xxx

Lecture Notes in Computer Graphics

2D graphics

Piotr Fulmański

Łódź, 4 kwietnia 2016

Spis treści

Spis treści iii

Przedmowa v

1 Podstawowe algorytmy rastrowe: rysowanie odcinka i okręgu 1

1.1 Pierwsze podejście do rysowania odcinka 1

1.2 Drugie podejście do rysowania odcinka 10

1.2.1 Funkcja uwikłana 10

1.2.2 Algorytm rysowania dowolnej krzywej 13

1.3 Kolejne usprawnienie 15

1.4 Konsekwencje usprawnienia: przypadek odcinka 16

1.4.1 Po pierwsze... 16

1.4.2 Po drugie... 17

1.4.3 Po trzecie... 17

1.5 Uogólnienie na inne oktanty okładu współrzędnych 19

1.6 Rasteryzajca okręgu 23

1.7 Antyaliasing 28

1.8 Wypełnianie wielokątów 29

Bibliografia 31

Spis rysunków 33

Spis tabel 35

Skorowidz 37

Przedmowa

Niniejszy dokument stanowi ucywilizowaną – to znaczy precyzyjniej i czytelniej napisaną – wersję notatek jaki przygotowuję sobie przed każdymi zajeciami. Ponieważ zwykłe kartki lubią się zagubić a ponad to na ogół nie ma ich pod ręką gdy są potrzebne, stąd ich treść bardzo szybko zamieniam na formę elektroniczną. Skoro zaś już taką formę mam, to nie widzę powodu aby nie udostępniać tego co udało mi się zebrać, co też niniejszym czynię. Mam nadzieję, że nie powoduje to u nikogo zaniechania własnych poszukiwać, ale z pewnością ułatwi uporządkowanie tego co na zajeciach było przerabiane.

ROZDZIAŁ 1

Podstawowe algorytmy rastrowe: rysowanie odcinka i okręgu

1.1 Pierwsze podejście do rysowania odcinka

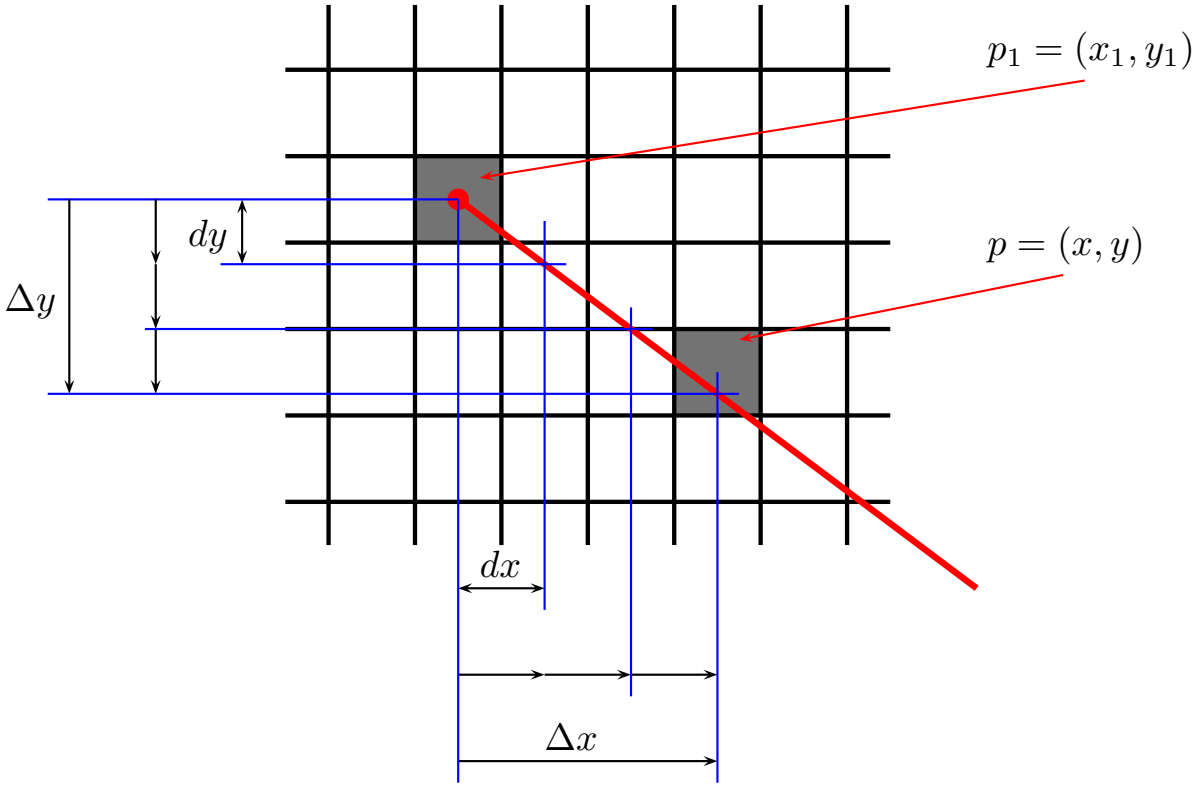
Punkt wyjscia do rozważań w tym rozdziale stanowi sytuacja przedstawiona na rysunku 1.1.

Ogólny schemat algorytmu pozwalającego narysować odcinek przedstawia listing 1.1

```
float y;
int x = x1;
while (x <= x2){
        // Zmienna x wieksza sie o 1 od x1 do x2
        x = x + 1;
        // a y?!
        y = y1 + ?
}
```

Listing 1.1: src/sra/01.java

Problemem jest właściwe dobranie wartości zmiany dla zmiennej y. Posługując się przedstawionym



Rysunek 1.1: Rysunek sytuacyjny, stanowiący punkt odniesienia do rozważań w tym rozdziale. Zauważmy, że piksel ma niezerową wysokość i szerokość a więc najwygodniej współrzędne piksela utożsamiać z jego środkiem. W konsekwencji piksel $p = (x, y)$ ma wymiary $[x - w/2, x + w/2] \times [y - h/2, y + h/2]$, gdzie w i h to odpowiednio szerokość i wysokość piksela. Dla prostoty możemy przyjąć, że oba wspomniane wymiary mają wartość 1

rysunkiem można zauważyć następujące zależności

$$\frac{\Delta x}{\Delta y} = \frac{dx}{dy}$$
$$\Delta x \cdot dy = \Delta y \cdot dx$$
$$\frac{\Delta x \cdot dy}{dx} = \Delta y$$
$$\frac{(x - x1)dy}{dx} = \Delta y$$

Formuła ta określa wielkość zmiany Δy współrzędnej y dla zadanej współrzednej x (tj. współrzędnej $x1$ przesuniętej o Δx). Jeśli względnym numerem kolumny jest nk (tj. jesteśmy w kolumnie o numerze nk licząc względem kolumny 0 w której znajduje się punkt $p1$) to mamy wówczas nk zmian przechodząc od kolumny $x1$ od kolumny x.

O ile wiadomo, że pojedyńczy przyrost (zmiana) x oznaczana jako dx wynosi 1 (tak przynajmniej dla ułatwienia się umawiamy) – przechodzimy przez kolejne kolumny – to przysrost dy wymaga już trochę arytmetyki, gdyż zależy od nachylenia rysowanego odcinka względem osi OX. Przyjmując, że równanie prostej jest postaci

$$y = ax + b$$

wartość współczynnika kierunkowego a można obliczyć jako

$$a = \frac{y_2 - y_1}{x_2 - x_1}.$$

Przy okazji widzimy, że współczynnik kierunkowy dla prostej jest stały (zależy tylko i wyłącznie od współrzędnych punktu początkowego i końcowego, ale nie zależy do x leżącego pomiędzy współrzędnymi $x1$ oraz $x2$), a więc istotnie wszystkie zmiany na osi OY są takie same. Stąd zaś wynika, że

$$\Delta y = a \cdot \Delta x.$$

a w szczególności

$$dy = a \cdot dx.$$

Teraz możemy w końcu uzupełnić nasz kod do postaci pokazanej na listingu 1.2.

```
float y;
int x = x1;
float dy = (y2-y1) / (x2-x1) * dx;
while (x <= x2){
        x = x + 1;
        y = y1 + dy * (x - x1)/dx;
}
```

Listing 1.2: src/sra/02.java

Zauważmy, że będzie on równoważny kodowi z listingu 1.3 (pozbywamy się mnożenia przez dx a następnie dzielenia przez ten sam składnik).

```
float y;
int x = x1;
float dy = (y2-y2) / (x2-x2);
while (x <= x2){
        x = x + 1;
        y = y1 + dy * (x - x1);
```

```
}
```

Listing 1.3: src/sra/03.java

Lepszym rozwiązaniem będzie poniższy kod, gdzie wykonujemy tylko jedno mnożenie przez czynnik dx.

```
float y;
int x = x1;
float dy = (y2-y1) / (x2-x1) * dx;
while (x <= x2){
        x = x + 1;
        y = y + dy;
}
```

Listing 1.4: src/sra/04.java

Zauważmy, że różnica pomiędzy podejściem przedstawionym na listingu 1.4 w porównaniu z 1.3 polega na

- Zwiększeniu wydajności (mniej mnożeń) w przypadku realizacji na urządzeniu jednowątkowym.
- Innym sposobie obliczania wartości y. W rozwiązaniu z listingu 1.3 jest to zawsze przesunięcie względem stałego punktu odniesienia jakim jest y_1. Zatem biorąc dwa dowolne punkty $x_1 \leqslant x_a \leqslant x_b \leqslant x_2$ obliczenia wykonane dla punktu x_a nie mają wpływu na punkt x_b. Co więcej, można je wykonać w dowolnej kolejności a nawet wszystkie w tym samym czasie (równolegle). W przypadku rozwiązaniu z listingu 1.4 obliczenia wykonane dla punktu x_a mają wpływ na punkt x_b i wszelkie błędy popełnione dla punktów wcześniejszych przenoszą się na kolejne punkty.

Jak do tej pory wszystko wydaje się być dobrze z jednym małym wyjątkiem: zmienna y przyjmuje wartości rzeczywiste (porównaj rysunek 1.2). Nie jest to dobra wiadomość jako, że w przypadku rastra można używać jedynie współrzędnych całkowitych. Tak więc otrzymywane wartości rzeczywiste należy zaokrąglić (przybliżyć do najbliższej liczby całkowitej) co w efekcie da obraz podobny do przedstawionego na rysunku 1.3. Otrzymamy wówczas algorytm pokazany na listingu 1.5. Jest to właściwie ten sam kod co na listingu 1.4, ale przedstawiony w postaci funkcji i, co ważniejsze, z zaznaczonymi miejscami, które wprowadzają do obliczeń niedokładności

- niedokładność na etapie przygotowywania danych „bazowych" – obliczona wartości zmiennej a już w tym momencie może nie być dokładna i odbiegać od wartości teoretycznej;
- konieczność zaokrąglenia wartości zmiennoprzecinkowej do liczby całkowitej;
- akumulacja błędów przy przyrostowym zwiększaniu zmiennej y.

```
void drawLine (int x1, int y1, int x2, int y2){
        // Zmienna a nie jest dokladne a jedynie przyblizone.
        float a = (float)(y2 - y1) / (x2 - x1);
        int x = x1;
        float y = y1;

        while(x <= x2){
                // Zaokraglenie zmiennej y.
                setPixel(x, round(y));
                // Zmienna x ma wartosc dokladna - arytmetyka calkowitoliczbowa.
                x = x + 1;
                // Zmienna y ma wartosc przyblizona - arytmetyka zmiennoprzecinkowa.
                // Niestety akumulowanie bledow na skutek operowania
                // przyblizonym 'a' i operacjami dajacymi przyblizone wyniki.
                y = y + a;
        }
}
```

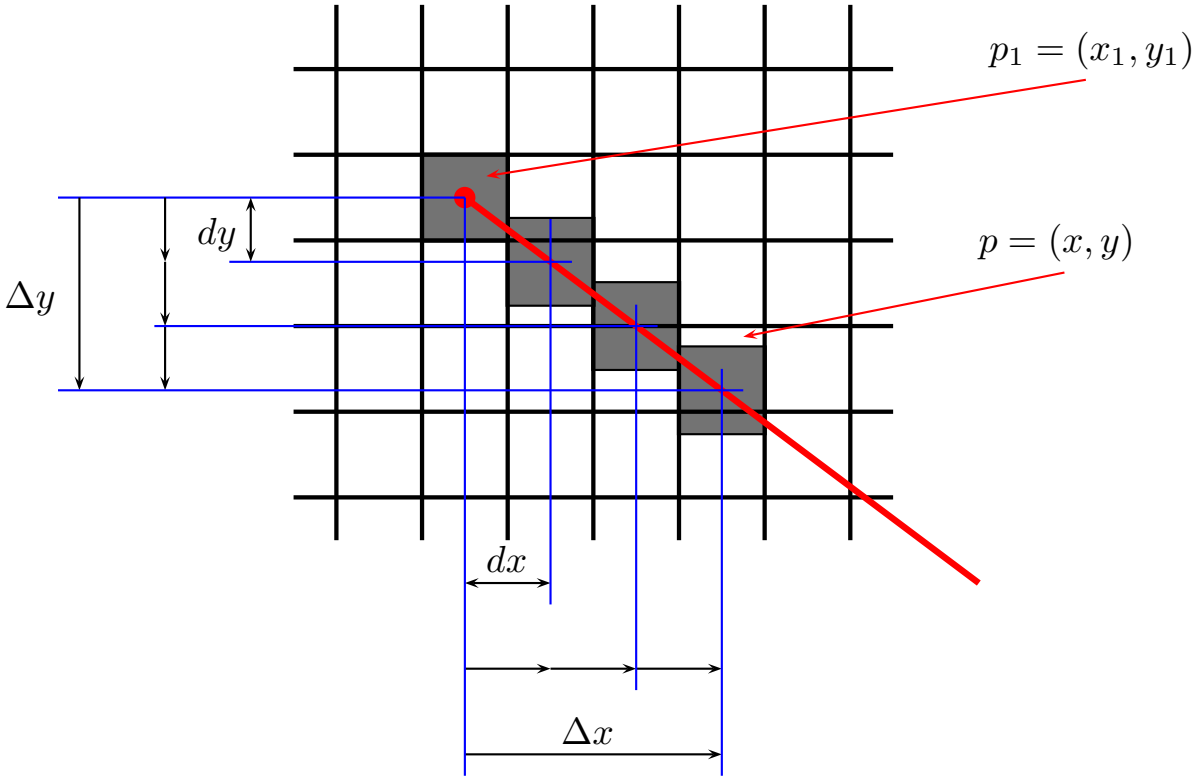
Listing 1.5: src/sra/05.java

Efekt działanie tego algorytmu przedstawia rysunek 1.4.

If you want to know more... 1.1 (Akumulacja błędów zaokrągleń)**.** *Zaokrąglanie to zastąpienie danej liczby inną o mniejszej ilości końcowych cyfr znaczących.*

Zaokrąglanie opiera się na pozycyjnym zapisie liczb i polega na:

1. *ustaleniu dokładności zaokrąglenia tj. na wskazaniu cyfry końcowej, względem której określane jest zaokrąglenie,*
2. *zastąpieniu zerami wszystkich cyfr na prawo od wskazanej cyfry,*
3. *zwiększeniu wskazanej cyfry o jeden, jeśli sąsiednia z prawej cyfra przed usunięciem była większa lub równa 5.*



Rysunek 1.2: Idealny obraz odcinka po rasteryzacji

Jako przykład zaokrąglenia rozważmy liczbę 987,654, która zaokrąglona do setek ma wartość 1000, do dziesiątek wartość 990, do jedności wartość 988, do części dziesiętnych wartość 987,6 i do części setnych 987,65.

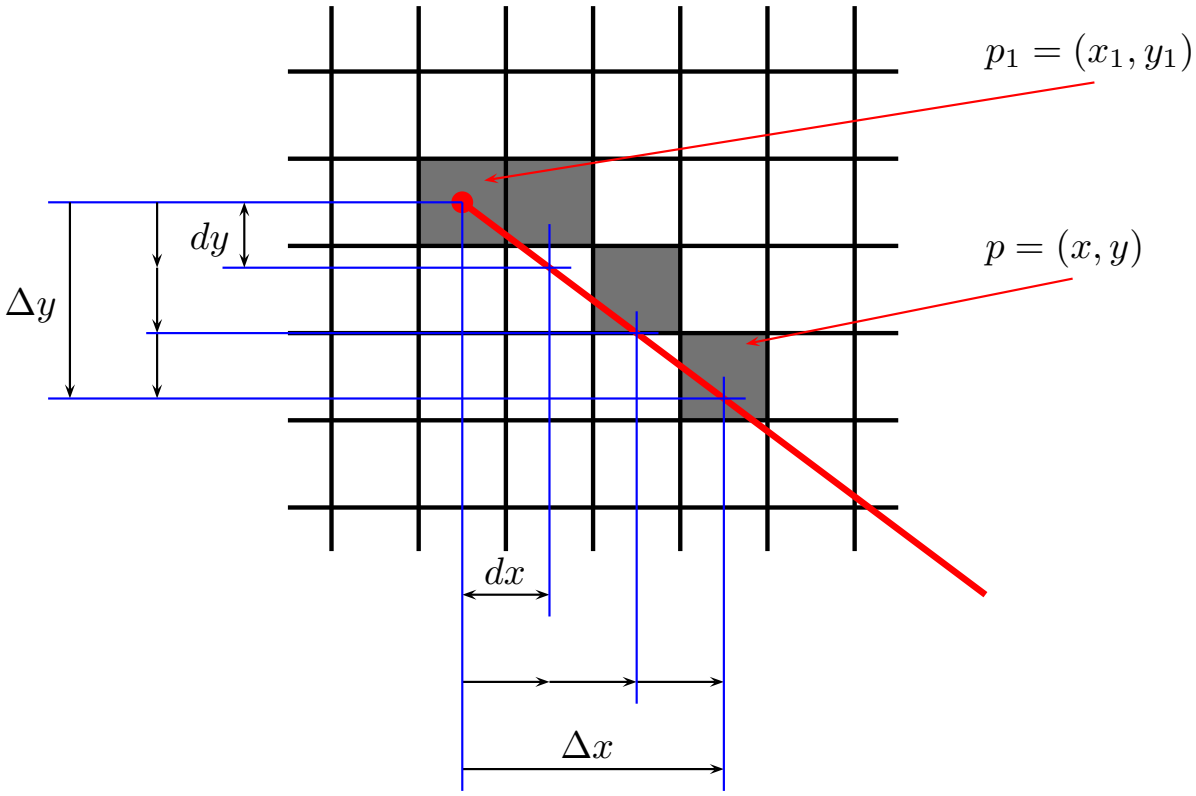
Błąd zaokrąglania (ang. rounding error*) to utrata cyfr znaczących ułamków reprezentujących liczby zmiennopozycyjne, powodowana koniecznością obcięcia wyników obliczeń.*

Błędy zaokrągleń pojawiają się podczas obliczeń na skutek konieczności zaokrąglania obliczonych wartości ze względu na ograniczoną długość słów binarnych, np. dzielenie dwóch liczb wymiernych prowadzi często do konieczności zaokrąglenia powstałej w wyniku dzielenia liczby niewymiernej

$$\frac{1}{3} = 0,33333\ldots \approx 0,33333.$$

Kumulowanie się błędów a czasem i ich powstawanie podczas obliczeń jest efektem ograniczonej dokładności reprezentacji wyniku działań w pamięci maszyny cyfrowej.

Rozważmy następujący przykład w którym wykorzystamy liczbę $x = 2,31$ reprezentowaną z błędem 0,02 oraz liczbę $y = 1,42$ reprezentowaną z błędem 0,03. Jeśli teraz rozważymy elemen-



Rysunek 1.3: Rzeczywisty obraz odcinka po rasteryzacji

Rysunek 1.4: Rzeczywisty obraz odcinka $(0, 100) - (400, 101)$ po rasteryzacji będący wynikiem akumulacji błędów zmiennoprzecinkowych

tarną operację $x - y$ wówczas zauważamy, że

- *najmniejsza możliwa wartość różnicy x i y wynosi $(2,31 - 0,02) - (1,42 + 0,03) = 0,84$;*
- *największa możliwa wartość różnicy x i y wynosi $(2,31 + 0,02) - (1,42 - 0,03) = 0,94$;*

W konsekwencji widzimy, że wartość różnicy mieści się w przedziale $[0,84, 0,94]$ a więc wynikiem jest liczba 0,89 reprezentowana z błędem 0,05 a więc większym, niż zadane liczby.

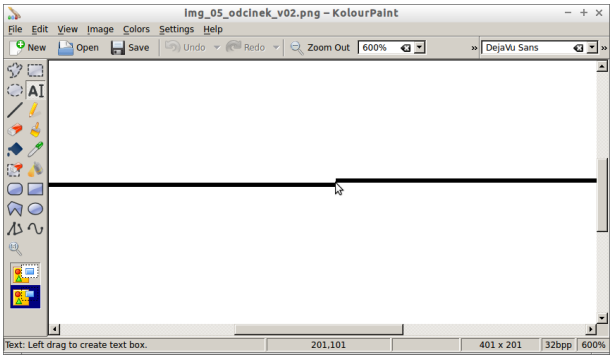
Ograniczona dokładność będzie miała także wpływ np. na wyniki dodawania do siebie liczby bardzo dużej i bardzo małej. Dla przykładu, gdy posługujemy się pozycyjnym zmiennoprzecinkowym systemem liczbowym z mantysą o długości 6 cyfr to możemy w nim zapisać zarówno liczbę 12300,0 jak i 0,00123. Niestety wynik dodawania 12300,00123 zostanie zapisany jako 12300,0.

Aby uniknąć akumulacji błędów możemy w tym celu nieznacznie zmienić nasz program, jak pokazano na listingu 1.6. Efekt działania przedstawia rysunek 1.5.

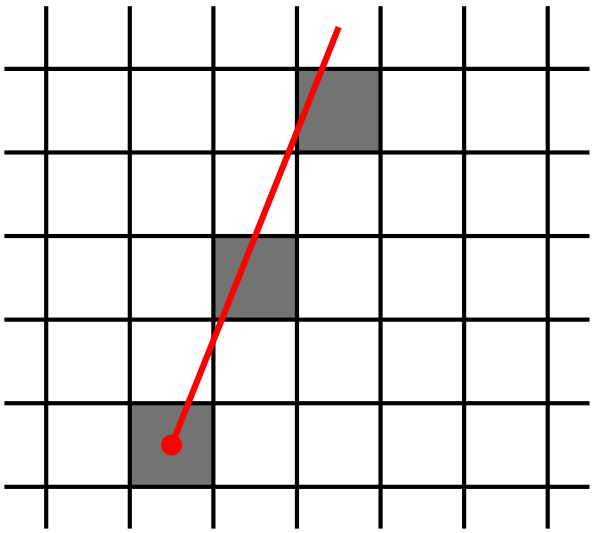
```
void drawLine (int x1, int y1, int x2, int y2){
        float a = (float)(y2 - y1) / (x2 - x1);
        int x = x1;
        int y = y1;
        float error = 0.0

        while(x <= x2){
                setPixel(x, y);
                x = x + 1;
                error += a
                if (error >= 0.5){
                        y = y + 1;
                        error -=1.0;
                }
        }
}
```

Listing 1.6: src/sra/05_02.java

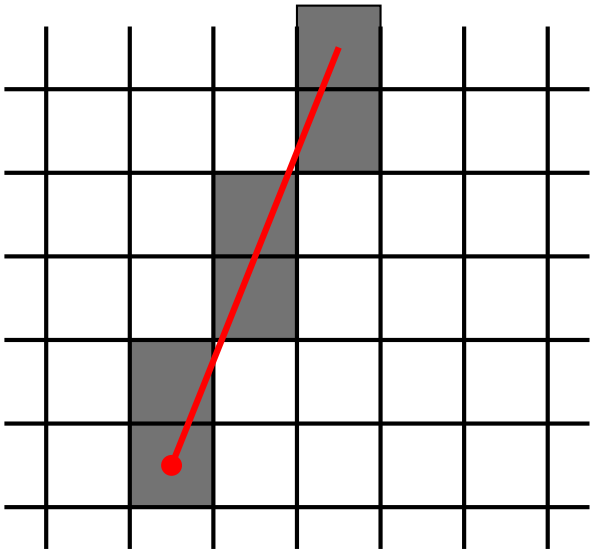
Problem stanowi fakt, iż działanie algorytmu ograniczono nakładając warunek $|a| \leqslant 1$, a więc stosujemy go do odcinków o kącie nachylenia do osi OX mniejszym niż 45 stopni. Przy kącie nachylenia większym, wartości wzdłuż osi OY muszą rosnąć szybciej niż wzdłuż osi OX. Oznaczałoby to albo powstawanie „dziur" (rysunek 1.6) albo konieczność „zamalowywania" więcej niż jednego piksela w kolumnie (rysunek 1.7).



Rysunek 1.5: Rzeczywisty obraz odcinka $(0, 100) - (400, 101)$ po rasteryzacji z uwzględnieniem akumulacji błędów (algorytm 1.6). Porównujac ten rysunek z rysunkiem 1.4 widzimy w jego środkowej części „schodek” co wydaje się być zgodne z naszą intuicją



Rysunek 1.6: Przypadek gdy $|a| > 1$ – powstawanie „dziur”



Rysunek 1.7: Przypadek gdy $|a| > 1$ – więcej niż jeden piksel w kolumnie

x	-4	-3	-2	-1	0	1	2	3	4
$y = g(x)$	-1	-0.5	0	0.5	1	1.5	2	2.5	3

Tabela 1.1: Wartości funkcji $g(x) = ax + b$ dla $a = 0.5$, $b = 1$ w wybranych punktach

I właśnie dlatego w dalszej części przyjmujemy, że nachylanie odcinka (a w ogólności krzywej), a więc kąt pomiędzy styczną a osią OX, nie może przekraczać 45 stopni; jeśli odcinek (krzywa) może zostać opisana funkcją postaci $y = g(x)$ to musi być spełniony warunek $0 \leqslant f'(x) \leqslant 1$. Dodatkowo w przypadku krzywej zakładać będziemy, że w rozważanym zakresie jest monotoniczna, tj.: nierosnąca, albo niemalejąca.

1.2 Drugie podejście do rysowania odcinka

1.2.1 Funkcja uwikłana

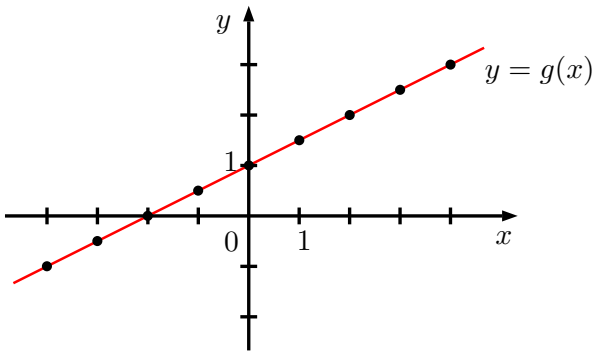
Dla funkcji postaci

$$y = g(x) = ax + b$$

ustalmy wartości parametrów a i b i policzmy wartość $g(x)$ dla kilku wybranych wartości x (porównaj tabela 1.1 oraz rysunek 1.8)

Rozważmy teraz funkcję

$$f(x, y) = y - ax - b$$



Rysunek 1.8: Wykres funkcji $g(x)$ dla wartości z tabeli 1.1

x	-4	-3	-2	-1	0	1	2	3	4
$y = g(x)$	-1	-0.5	0	0.5	1	1.5	2	2.5	3
$f(x, y)$	0	0	0	0	0	0	0	0	0

Tabela 1.2: Wartości funkcji $f(x, y)$ w punktach z tabeli 1.1

x	-4	-3	-2	-1	0	1	2	3	4
$y = g(x)$	-1	-0.5	0	0.5	1	1.5	2	2.5	3
$f(x, y + 1)$	1	1	1	1	1	1	1	1	1
$f(x, y + 0.5)$	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
$f(x, y)$	0	0	0	0	0	0	0	0	0
$f(x, y - 0.5)$	-0.5	-0.5	-0.5	-0.5	-0.5	-0.5	-0.5	-0.5	-0.5
$f(x, y - 1)$	-1	-1	-1	-1	-1	-1	-1	-1	-1

Tabela 1.3: Wartości funkcji $f(x, y)$ w wybranych punktach

i dla otrzymanych w tabeli 1.1 wartości x i y obliczmy wartość funkcji $f(x, y)$ (porównaj tabela 1.2 i rysunek 1.8). Pozostawiając wartości x jak poprzednio zmieńmy wartości y (porównaj tabela 1.3 i rysunek 1.9). Jak widać, jedynie dla punktów leżących dokładnie na prostej otrzymujemy

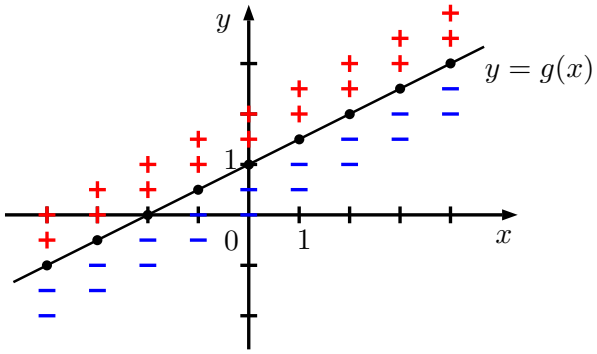
$$f(x, y) = 0.$$

Dla punktów leżących pod prostą otrzymujemy wartości ujemne, a dla leżacych nad prostą wartości dodatnie. I jest to pewna własność tzw. funkcji uwikłanej. Na przykład wartości funkcji uwikłanej opisującej okrąg

$$f(x, y) = x^2 + y^2 - r^2$$

są dodatnie dla punktów leżących poza okręgiem i ujemne dla punktów leżących wewnątrz okręgu.

Przy tej okazji zauważmy, że prosta może mieć dwie strony dodatnie. Otóż równanie prostej



Rysunek 1.9: Wykres znaku wartości funkcji $f(x, y)$ dla wartości z tabeli 1.3; wartości ujemne oznaczono symbolem '−', dodatnie symbolem '+' a czarną kropką wartości równe zero

postaci

$$y = ax + b$$

możemy przekształcić do postaci

$$y - ax - b = 0$$

lub postaci równoważnej

$$0 = ax + b - y.$$

W obu przypadkach uzyskujemy tą samą prostą. Zauważmy jednak, że w obu przypadkach inna półpłaszczyzna będzie półpłaszczyzną dodatnią. I tak oznaczając

$$f_1(x, y) = y - ax - b$$

$$f_2(x, y) = ax + b - y$$

i podstawiając np. punkt $(1/a, 1)$ uzyskujemy

$$f_1(1/a, 1) = -b$$

$$f_2(1/a, 1) = +b$$

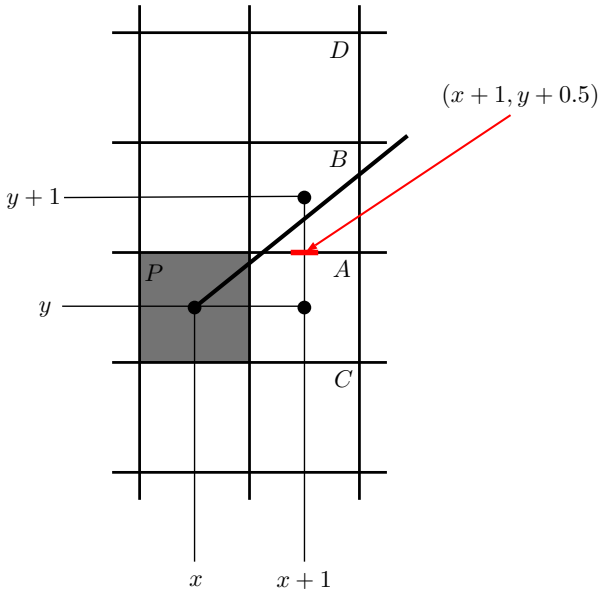
Dokładnie mówiąc, dla prostej opisanej równaniem w postaci ogólnej

$$Ax + By + C = 0,$$

wektor kierunkowy ma postać $[-B, A]$, wektor o współrzędnych $[A, B]$ jest wektorem prostopadłym wskazującym dodatnią półpłaszczyznę. Drugi z wektorów prostopadłych postaci $[-A, -B]$ wskazuje ujemną półpłaszczyznę*.

Teraz możemy przejść do kolejnej wersji algorytmu.

*Mając dany wektor (x, y) wektor prostopadły uzyskujemy zamieniając współrzędne wektora miejscami i mnożąc jedną z nich przez -1.



Rysunek 1.10: Wybór kolejnego piksela

1.2.2 Algorytm rysowania dowolnej krzywej

Załóżmy, że krzywa w przedziale $[x_1, x_2]$ spełnia podane wcześniej założenia, tj.

- Nachylanie odcinka (a w ogólności krzywej), a więc kąt pomiędzy styczną a osią OX, nie może przekraczać 45 stopni; jeśli odcinek (krzywa) może zostać opisana funkcją postaci $y = g(x)$ to musi być spełniony warunek $0 \leqslant g'(x) \leqslant 1$.
- Krzywa w rozważanym zakresie jest monotoniczna, tj.: nierosnąca, albo niemalejąca.

Algorytm zaczyna działanie od punktu $p_1 = (x_1, y_1)$ i „stawia" piksel o tych współrzędnych[†] (szary prostokąt na rysunku 1.10). Następnie współrzędna x jest zwiększana o 1, czyli kolejny piksel ma współrzędną

$$x = x_1 + 1.$$

Przechodzimy teraz do najważniejszej części: należy wybrać wartość współrzędnej y odpowiadającej współrzędnej x. Dzięki przyjętym warunkom wybór ten ogranicza się do pikseli A i B[‡] i determinuje go punkt przecięcia krzywej z linią pionową x_1+1. Jeśli punkt ten jest bliżej punktu A (środka piksela A) to wybierany jest punkt A, a w przeciwnym razie punkt B.

[†]W ogólności opisane rozumowanie prawdziwe jest dla dowolnego punktu $p = (x, y)$ z prostej i dlatego na rysunku 1.10 pominięto indeks 1 pisząc np. x zamiast x_1.

[‡]Piksel C nie jest brany pod uwagę, gdyż wówczas nie byłby spełniony warunek 2. Rozważanie piksela D przeczyłoby warunkowi 1.

Poprzednie zdanie implikuje konieczność liczenia odległości pmiędzy punktami (punkt przecięcia a punkt środkowy piksela). Liczenie odległości może być dobrym rozwiązaniem podczas implementacji antyaliasingu. Jeśli jednak chcemy narysować krzywą „dokładnie" lub zależy nam na szybkości działania to musimy poszukać innych metod – metod, które działanie swoje opierają na liczbach całkowitych. Na rysunku 1.10 poziomą czerwoną kreską zaznaczony został punkt środkowy (o współrzędnych $(x_1+1, y_1+0.5)$) rozgraniczający piksele A i B. Jeśli punkt środkowy znajduje się powyżej punktu przecięcia krzywej z linią pionową x_1+1, to wybierany zostaje punkt A, w przeciwnym razie (co ma miejsce w przykładzie) punkt B.

Do stwierdzenia która z tych sytuacji ma miejsce wykorzystana zostanie opisywana cecha funkcji uwikłanej. Jeśli zależność

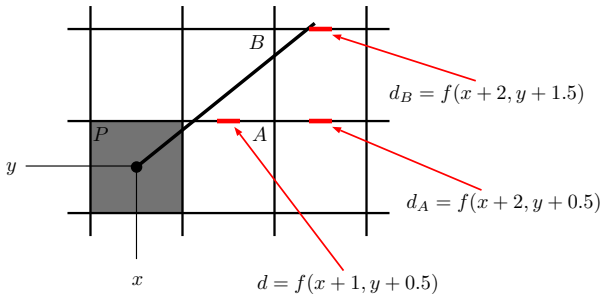
$$f(x, y) > 0$$

zachodzi dla wszystkich punktów powyżej krzywej wówczas w przypadku gdy wartość funkcji w punkcie środkowym $(x_1 + 1, y_1 + 0.5)$ jest mniejsza od zera (oznacza to, że punkt ten jest pod krzywą), to wybrany zostanie punkt B (konieczność zwiększenia zarówno współrzędnej x jak i y), w przeciwnym razie punkt A (zwiększamy tylko współrzedą x).

```
1  void drawCurve (int x1, int y1, int x2, function f(x,y)){
2          float d;
3          int x = x1;
4          int y = y1;
5
6          while(x <= x2){
7                  setPixel(x, y);
8                  d = f(x+1, y+0.5)
9                  if (d < 0){
10                         y += 1;
11                 }
12                 x += 1;
13         }
14 }
```

Listing 1.7: src/sra/06.java

Zauważmy, że podany algorytm jest słuszny dla dowolnej krzywej o ile spełnia ona podane wcześniej założenia. Bez wzgledu na to, czy mamy do czynienia z odcinkiem czy dowolną krzywą to nie jest potrzebna wartość y_2, gdyż algorytm jej nie używa. Trzeba natomiast do funkcji przekazać funkcję f opisującą krzywą w postaci uwikłanej.



Rysunek 1.11: Wartości w punktach środkowych dla piksela P, A oraz B

1.3 Kolejne usprawnienie

Podamy teraz usprawnie algorytmu z podrozdziału 1.2.2. W tym momencie przyniesie ono więcej szkody niż pożytku, ale w późniejszym czasie okaże sie bardzo pomocne.

Zauważmy, że wykonanie kodu z listingu 1.7 wymaga wielokrotnego obliczania wartości funkcji decyzyjnej $f(x, y)$ w linii 8. Ponieważ jednak współrzędne x i y rysowanych punktów zmieniają się o znane wartości, więc wartość funkcji w kolejnym kroku można (próbować) przewidzieć. Wartość funkcji f w punkcie środkowym wzgledem dowolnego punktu (x, y) jak wiemy wynosi (porównaj rysunek 1.11)

$$d = f(x + 1, y + 0.5)$$

w punkcie A

$$d_A = f(x + 2, y + 0.5)$$

natomiast w punkcie B

$$d_B = f(x + 2, y + 1.5)$$

Tak więc przy wyborze punktu A wartość funkcji decyzyjnej f a tym samym zmiennej decyzyjnej d zmieni się o

$$delta_A = d_A - d,$$

po wybraniu zaś punku B o

$$delta_B = d_B - d.$$

Wykorzystując nowo wprowadzone wielkości otrzymujemy kod jak na listingu 1.8.

```
1 void drawCurve (int x1, int y1, int x2, function f(x,y)){
2         float d = f(x1+1, y1+0.5)
3         int x = x1;
4         int y = y1;
```

```
5
6           while(x <= x2){
7                   setPixel(x, y);
8                   if (d < 0){
9                           d += f(x+2, y+1.5)-f(x+1, y+0.5)
10                          y += 1;
11                  } else{
12                          d += f(x+2, y+0.5)-f(x+1, y+0.5)
13                  }
14                  x += 1;
15          }
16  }
```

Listing 1.8: src/sra/07.java

Zaproponowana modyfikacja wydaje się wcale niczego nie usprawniać: zamiast jednokrotnego obliczania wartości funkcji f w każdej iteracji, teraz obliczamy dwie wartości (linia 9 lub 12). Niebawem jednak okaże się, że wprowadzona zmiana jest bardzo korzystna.

1.4 Konsekwencje usprawnienia: przypadek odcinka

W szczególnym przypadku, gdy krzywa jest odcinkiem algorytm podany w poprzednim podrozdziale (1.3) można znacząco uprościć.

1.4.1 Po pierwsze...

...wartość zmiennej d jest równa

$$f(x+1, y+0.5) = (y-0.5) - a(x+1) - b = y - a(x+1) - b - 0.5.$$

Wartości różnic $delta_A$ oraz $delta_B$.

$$delta_A = f(x+2, y+0.5) - f(x+1, y+0.5) = (y+0.5) - a(x+2) - b - (y+0.5) + a(x+1) + b = -a$$

$$delta_B = f(x+2, y+1.5) - f(x+1, y+0.5) = (y+1.5) - a(x+2) - b - (y+0.5) + a(x+1) + b = -a - 1$$

Ponieważ

$$a = \frac{y_2 - y_1}{x_2 - x_1}$$

więc wartości różnic $delta_A$ oraz $delta_B$ są stałe i zależą wyłącznie od współrzędnych końców odcinka. Takie różnice, tj. różnice, które są stałe, nazywać będziemy różnicami pierwszego rzędu. Jeśli natomiast zależą one od współrzędnych (co będzie miało miejsce w dalszej części przy okazji omawiania rasteryzaji okręgu), to wówczas można policzyć odpowiednie przyrosty dla tychże przyrostów – takie przyrosty nazywamy różnicami drugiego rzędu. Jeśli i one (nadal) zależą od współrzędnych, to możemy obliczać kolejne różnice.

1.4.2 Po drugie...

...dla algorytmu rysowania odcinka nie ma zupełnie znaczenia w jakim punkcie zaczepiony jest odcinek (gdzie jest położny): współczynnik kierunkowy będzie taki sam, kolejne piksele tworzące wykres odcinka będą względnie takie same. Dlatego w dalszej części przyjmiemy, że współrzędne (x_1, y_1) odcinka wynoszą $(0, 0)$ czyli odcinek zaczyna się w początku układu współrzędych. Implementacja algorytmu będzie oczywiście uwzgledniała dowolne inne położenie i będzie dokonywane odpowiednie przesunięci. Dlatego w dalszej części zamiast funkcji postaci

$$f(x, y) = y - ax - b = y - \frac{y_2 - y_1}{x_2 - x_1}x - b$$

używać będziemy

$$f(x, y) = y - ax = y - \frac{y_2 - y_1}{x_2 - x_1}x$$

1.4.3 Po trzecie...

...powiedzmy sobie szczerze: dotychczasowe obliczenia są matematycznie poprawne, ale mało praktyczne z punktu widzenia implementacyjnego.

- Ponieważ obliczenie wartości a wymaga dzielenia, więc wymusza obliczenia na liczbach rzeczywistych reprezentowanych w komputerze jako liczby zmiennoprzecinkowe a więc zwykle jako przybliżone a nie dokładne.

- Obliczenie wartości początkowej funkcji f wymaga wielu operacji w tym mnożenia i dodania liczby 0.5 a więc znowu mamy liczby rzeczywiste a liczb rzeczywistych w komputerach nie lubimy.

Zauważmy, że funkcja

$$f(x, y) = y - ax = y - \frac{y_2 - y_1}{x_2 - x_1}x$$

x	2	3
y	2	3
$f^*(x, y-1)$	5	6
$f^*(x, y-0.5)$	8	9
$f^*(x, y)$	8	9
$f^*(x, y+0.5)$	8	9
$f^*(x, y+1)$	8	9

Tabela 1.4: Wartości funkcji f^*

wyznacza taką samą prostą co funkcja w której każdy składnik został przemnożony przez stałą (w naszym przypadku będzie to $2(x_2 - x_1)$)

$$f^*(x, y) = 2(x_2 - x_1)y - 2(y_2 - y_1)x$$

gdyż dokładnie dla takich samych par (x, y) przyjmuje wartość równą 0 dla pozostałych zaś zachowuje wartość znaku (porównaj tabela 1.4) a więc z naszego punktu widzenia jest tak samo dobra, a jak się okaże za chwile, nawet lepsza.

Przyjmując oznaczenia

$$dX = x_2 - x_1$$
$$dY = y_2 - y_1$$

policzmy wartości różnic $delta_A$ oraz $delta_B$

$$\begin{aligned} delta_A &= f^*(x+2, y+0.5) - f^*(x+1, y+0.5) \\ &= 2dX(y+0.5) - 2dY(x+2) - 2dX(y+0.5) + 2dY(x+1) = -2dY, \\ delta_B &= f^*(x+2, y+1.5) - f^*(x+1, y+0.5) \\ &= 2dX(y+1.5) - 2dY(x+2) - 2dX(y+0.5) + 2dY(x+1) = 2dX - 2dY, \end{aligned}$$

oraz początkową zmiennej d

$$d = f^*(x+1, y+0.5) = 2dX(y+0.5) - 2dY(x+1) = 2dXy + dX - 2dY(x+1).$$

Wartość d obliczona względem początku rysowanego odcinka a więc względem punktu $(0, 0)$ wynosi

$$d = f^*(0+1, 0+0.5) = 2dX(0+0.5) - 2dY(0+1) = dX - 2dY.$$

Finalną wersję algorytmu przedstawia listing 1.9.

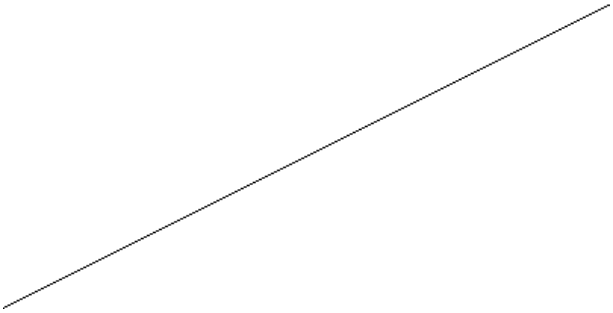
```
void drawLine (int x1, int y1, int x2, int y2){
    int dX = x2 - x1;
    int dY = y2 - y1;

    int d = dX - 2*dY;
    int deltaA = -2*dY;
    int deltaB = 2*dX - 2*dY;
    int x = x1;
    int y = y1;

    while(x <= x2){
        setPixel(x, y);
        if (d < 0){
            d += deltaB
            y += 1;
        } else{
            d += deltaA
        }
        x += 1;
    }
}
```

Listing 1.9: src/sra/08.java

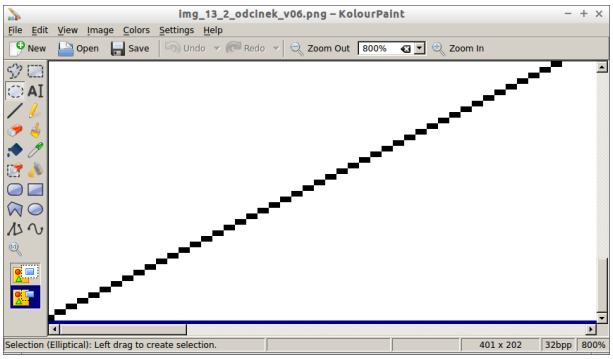
1.5 Uogólnienie na inne oktanty okładu współrzędnych

Podany algorytm będzie działał poprawnie (rysunek 1.12, 1.13) o ile odcinek mieści się w pierwszym oktancie układu współrzędnych (rysunek 1.14). W przeciwnym razie uzyskane efekty mogą odbiegać od oczekiwanych, co ilustruje rysunek 1.15. Narysowanie odcinka w innych oktantach wymaga odpowiedniej zmiany współrzędnych (porównaj rysunek 1.16). W oktantach I, IV, V, VIII moduł nachylenia odcinka mieści się w przedziale 0-45 stopni a więc do jego rasteryzacji można wykorzystać dotychczasowy algorytm pamiętając aby

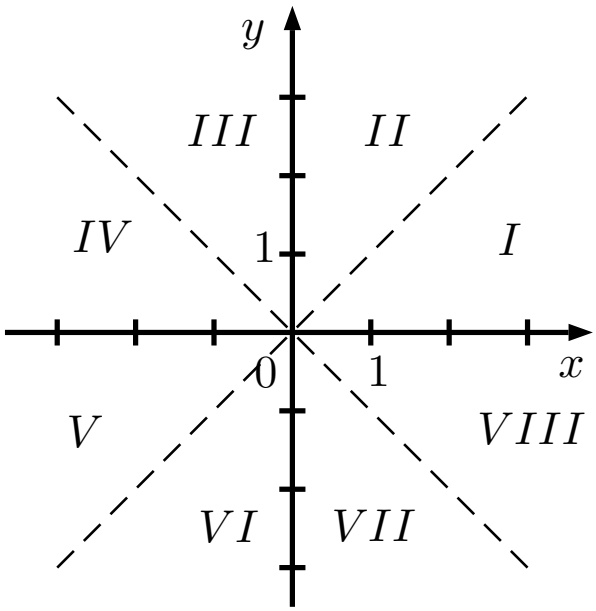
- operować na modułach dX i dY,
- uzależnić przyrosty x i y od znaku, odpowiednio, dX i dY.



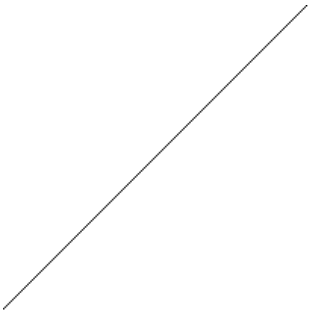
Rysunek 1.12: Przykład poprawnego działania algorytmu z listingu 1.9: wykres odcinka $(0, 1) - (400, 201)$ leżącego na prostej $y = 0.5x + 1$



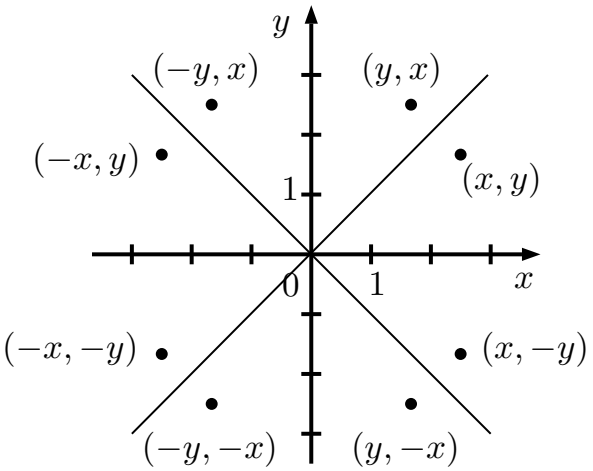
Rysunek 1.13: Zbliżenie na wykres odcinka $(0, 1) - (400, 201)$ leżącego na prostej $y = 0.5x + 1$ – widać regularny, zgodny z oczekiwaniem wzór: *dwa w prawo, jeden do góry*



Rysunek 1.14: Oktanty układu współrzędnych. Oktanty I, IV, V, VIII to obszary w których moduł nachylenia odcinka do osi OX jest z przedziału 0-45 stopni



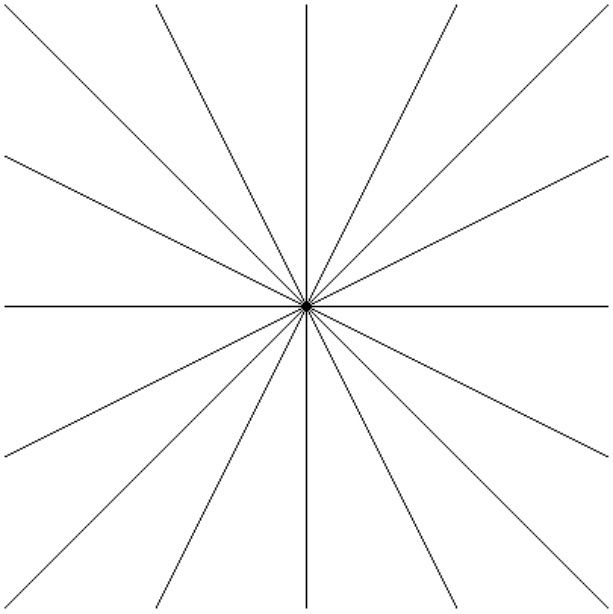
Rysunek 1.15: Przykłady błędnego działania algorytmu w wersji podstawowej (z listingu 1.9): wykres odcinka $(0, 1) - (200, 301)$. Jak widać otrzymany odcinek nachylony jest pod kątek 45 stopni a więc nie może przechodzić przez oba zadane punkty



Rysunek 1.16: Zależności pomiędzy współrzędnymi punktów rozmieszczonych w różnych oktantach

W przypadku oktantach II, III, VI, VII należy dokonać wymiany x z y. Kod z listingu 1.10 zawiera wszystkie opisywane zmiany natomiast rysunek 1.17 przdstawia efekty jego działania dla kilku odcinków

```
void drawLine (int x1, int y1, int x2, int y2){
    int d;
    int deltaA, deltaB;
    int x = x1;
    int y = y1;

    int dX = x2 - x1;
    int dY = y2 - y1;

    // W przyrostach uwzgledniam znaki dX i dY
    int dx = Math.signum(dX);
```



Rysunek 1.17: Przykłady działania algorytmu przedstawionego na listingu 1.10

```
int dy = Math.signum(dY);

// Przeniesienie do wlasciwej cwiartki
dX = Math.abs(dX);
dY = Math.abs(dY);

if (dx >= dy){ // Odcinek lezy w oktancie: I, IV, V lub VIII
    d = dX - 2*dY;
    deltaA = -2*dY;
    deltaB = 2*dX - 2*dY;

    // Nie mozna uzyc petli while jak poprzednio,
    // gdyz zmienna x (y) moze zarowno rosnac jak i malec
    // zaleznie od przypadku
    for(int i=0; i<dX; ++i){
        setPixel(x, y);
        if (d < 0){
            d += deltaB;
            y += dy;
        } else {
            d += deltaA;
        }
        x += dx;
```

```
        }
    } else {
        // We wzorach ponizej dokonano zamiany wspolrzednych
        d = dY -2*dX;
        deltaA = -2*dX;
        deltaB = 2*dY - 2*dX;

        for(int i=0; i<dY; ++i){
            setPixel(x, y);
            if (d < 0){
                d += deltaB;
                x += dx;
            } else {
                d += deltaA;
            }
            y += dy;
        }
    }
}
```

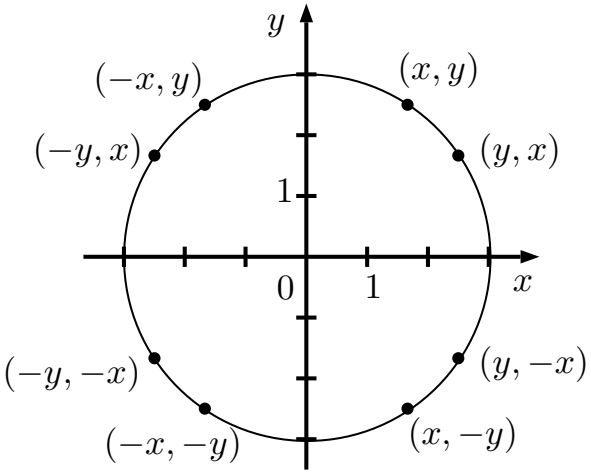
Listing 1.10: src/sra/09.java

1.6 Rasteryzajca okręgu

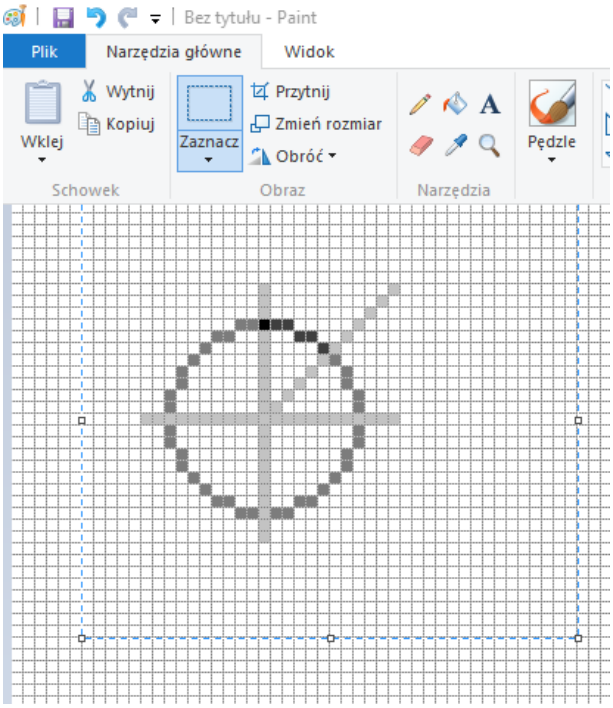
Choć okręg wydaje się być zadaniem trudniejszym, to mając w pamięci dotychczasowe rozważania okaże się, że wcale nie jest to takie trudne zadanie. Podobnie jak w przypadku odcinka przyjmiemy kilka założeń, które znacząco ułatwią rozważania.

- Środek okręgu znajduje się w punkcie $(0, 0)$ a promeń wynosi r.
- Rasteryzaji poddajemy fragment okręgu z oktantu II – fragment w którym łuk okręgu oparty jest na kącie 45 stopni i mieści się w pełni w jednym oktancie.
- Pozostałe fragmenty w łatwy sposób uzyskamy dokonując odbić symetrycznych (porównaj rysunek 1.18).

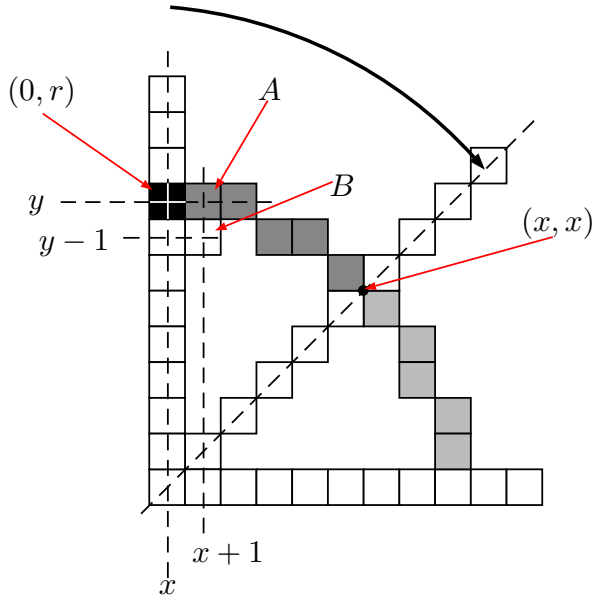
Proces rasetryzacji rozpoczniemy od punktu $(0, r)$ po czym zwiększać będziemy współrzędną x co spowoduje zmiany zmiennej y od r do x (porównaj rysunek 1.20).



Rysunek 1.18: Oktanty i odbicia symetryczne punktu z łuku



Rysunek 1.19: Okrąg narysowany w programie do grafiki rastrowej. Wyróżniono fragment w oktancie II. Powiększenie tego oktantu przedstawiono na rysunku 1.20



Rysunek 1.20: Łuk okręgu w rozważanym oktancie (porównaj także rysunek 1.19)

Posługując się równaniem uwikłanym okręgu

$$f(x,y) = x^2 + y^2 - r^2$$

obliczamy, podobnie jak poprzednio, wartość początkową d (to znaczy wartość d w punkcie $(0,r)$) i przyrosty $delta_A$ i $delta_B$ (przy czym musimy pamiętać, że y maleje a nie rośnie jak w przypadku odcinka)

$$d = f(0+1, r-0.5) = 1 + r^2 - r + 0.25 - r^2 = -r + 1.25$$

$$\begin{aligned} delta_A &= f(x+2, y-0.5) - f(x+1, y-0.5) \\ &= x^2 + 4x + 4 + y^2 - y + 0.25 - r^2 - x^2 - 2x - 1 - y^2 + y - 0.25 + r^2 = 2x + 3 \\ delta_B &= f(x+2, y-1.5) - f(x+1, y-0.5) \\ &= x^2 + 4x + 4 + y^2 - 3y + 2.25 - r^2 - x^2 - 2x - 1 - y^2 + y - 0.25 + r^2 = 2x - 2y + 5 \end{aligned}$$

Jak widać otrzymane różnice rzędu pierwszego zależą od chwilowych wartości współrzędnych x i y. Policzymy zatem różnice rzędu drugiego. Wartości początkowe w punkcie $(0,r)$ wynoszą

$$delta_A(0,r) = 3,$$

$$delta_B(0,r) = -2r + 5.$$

Rysunek 1.21: Przykłady działania algorytmu przedstawionego na listingu 1.11

Ponieważ przy wyborze punktu A wzrasta tylko współrzędna x, więc

$$delta_{AA} = delta_A(x+1, y) - delta_A(x, y) = 2,$$
$$delta_{BA} = delta_B(x+1, y) - delta_A(x, y) = 2.$$

natomiast po wybraniu B maleje dodatkowo także i y

$$delta_{AB} = delta_A(x+1, y-1) - delta_B(x, y) = 2$$
$$delta_{BB} = delta_B(x+1, y-1) - delta_B(x, y) = 4$$

Jak widać przyrosty drugiego rzędu nie zależą już od x ani od y. Ponieważ we wzorze na obliczanie wartości d występuje wartość 1.25 więc możemy „pozbyć się" jej, podobnie jak postąpilismy w przypadku odcinka, posługując się równaniem uwikładnym okręgu postaci

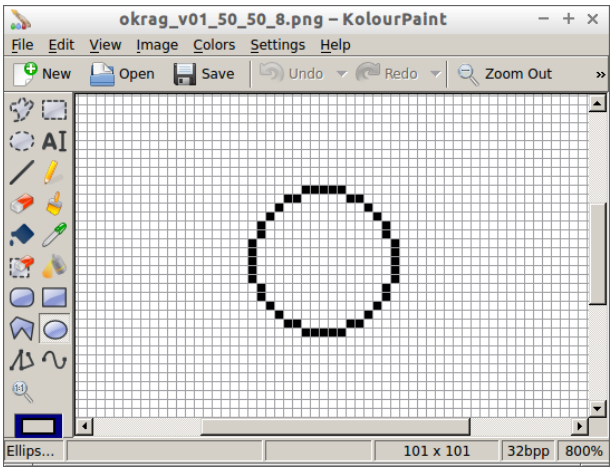
$$f(x, y) = 4x^2 + 4y^2 - 4r^2$$

lub, co na jedno wychodzi,

$$d = f(0+1, r-0.5) = 4 + 4r^2 - 4r + 1 - 4r^2 = -4r + 5$$

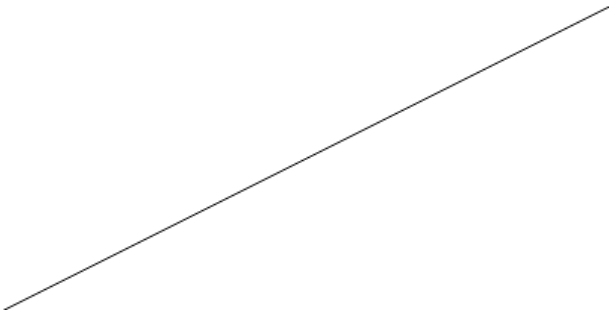
mnożąc obliczone już wartości d i wszystkie przyrosty przez 4. Na listingu 1.11 przedstawiono otrzymany algorytm a efekt jego działania przedstawia rysunek 1.21 oraz powiększenie na rysunku 1.23 (porównaj to powiększenie z rysunkiem 1.19)

```
void drawSymetricPoints(int x0, int y0, int x, int y){
        setPixel(x0-x, y0-y);
        setPixel(x0-x, y0+y);
        setPixel(x0+x, y0-y);
        setPixel(x0+x, y0+y);
        setPixel(x0-y, y0-x);
        setPixel(x0-y, y0+x);
        setPixel(x0+y, y0-x);
```



Rysunek 1.22: Powiększenie rysunku 1.21 będącego efektem działania algorytmu przedstawionego na listingu 1.11 (porównaj to powiększenie z rysunkiem 1.19)

```
        setPixel(x0+y, y0+x);
}

void drawCircle(x0, y0, r){
        int d = 5.0-4*r;
        int x = 0;
        int y = r;

        int deltaA = (-2*r+5)*4;
        int deltaB = 3*4;

        while (x <= y){
                drawSymetricPoints(x0, y0, x, y);
                if (d > 0){
                        d += deltaA;
                        deltaA += 4*4;
                        y -= 1;
                } else {
                        d += deltaB;
                        deltaA += 2*4;
                }
                deltaB += 2*4;
                x += 1;
        }
}
```



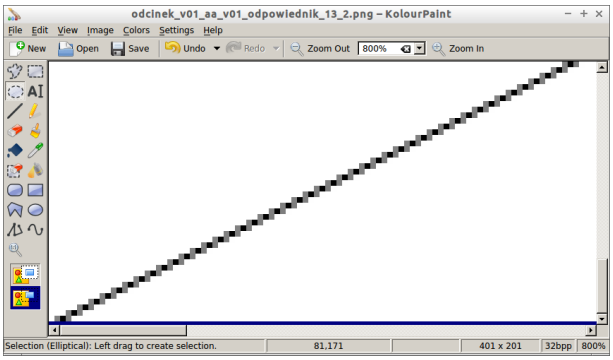
Rysunek 1.23: Działanie algorytmu wygładzającego krawędzie na przykładzie: wykresu odcinka $(0,1)-(400,201)$ leżącego na prostej $y = 0.5x + 1$ (porównaj także rysunek 1.12)

Listing 1.11: src/sra/10.java

1.7 Antyaliasing

```
void drawLine (int x1, int y1, int x2, int y2){
        float a = (256*(y2 - y1)) / (x2 - x1);
        int x = x1;
        float y = y1;
        int color, i;

        while(x <= x2){
                color =  y & 255;
                i = y >> 8;

                setPixel(x, y+i, color);
                setPixel(x, y+i+1, 255-color);

                x = x + 1;
                y = y + a;
        }
}
```

Listing 1.12: src/sra/11.java: Algorytm rysowania odcinka z wygładzaniem krawędzi. Zasadniczo jest to ten sam kod co na listingu 1.5 z tą różnicą, że otrzymywane wartości rzeczywiste wykorzystane zostają do podjęcia decyzji o stopniu „rozmycia” danego piksela



Rysunek 1.24: Zbliżenie na wykres odcinka $(0,1)-(400,201)$ otrzymanego za pomocą algorytmu wykorzystującego wygładzanie krawędzi (porównaj także rysunek 1.13)

1.8 Wypełnianie wielokątów

Założenia

1. Wierzchołek opisywany jest przez jego współrzędne (x, y).
2. Figura wyznaczona jest przez n wierzchołków numerowanych od 1 do n.
3. Przyjmujemy, że krawędź określa par kolejnych wierzchołków.

1. Utwórz tablicę krawędzi. W tablicy musi znaleźć się także krawędź „zamykająca", tj. $(x_n, y_n) - (x_1, y_1)$);
2. Usuń z tablicy krawędzie poziome.
3. Jeśli współrzędna y końca krawędzi jest mniejsza niż współrzędna y początku, to zamień końce.
4. Posortuj tablicę w kolejności rosnących współrzędnych y początków krawędzi.
5. Utwórz początkowo pustą tablicę krawędzi aktywnych, czyli przeciętych kolejną linią poziomą.
6. Zainicjalizuj zmienną y wartością współrzędnej y pierwszej krawędzi w tablicy.
7. Tak długo jak tablicę krawędzi aktywnych nie jest pusta powtarzaj

   a) x

- **Krok 1:** Realizacja punktu **??**

```
tablicaKrawedzi = [{(2,2)-(5,8)},
 {(5,8)-(11,8)},
 {(11,8)-(12,3)},
 {(12,3)-(9,4)},
 {(9,4)-(6,4)},
 {(6,4)-(2,2)}
                 ]
```

- **Krok 2:** Realizacja punktu ??

```
tablicaKrawedzi = [{(2,2)-(5,8)},
 usunieta
 {(11,8)-(12,3)},
 {(12,3)-(9,4)},
 usunieta
 {(6,4)-(2,2)}
                 ]
```

- **Krok 3:** Realizacja punktu ??

```
tablicaKrawedzi = [{(2,2)-(5,8)},

 {(12,3)-(11,8)-}, <- zamiana
 {(12,3)-(9,4)},

 {(2,2)-(6,4)-} <- zamiana
                 ]
```

- **Krok 4:** Realizacja punktu ??

```
tablicaKrawedzi = [{(2,2)-(5,8)},
 {(2,2)-(6,4)},
 {(12,3)-(11,8)},
 {(12,3)-(9,4)}
                 ]
```

- **Krok 5:** Realizacja punktu **??**

```
tablicaKrawedziAktywnych = []
```

- **Krok 6:** Realizacja punktu **??**

```
y := 2
```

Bibliografia

[1] xx, *xx*, http://xxx, retrived 2013-01-17.

Spis rysunków

1.1 Rysunek sytuacyjny, stanowiący punkt odniesienia do rozważań w tym rozdziale. Zauważmy, że piksel ma niezerową wysokość i szerokość a więc najwygodniej współrzędne piksela utożsamiać z jego środkiem. W konsekwencji piksel $p = (x, y)$ ma wymiary $[x - w/2, x + w/2] \times [y - h/2, y + h/2]$, gdzie w i h to odpowiednio szerokość i wysokość piksela. Dla prostoty możemy przyjąć, że oba wspomniane wymiary mają wartość 1 2

1.2 Idealny obraz odcinka po rasteryzacji 6

1.3 Rzeczywisty obraz odcinka po rasteryzacji 7

1.4 Rzeczywisty obraz odcinka $(0, 100) - (400, 101)$ po rasteryzacji będący wynikiem akumulacji błędów zmiennoprzecinkowych 7

1.5 Rzeczywisty obraz odcinka $(0, 100) - (400, 101)$ po rasteryzacji z uwzględnieniem akumulacji błędów (algorytm 1.6). Porównujac ten rysunek z rysunkiem 1.4 widzimy w jego środkowej części „schodek” co wydaje się być zgodne z naszą intuicją 9

1.6 Przypadek gdy $|a| > 1$ – powstawanie „dziur” 9

1.7 Przypadek gdy $|a| > 1$ – więcej niż jeden piksel w kolumnie 10

1.8 Wykres funkcji $g(x)$ dla wartości z tabeli 1.1 11

1.9 Wykres znaku wartości funkcji $f(x, y)$ dla wartości z tabeli 1.3; wartości ujemne oznaczono symbolem ’−’, dodatnie symbolem ’+’ a czarną kropką wartości równe zero 12

1.10 Wybór kolejnego piksela 13

1.11 Wartości w punktach środkowych dla piksela P, A oraz B 15

1.12 Przykład poprawnego działania algorytmu z listingu 1.9: wykres odcinka $(0, 1) - (400, 201)$ leżącego na prostej $y = 0.5x + 1$ 20

1.13 Zbliżenie na wykres odcinka $(0, 1) - (400, 201)$ leżącego na prostej $y = 0.5x + 1$ – widać regularny, zgodny z oczekiwaniem wzór: *dwa w prawo, jeden do góry* 20

1.14 Oktanty układu współrzędnych. Oktanty I, IV, V, VIII to obszary w których moduł nachylenia odcinka do osi OX jest z przedziału 0-45 stopni 20
1.15 Przykłady błędnego działania algorytmu w wersji podstawowej (z listingu 1.9): wykres odcinka $(0, 1) - (200, 301)$. Jak widać otrzymany odcinek nachylony jest pod kątek 45 stopni a więc nie może przechodzić przez oba zadane punkty 21
1.16 Zależności pomiędzy współrzędnymi punktów rozmieszczonych w różnych oktantach 21
1.17 Przykłady działania algorytmu przedstawionego na listingu 1.10 22
1.18 Oktanty i odbicia symetryczne punktu z łuku 24
1.19 Okrąg narysowany w programie do grafiki rastrowej. Wyróżniono fragment w oktancie II. Powiększenie tego oktantu przedstawiono na rysunku 1.20 24
1.20 Łuk okręgu w rozważanym oktancie (porównaj także rysunek 1.19) 25
1.21 Przykłady działania algorytmu przedstawionego na listingu 1.11 26
1.22 Powiększenie rysunku 1.21 będącego efektem działania algorytmu przedstawionego na listingu 1.11 (porównaj to powiększenie z rysunkiem 1.19) 27
1.23 Działanie algorytmu wygładzającego krawędzie na przykładzie: wykresu odcinka $(0, 1)-(400, 201)$ leżącego na prostej $y = 0.5x + 1$ (porównaj także rysunek 1.12) 28
1.24 Zbliżenie na wykres odcinka $(0, 1) - (400, 201)$ otrzymanego za pomocą algorytmu wykorzystującego wygładzanie krawędzi (porównaj także rysunek 1.13) 29

Spis tabel

1.1 Wartości funkcji $g(x) = ax + b$ dla $a = 0.5$, $b = 1$ w wybranych punktach 10
1.2 Wartości funkcji $f(x, y)$ w punktach z tabeli 1.1 11
1.3 Wartości funkcji $f(x, y)$ w wybranych punktach 11
1.4 Wartości funkcji f^* 18

Skorowidz

accumulator, 40
assember, 25
assembling, 25
assembly, 25
    language, 25

execution
    out-of-order, 37
    speculative, 37, 39

hazard, 42

instruction
    pipeline, 41
    pointer, 40

labels, 16
language
    assembly, 25
little endian, 49
long mode, 36

memory
    protected, 35
    virtual, 35
memory management unit, 50
memory protection, 50
memory segmentation, 49
mode
    long, 36
    protected, 35
    real, 35
    virtual, 36

page table, 50
paging, 35
processor status word, 40
program counter, 40
protected mode, 35

real mode, 35
register, 37
    accumulator, 40, 42
    address, 40
    control and status, 41
    data, 40
    destination index, 43
    floating point, 40
    general purpose, 40
    instruction, 40
    instruction pointer, 40
    processor status word, 40
    program counter, 40, 41
    renaming, 37, 38
    source index, 43
    special purpose, 40
    stack pointer, 40, 43
        base, 43
    status, 40
    user-accessible, 40
    vector, 41
register base, 42
register counter, 42

register data, 42

segmentation fault, 50
stack pointer, 40

virtual mode, 36