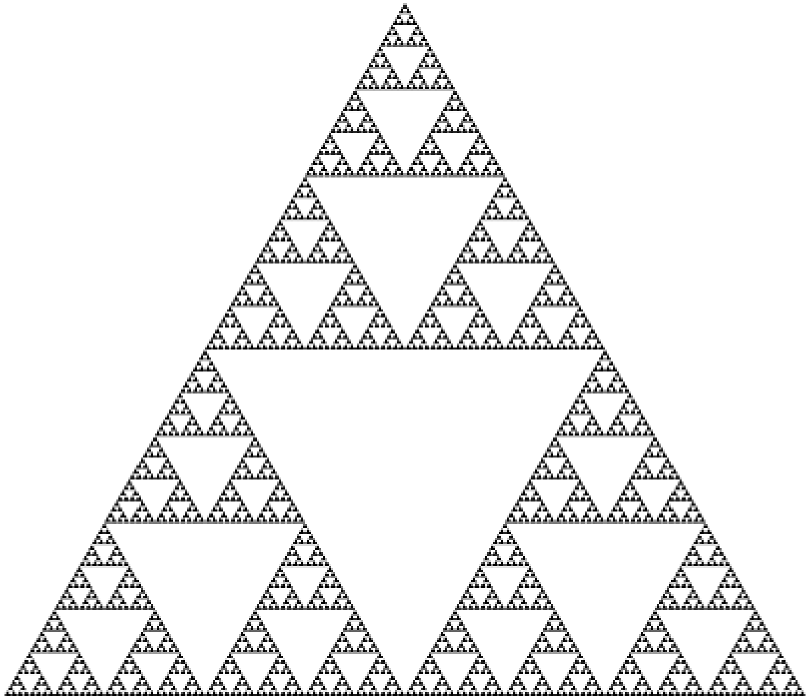

Figure 1: Rys.1. Trójkąt Sierpińskiego.

Na dzisiejszych zajęciach będziemy zajmować się fraktalami. Fraktale to zbiory matematyczne charakteryzujące się tzw. samopodobieństwem – w każdym powiększeniu wyglądają identycznie[1].

Trójkąt Sierpińskiego

Idea trójkąta Sierpińskiego polega na tym, że każdy trójkąt zawiera w sobie trzy takie same trójkąty o dwukrotnie krótszych bokach (patrz Rys. 1).

Trójkąt Sierpińskiego można stworzyć na różne sposoby. My wykorzystamy **rekurencję**.

[1] Z tego powodu wydają się bardzo skomplikowane. Okazuje się jednak, że fraktale stanowią całkiem niezły sposób opisu przyrody – wystarczy spojrzeć na kawałek wybrzeża Wielkiej Brytanii albo na gałąź choinki, żeby stwierdzić, że przypominają całość, tylko w pomniejszeniu.

Rekurencyjne wywoływanie funkcji

Rekurencyjne wywoływanie funkcji polega na tym, że dana funkcja wywołuje samą siebie. Jak łatwo się domyślić, takie wywoływanie trwałoby w nieskończoność. Aby tego uniknąć, nakłada się warunek na wywołanie funkcji.

Cały mechanizm, ilustruje to poniższy przykład:

```
void recursiveDraw(double x, double y, double R) {
  circle(x, y, R);

  if(x < 500) {
    recursiveDraw(x + 5, y + 5, 0.88 * R);
  }
}
```

Funkcja `recursiveDraw` rysuje okrąg o zadanych parametrach oraz wywołuje samą siebie z innymi wartościami argumentów, dopóki wartość x nie przekroczy 500.

W wypadku trójkąta Sierpińskiego, napiszemy funkcję, która generuje fraktal do pewnej „głębokości”. Głębokość 1 oznacza pojedynczy trójkąt, 2 – trzy trójkąty, 3 – dziewięć, itd. Funkcję taką można napisać zauważając, że trójkąt o głębokości n składa się z trzech trójkątów o głębokości $n - 1$.

Ćwiczenia

Napisz funkcję `sierpinski(x, y, d, n)`, która:

1. Dla $n = 1$ rysuje trójkąt równoboczny o wysokości d oraz lewym dolnym wierzchołku w punkcie (x, y)
2. Dla $n > 1$ wywołuje trzy razy funkcję `sierpinski()` z:
    - odpowiednio zmienionymi współrzędnymi (x, y),
    - dwa razy zmniejszonym d,
    - głębokością $n - 1$.
3. Na początku funkcji umieść instrukcję `animate()` w celu spowolnienia rysowania i zobacz w jakiej kolejności powstają kolejne trójkąty.

Zbiór Julii, zbiór Mandelbrota

Wyobraźmy sobie ciąg liczb zespolonych takich, że każdy wyraz zależy od poprzedniego zgodnie ze wzorem: $z_{n+1} = z_n^2 + c$. Taki ciąg ma nietypowe własności. Między innymi, jego rozbieżność zależy w bardzo złożony sposób od wyrazu początkowego z_0 i stałej c. Zbiór takich z_0, dla których ciąg ten nie jest rozbieżny nazwany został zbiorem *Julii* (zależy on od c). Zaś zbiór takich c, dla których ciąg ten nie jest rozbieżny oraz $z_0 = 0$ – zbiorem *Mandelbrota*.

Mapa kolorów

Do zobrazowania tych zbiorów potrzebna jest umiejętność kolorowania pojedynczych pikseli. Wyobraźmy sobie, że nasze okno grafiki to układ dwuwymiarowy, gdzie $x \in [-4, 4]$ i $y \in [-3, 3]$. Używając funkcji `setcolor()`, możemy narysować wykres funkcji $\sin(x^2 + y^2)$ dla tych współrzędnych.

Funkcja `setcolor()` jako argument przyjmuje liczbę z przedziału od 0 do 1 i ustawia kolor rysowania.

Przeanalizuj poniższy program:

```
double fun(double x, double y) {
  return (sin((x * x + y * y) * 50.0) + 1.0) / 2.0;
}

void main() {
  double r;
  int i, j;

  graphics(800, 600);

  for(i = 0; i < 800; i++)
    for(j = 0; j < 600; j++) {
      r = fun(i / 200.0 - 2.0, j / 200.0 - 1.5);
      setcolor(r);
      point(i, j);
    }

  wait();
}
```


Rozbieżność

Napisz funkcję `divergence(zR, zI, cR, cI)`, która sprawdza czy ciąg opisany równaniem

$$\begin{cases} z_0 = zR + zI \cdot i \\ z_{n+1} = z_n^2 + c \\ c = cR + cI \cdot i \end{cases}$$

jest zbieżny.

Przyjmij, że maksymalna liczba iteracji to 600. Jeśli ciąg jest rozbieżny, niech funkcja zwraca liczbę iteracji, po której $|z_n| > 2$ podzieloną przez 600. Jeżeli ciąg jest zbieżny (tzn. po 600 iteracjach nadal $|z_n| < 2$), niech funkcja zwraca 0. **Przypomnienie:** Działania na liczbach zespolonych wykonuj jak zwykłe mnożenie dwumianów, pamiętając jedynie, że $i^2 = -1$.

Ćwiczenia

- Pokoloruj piksele, tak jak w przykładzie z funkcją sin(), wywołując `divergence(x, y, -0.3, 0.63)`. Dla ułatwienia, wywołanie to możesz umieścić wewnątrz `fun(x, y)` albo bezpośrednio wpisać jego wynik do zmiennej r.
- Zobacz, jak wygląda zbiór dla innych c, np $c = -0.1 + 0.65 \cdot i$, $c = 0$, $c = 1$, zmieniając liczbę iteracji jeśli zajdzie potrzeba.
- Pokoloruj piksele zgodnie z `divergence(0, 0, x, y)`. Powinieneś otrzymać zbiór *Mandelbrota*.
- Zmodyfikuj funkcję tak, aby zwracała logarytm z liczby iteracji.
- Zmodyfikuj kod tak, aby zobaczyć obszar o środku w punkcie $(-0.345, 0.635)$.
- Powiększ obszar 200 razy.

* Antyaliasing - dla dociekliwych

Antyaliasing ma na celu usunięcie efektów reprezentacji ze skończoną rozdzielczością. Dla przykładu, jeśli mamy literę, to jej krawędź jest gładką krzywą, która przechodzi tylko częściowo przez piksel. Jeśli zaczernimy tylko piksele wewnątrz krzywej, uzyskamy bardzo nienaturalny efekt. Jeśli jednak użyjemy odcienia szarości, proporcjonalnego do „pola"'przykrytego tą literą, uzyskamy ładną, gładką czcionkę. Wyobraźmy sobie, że mamy obraz składający się z dużej liczby gęsto ułożonych czarnych linii na białym tle. Chcąc taki obraz zmniejszyć, możemy wziąć np. co trzeci piksel w każdą stronę. Jednak takie podejście spowoduje, że raz trafimy na

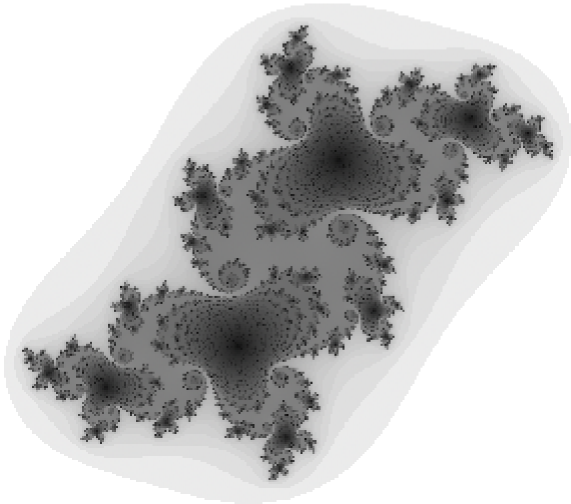

Figure 2: Rys.2. Zbiór Julii.

w pełni czarny, a raz na w pełni biały piksel. Ostatecznie uzyskamy biało-czarną kaszę. Drugim podejściem byłoby wyznaczenie średniej z każdej kostki 3×3. Takie podejście da nam pożądany efekt rozmazania zbyt małych struktur i kolor szary w miejscu losowej kaszy. Taki rodzaj antyaliasingu nazywany jest super-samplingiem. W wypadku obrazów takich jak zbiór *Mandelbrota* czy zbiór *Julii*, możemy dla każdego piksela obliczyć $k \times k$ wartości funkcji `fun()` i uśrednić wynik. Taki zabieg wygładzi obraz i usunie „odstające" piksele.

Ćwiczenia

- Stwórz funkcję `fun2(x, y)`, która oblicza średnią wartość funkcji `fun` w czterech punktach: (x, y), $(x+s, y)$, $(x, y+s)$ i $(x+s, y+s)$, gdzie $s = 1/800$. Pokoloruj piksele za jej pomocą.
- Pokoloruj połowę obrazu z antyaliasingiem, a połowę bez (Np. zbiór Julii dla $c = -0.1 + 0.65 \cdot i$). Czy widać różnicę?
- * Spróbuj uogólnić technikę z $k = 2$ na dowolne k.
- * Zamiast średniej oblicz maksimum lub minimum.
- Zamień funkcję `setcolor()` na `setgray()`.

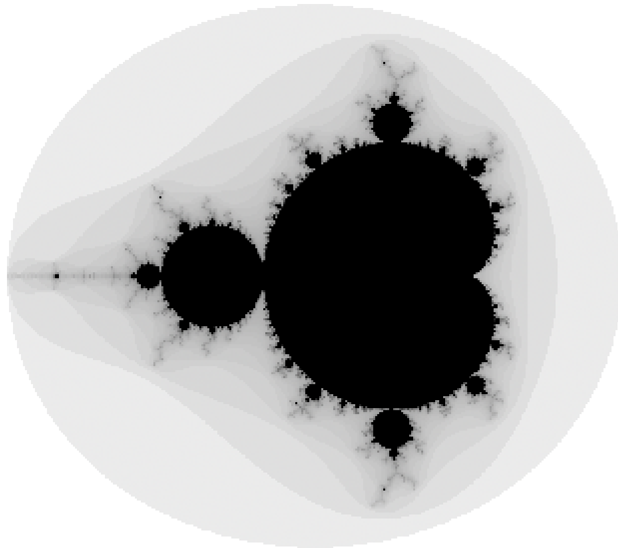

Figure 3: Rys.3. Zbiór Mandelbrota.