P_i, na podstawie wyliczonej nowej, zaktualizowanej wartości zmiennej decyzyjnej d_{i+1} (ustalenie zmiennej d_{i+1} następuje przez zaktualizowanie bieżącej wartości zmiennej d_i zgodnie z prostymi formułami (6.10) lub (6.12)) dokonujemy wyboru określonego punktu dla kolejnego *i*+1 kroku (S_{i+1} jeśli $d_{i+1} < 0$ lub T_{i+1} jeśli $d_{i+1} \geq 0$). Aby swobodnie poruszać się w tym algorytmie, potrzebujemy jeszcze tylko początkowej, startowej wartości zmiennej decyzyjnej (możemy ją oznaczyć przez d_p). Zdołamy ją jednak wyznaczyć korzystając z zależności (6.7) oraz wiedząc, że zaczynamy od punktu początkowego $P_p(x_p, y_p)$:

$$d_p = 2dy(x_p + 1 - x_p) + 2y_p dx - 2y_p dx - dx = 2dy - dx \tag{6.13}$$

7. Literatura

1. James D. Foley: *Wprowadzenie do grafiki komputerowej*, WNT, 2001.

2. J. Zabrodzki: *Grafika komputerowa*, WNT, 1994

3. Alois Zingl: *The Beauty of Bresenham's Algorithm*, http://members.chello.at/~easyfilter/bresenham.html