xxx

Lecture Notes in Computer Graphics

2D graphics

Piotr Fulmański

Łódź, 2014

Spis treści

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 zwykle nie ma ich pod ręką gdy są potrzebne, więc zwykle 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

Simple raster algorithms

1.1 Pierwsze podejście do rysowania odcinka

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

Rysunek 1.1: Pierwszy rysunek sytuacyjny, stanowiący punkt odniesienia do rozważań w tym rozdziale

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
    // a y?!
    y = y1 + ?
}
```

Listing 1.1: src/sra/01.java

Problemem jest właściwe dobranie wartości zmiany dla zmiennej y. Posługując się przedstawionym rysunkiem można użyć następującej formuły

$$dy(x - x_1)/dx$$

gdzie

- dy – pojedyńcza zmiana wysokości przy prześciu od kolumny k do $k + 1$;

- $x - x_1$ – numer kolumny (nk) względem punktu odniesienia jakim jest punkt p_1;
- dx – szerokość kolumny.

Formuła ta określa wielkość zmiany współrzędnej y dla zadanej współrzednej x. Jeśli numerem kolumny jest (nk) to mamy wówczas nk zmian przechodząc od kolumny $x1$ od kolumny x.

O ile wiadomo, że przyrost (zmiana) x oznaczana jako dx wynosi 1 – przechodzimy przez kolejne kolumny – to przysrost dy wymaga już trochę arytmetyki. 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 x tych punktów), a więc istotnie wszystkie zmiany na osi OY są takie same. Stąd zaś wynika, że

$$dy = a \cdot dx.$$

Teraz możemy w końcu uzupełnić nasz kod

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

Listing 1.2: src/sra/02.java

Zauważmy, że będzie on równoważny kodowi

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

Listing 1.3: src/sra/03.java

Rysunek 1.2: Idealny obraz odcinka po rasteryzacji

Rysunek 1.3: Rzeczywisty obraz odcinka po rasteryzacji

(pozbywamy się mnożenia przez dx a następnie dzielenia przez ten sam składnik). Lepszym rozwiązaniem będzie kod poniższy

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

Listing 1.4: src/sra/04.java

Jak do tej pory wszystko wydaje się być dobrze z jednym małym wyjątkiem: zmienna y przyjmuje wartości rzeczywiste (porównaj rysunke 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). Otrzymamy wówczas

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

    while(x <= x2){
        setPixel(x, round(y));
        x += 1; // wartosc dokladna - arytmetyka calkowitoliczbowa
        y += a; // wartosc przyblizona - arytmetyka zmiennoprzecinkowa
    }
}
```

Listing 1.5: src/sra/05.java

Efekt działanie tego algorytmu przedstawia rysunek 1.3.

If you want to know more... 1.1 (Aumulacja błędów zaokrągleń)**.** *tutu do dopisania*

Rysunek 1.4: Efekt działanie algorytmu dla przypadku gdy $|a| \leqslant 1$

Rysunek 1.5: Efekt działania algorytmu dla przypadku gdy $|a| > 1$

Rysunek 1.6: Wykres funkcji $f(x)$ dla wartości z tabeli 1.1

Dodatkowo o ile dla $|a| \leqslant 1$ działanie algorytmu będzie prawidłowe – patrz rysunek 1.4 – to dla $|a| > 1$ już nie – patrz rysunek 1.5. 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 $y = f(x)$ 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

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

ustalmy wartosci parametrów a i b i policzmy wartość $f(x)$ dla kilku wybranych wartości x (porównaj rysunek 1.6)

1	2	3
4	5	6
7	8	9

Tabela 1.1: A simple table

Rozważmy teraz funkcję

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

i dla otrzymanych w tabeli 1.1 wartości x i y obliczmy wartość funkcji $f(x, y)$ (porównaj tabela 1.2 i rysunek 1.7). Pozostawiając wartości x jak poprzednio zmieńmy wartości y (porównaj tabela 1.3 i rysunek 1.8). Jak widać, jedynie dla punktów leżących dokładnie na prostej otrzymujemy

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

1	2	3
4	5	6
7	8	9

Tabela 1.2: A simple table

Rysunek 1.7: Wykres funkcji $f(x, y)$ dla wartości z tabeli 1.2

x	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 1.3: A simple table

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.

Po tej obserwacji możemy przejść do kolejnej wersji algorytmu.

1.2.2 Algorytm rysowania dowolnej krzywej

Załóżmy, że krzywa w przedziale $[x_1, x_2]$ spełnia podane wcześniej założenia. Algorytm zaczyna działanie od punktu (x_1, y_1) i „stawia" piksel o tych współrzędnych (zaczerniony okrąg na rysunku 1.9). Następnie współrzędna x jest zwiększana o 1, czyli kolejny piksel ma współrzędna

$$x = x_1 + 1.$$

Pozostaje teraz najwżniejsza część: należy wybrać wartość współrzędnej y. Dzięki przyjętym warunkom wybór ten ogranicza się do pikseli A i B i determinuje go punkt przecięcia krzywej z linią pionową

Rysunek 1.8: Wykres znaku wartości funkcji $f(x, y)$ dla wartości z tabeli 1.3

Rysunek 1.9: Wybór kolejnego piksela

$x_1 + 1$. Jeśli punkt ten jest bliżej punktu A to oczywiście wybierany jest punkt A, a w przeciwnym razie punkt B.

Liczenie odległości może być dobrym rozwiązaniem podczas implementacji antyaliasingu. Jeśli jednak chcemy narysować okrąg „dokładnie" lub zależy nam na szybkości działania to musimy poszukać innych metod – metod, które działanie swoje opierają na liczbach całkowitych. Na rysunku 1.9 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 punkt B.

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

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

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

```
1  // Nie jest potrzebna wartosc y2
2  void drawLine (int x1, int y1, int x2){
3      float d;
4      int x = x1;
5      int y = y1;
6
7      while(x <= x2){
8          setPixel(x, y);
9          d = f(x+1, y+0.5)
10         if (d < 0){
11             y += 1;
12         }
13         x += 1;
14 }
```

Listing 1.6: 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.

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 1.6 wymaga wielokrotnego obliczania wartości $f(x, y)$ w linii 8. Ponieważ jednak współrzędne x i y rysowanych punktów zmieniają się o znane wartości, więc wartość funkcji w kolejnym kroku można przewidzieć. Wartość funkcji f w punkcie środkowym wzgledem punktu (x_1, y_1) jak wiemy wynosi

$$d = f(x_1 + 1, y_1 + 0.5)$$

w punkcie A

$$d_A = f(x_1 + 2, y_1 + 0.5)$$

natomiast w punkcie B

$$d_B = f(x_1 + 2, y_1 + 1.5)$$

Względem dowolnego punktu (x, y) leżącego na krzywej wartości te wynoszą odpowiednio

$$d = f(x + 1, y + 0.5),$$
$$d_A = f(x + 2, y + 0.5),$$
$$d_B = f(x + 2, y + 1.5).$$

Tak więc przy wyborze punktu A wartość funkcji f 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 1.7.

```
1 // Nie jest potrzebna wartosc y2
2 void drawLine (int x1, int y1, int x2){
3     float d = f(x1+1, y1+0.5)
```

```
4       int x = x1;
5       int y = y1;
6
7       while(x <= x2){
8           setPixel(x, y);
9           if (d < 0){
10              d += f(x+2, y+1.5)-f(x+1, y+0.5)
11              y += 1;
12          } else{
13              d += f(x+2, y+0.5)-f(x+1, y+0.5)
14          }
15          x += 1;
16      }
17  }
```

Listing 1.7: src/sra/07.java

Zmienna d nazywana jest zmienną decyzyjną lub błędem aproksymacji.

Zaproponowana modyfikacja wydaje się wcale niczedo nie usprawniać: zamiast jednokrotnego obliczania wartości funkcji f w każdej iteracji teraz obliczamy dwie wartości (linia 9 lub 12). Jak napisałem jednak, uprzedzając bieg wydarzeń na początku tego podrozdziału, niebawem okaże się, że wprowadzona zmiana jest bardzo korzystna.

1.4 Konsekwencje usprawnienia

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

1.4.1 Po pierwsze...

...wartość zmiennej d jest równa

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

Wartości różnic $delta_A$ oraz $delta_B$.

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

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

Rysunek 1.10: Niezależność pewnych wartości od wyboru położenia odcinka

Ponieważ

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

więc wartości różnic $delta_A$ oraz $delta_B$ zależą wyłącznie od współrzędnych końców odcinka. Takie różnice, tj. różnice, które są stałe, nazywać będziemy różncami 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 zależą (nadal) 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 – porównaj rysunek 1.10. 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

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

używać będziemy

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

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.

x	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^*

- 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

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

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

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

gdyż dokładnie dla takich samych par (x, y) przyjmuje wartość równą 0 (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 $delta_B$

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

oraz początkową zmiennej d

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

Rysunek 1.11: Oktanty układu współrzędnych

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

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

Finalną wersję algorytmu przedstawia listing 1.8.

```
void drawLine (int x1, int y1, int x2, int y2){

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

    int d = 2*dY - dX;
    int deltaA = 2*dY;
    int deltaB = 2*dY - 2*dX;
    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 1.8: src/sra/08.java

1.5 Uogólnienie na inne oktanty okładu współrzędnych

Podany algorytm będzie działał poprawnie o ile odcinek mieści się w pierwszym oktancie układu współrzędnych – patrz rysunek 1.11. W przeciwnym razie uzyskane efekty mogą odbiegać od oczekiwanych, co ilustruje rysunek 1.12. Narysowanie odcinka w innych oktantach wymaga odpowiedniej

Rysunek 1.12: Przykłady błędnego działania algorytmu w wersji podstawowej

zamiany współrzędnych. W obszarach opisanych kolorem niebieskim 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.

W przypadku obszarów opisanych kolorem czerwonym należy dokonać wymiany x z y. Kod z listingu 1.9 zawiera wszystki opisywane zmiany.

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

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

    if (dx >= dy){ // Odcinek lezy w niebieskim oktancie
        d = 2*dY - dX;
        deltaA = 2*dY
        deltaB = 2*dY - 2*dX

        // Nie mozna zrobic 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 { // Odcinek lezy w czerwonym oktancie
        // We wzorach ponizej dokonano zamiany wspolrzednych
        d = 2*dX - dY;
        deltaA = 2*dX
        deltaB = 2*dX - 2*dY

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

Listing 1.9: 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.

Rysunek 1.13: Oktanty i odbicia symetryczne punktu z łuku

- Środek okręgu znajduje się w punkcie $(0, 0)$ a promeń wynosi r.
- Rasteryzaji poddajemy zaznaczony na zielono fragment okręgu – 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 1.13).

Proces rasetryzacji rozpoczniemy od punktu $(0, r)$ po czym zwiększać będziemy współrzędną x co spowoduje zmiany zmiennej y od r do x (porównaj rysunek 1.13).

Posługując się równaniem uwikłanym okręgu

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

obliczamy, podobnie jak poprzednio, wartość początkową d 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

$$\begin{aligned} delta_A(0, r) &= 3, \\ delta_B(0, r) &= -2r + 5. \end{aligned}$$

Ponieważ przy wyborze punktu A wzrasta tylko współrzędna x, więc

$$\begin{aligned} delta_{AA} &= delta_A(x + 1, y) - delta_A(x, y) = 2, \\ delta_{BA} &= delta_B(x + 1, y) - delta_A(x, y) = 2. \end{aligned}$$

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 1.10 przedstawiono otrzymany olgorytm.

```
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);
    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(x,y);
```

```
        if (d > 0){
            d += deltaA;
            deltaA += 4*4;
            y -= 1;
        } else {
            d += deltaB;
            deltaA += 2*4;
        }
        deltaB += 2*4;
        x += 1;
    }
}
```

Listing 1.10: src/sra/10.java

Bibliografia

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

Spis rysunków

Spis tabel

Skorowidz