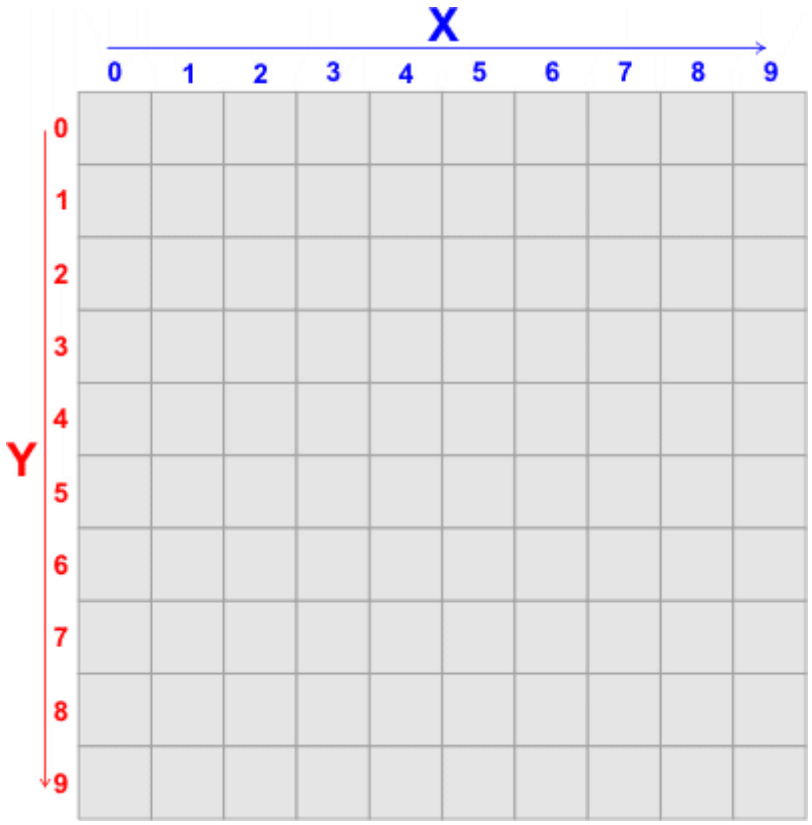
Algorytm Bresenhama - rysowanie odcinka

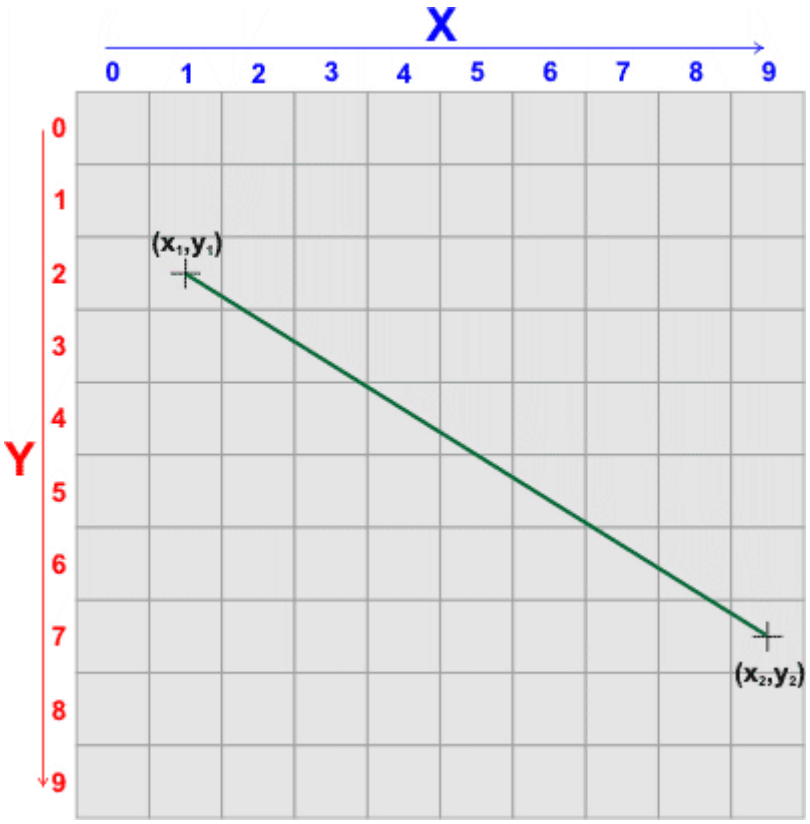
W rozdziale OL035 zaprojektowaliśmy funkcję gfxPlot() rysującą pojedyncze punkty na powierzchni graficznej. Dalej, bazując na tym rozwiązaniu, opracowaliśmy funkcje rysowania linii poziomych gfxHLine(), pionowych gfxVLine i ramek gfxRectangle(). Następnym krokiem będzie zaprojektowanie funkcji gfxLine() do rysowania odcinka linii prostej pomiędzy punktem początkowym x_p,y_p a punktem końcowym x_k,y_k. W tym celu zastosujemy popularny **algorytm Bresenhama**, który stosowany jest w urządzeniach rastrowych do kreślenia odcinków za pomocą pikseli.

Idea algorytmu Bresenhama jest następująca:

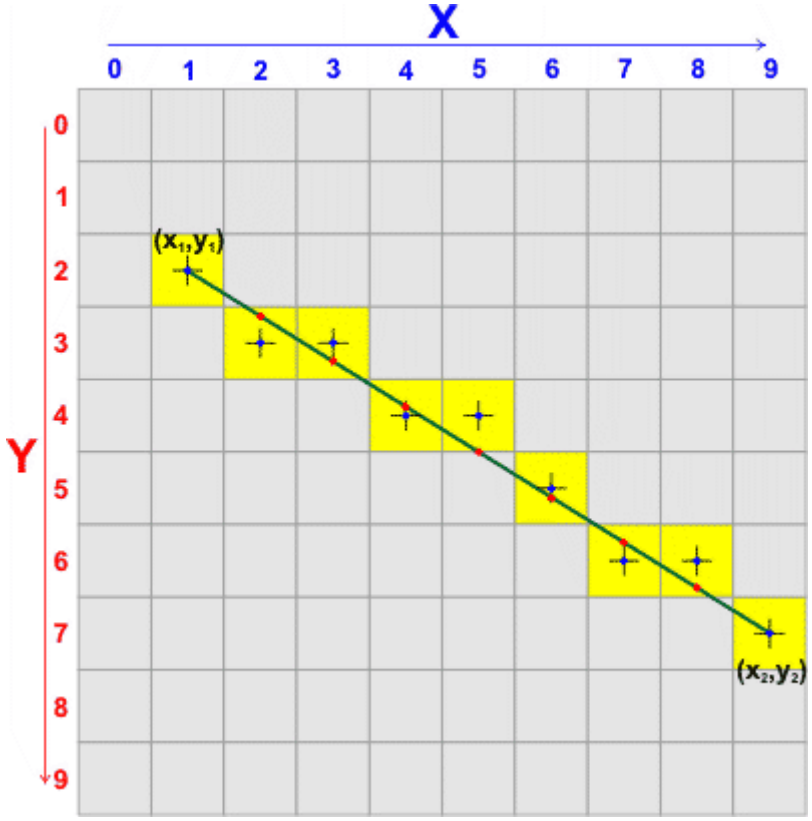
Mamy siatkę punktów. Każdy punkt posiada odpowiednie współrzędne całkowite x,y. Oś y jest skierowana w dół.



Na siatce wybieramy dwa punkty o współrzędnych x_1,y_1 oraz x_2,y_2. Współrzędne odnoszą się do środka piksela. Bez zmniejszania ogólności załóżmy, iż punkty te wybrano tak, aby odcinek (x_1,y_1)-(x_2,y_2) tworzył z osią OX kąt w przedziale od 0 do 45°. Inne przypadki, jak zobaczymy dalej w artykule, da się prosto sprowadzić do tego przypadku przy pomocy prostych transformacji współrzędnych. Wybrane punkty mają być połączone odcinkiem.



Zadanie sprowadza się do pokolorowania kolejnych pikseli, których środki leżą najbliżej punktu na odcinku o tej samej współrzędnej, co piksel. Zwróć uwagę, iż ze względu na nasze założenie, w kierunku X współrzędne kolejnych pikseli są zwiększane zawsze co 1, natomiast w kierunku Y odbywa się to mniej często. Kierunek X nazwiemy szybkim, a kierunek Y wolnym.



Algorytm Bresenhama określa, kiedy należy wykonać ruch w kierunku wolnym Y (bo w kierunku szybkim X ruch wykonujemy zawsze!). Rozważmy równanie prostej przechodzącej przez dwa punkty (x_1,y_1) i (x_2,y_2):

$$dx = (x_2 - x_1)$$
$$dy = (y_2 - y_1)$$
$$y = y_1 + (x - x_1)\frac{dy}{dx}$$

W powyższym równaniu x oznacza współrzędną środka kolejnego piksela (punkty niebieskie). Współrzędna y natomiast odnosi się do punktu leżącego na prostej zawierającej odcinek (punkty czerwone). Środek piksela jest zwykle w pewnej odległości od odpowiadającego mu punktu na odcinku. Oznaczmy tą odległość przez ddy, a współrzędną y środka piksela przez y_p. Zatem możemy zapisać:

$$y = y_p + ddy$$

Wstawmy to wyrażenie do równania prostej:

$$y_p + ddy = y_1 + (x - x_1)\frac{dy}{dx}$$

Teraz przekształćmy wyrażenie następująco:

$$y_p + ddy = y_1 + (x - x_1)\frac{dy}{dx} \Big/ - y_p$$

$$ddy = y_1 - y_p + (x - x_1)\frac{dy}{dx} \Big/ \cdot dx$$

$$ddy \cdot dx = (y_1 - y_p)dx + (x - x_1)dy$$

$$ddy \cdot dx = -(y_p - y_1)dx + (x - x_1)dy \;/ \cdot -1$$

$$-ddy \cdot dx = (y_p - y_1)dx - (x - x_1)dy$$

Wyrażenie -ddy•dx jest tzw. wyrażeniem błędu (ang. error term). Zgodnie ze wzorem krok w kierunku szybkim X powoduje zmniejszenie wyrażenia błędu o dy. Jeśli wartość wyrażenia będzie ujemna, to wykonujemy krok w kierunku wolnym Y, czyli zwiększamy je o dx. Ponieważ dx ≥ dy, to wyrażenie błędu sprowadzi się znów do wartości dodatniej lub 0. Na tej podstawie algorytm Bresenhama określa kiedy należy wykonać ruch w kierunku Y - stara się zawsze utrzymać wyrażenie błędu w zakresie liczb dodatnich i zera. Wartości ddy wcale nie musimy obliczać. Bardzo ważna jest wartość początkowa wyrażenia błędu. Zwykle przyjmuje się ją równą dx/2.

Sformułujmy algorytm Bresenhama dla tego przypadku:

Wejście

x_1,y_1 - całkowite współrzędne punktu startowego

x_2,y_2 - całkowite współrzędne punktu końcowego

Założenie: $x_2 > x_1$, $y_2 > y_1$, $x_2 - x_1 > y_2 - y_1$

Wyjście

Rysunek rastrowy odcinka łączącego punkt x_1,y_1 z punktem x_2,y_2.

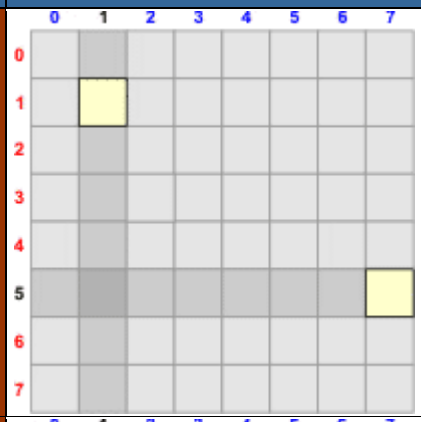
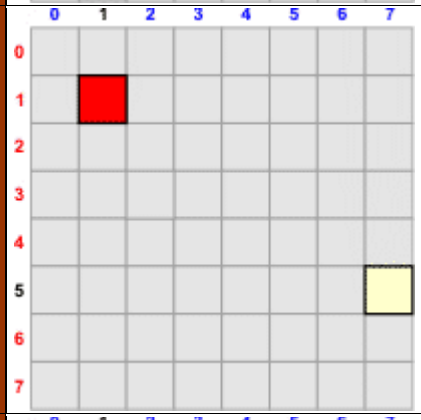
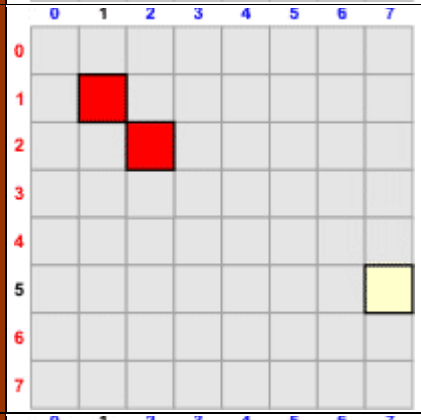
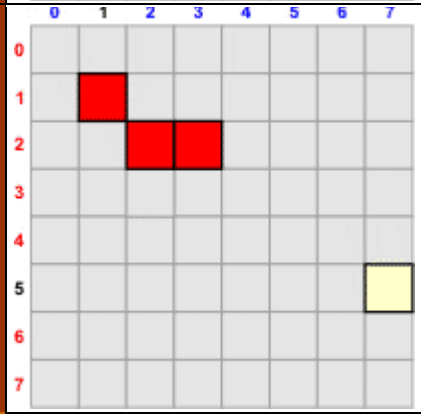
Zmienne pomocnicze

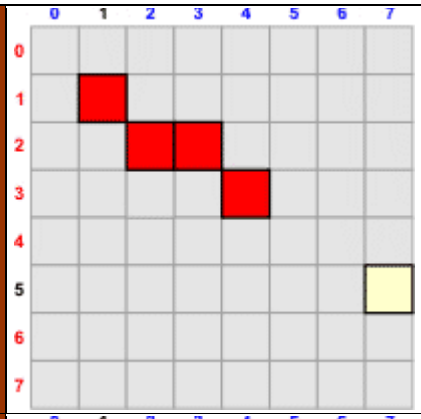
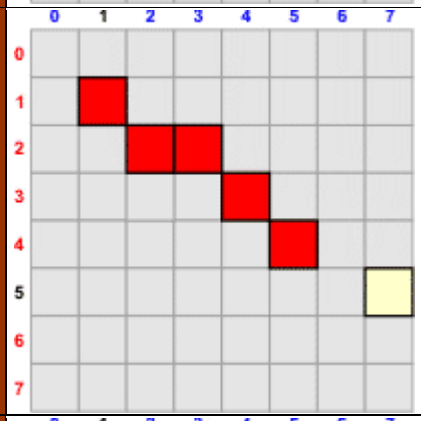
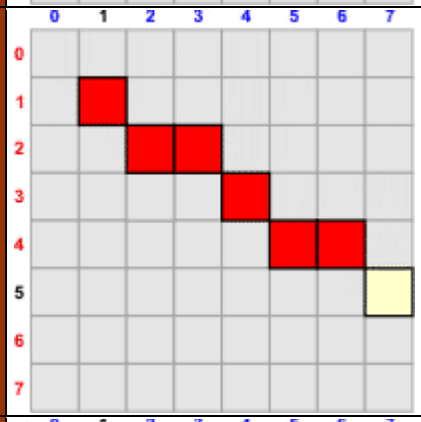
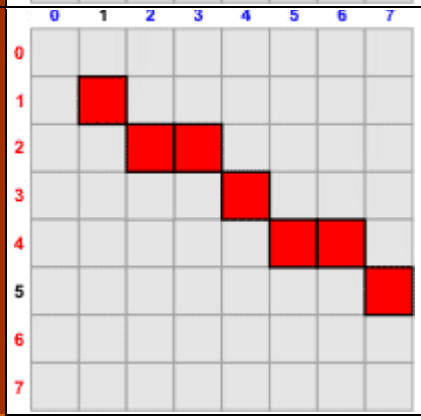
dx - odległość x_2 od x_1
dy - odległość y_2 od y_1
e - wartość wyrażenia błędu

Lista kroków

K01: dx ← x_2 - x_1 *; obliczamy odległości dx i dy*
K02: dy ← y_2 - y_1
K03: e ← dx / 2 *; ustalamy wartość początkową wyrażenia błędu*
K05: Zapal piksel x_1,y_1 *; zapalamy pierwszy piksel odcinka*
K04: Wykonuj dx razy kroki K06...K10 *; w pętli zapalamy kolejne piksele*
K06: x_1 ← x_1 + 1 *; ruch w kierunku szybkim*
K07: e ← e - dy *; modyfikujemy po ruchu wyrażenie błędu*
K08: Jeśli e ≥ 0, idź do kroku K11 *; jeśli e nieujemne, pomijamy ruch w kierunku wolnym*
K09: y_1 ← y_1 + 1 *; ruch w kierunku wolnym*
K10: e ← e + dx *; po ruchu wolnym modyfikujemy wyrażenie błędu*
K11: Zapal piksel x_1,y_1 *; zapalamy kolejne piksele odcinka*
K12: Koniec

Prześledzimy działanie tego algorytmu na prostym przypadku.

Lp.	Obrazek	Opis
1.		Połączymy odcinkiem punkty: (1,1) i (7,5) Zatem: $x_1 = 1, y_1 = 1$ $x_2 = 7, y_2 = 5$ $dx = x_2 - x_1 = 7 - 1 = 6$ $dy = y_2 - y_1 = 5 - 1 = 4$ $e = {}^{dx}/_{2} = {}^{6}/_{2} = 3$
2.		Zapalamy pierwszy punkt na współrzędnych x_1,y_1, czyli (1,1)
3.		Na osi X przesuwamy się o 1 (kierunek szybki): $x_1 = x_1 + 1 = 1 + 1 = 2$ Modyfikujemy wyrażenie błędu: $e = e - dy = 3 - 4 = -1$. Ponieważ e jest ujemne, wykonujemy ruch w kierunku wolnym Y: $y_1 = y_1 + 1 = 1 + 1 = 2$ Modyfikujemy wyrażenie błędu: $e = e + dx = -1 + 6 = 5$ Zapalamy punkt (x_1,y_1), czyli (2,2).
4.		Na osi X przesuwamy się o 1. $x_1 = x_1 + 1 = 2 + 1 = 3$ Obliczamy wyrażenie błędu: $e = e - dy = 5 - 4 = 1$ Ponieważ e nie jest ujemne , nie przesuwamy się na osi Y. Zapalamy punkt (3,2).

5.		Na osi X przesuwamy się o 1 i wyliczamy nowe wyrażenie błędu: $x_1 = x_1 + 1 = 3 + 1 = 4$ $e = e - dy = 1 - 4 = -3$. Ponieważ e jest ujemne, wykonujemy ruch w kierunku wolnym Y: $y_1 = y_1 + 1 = 2 + 1 = 3$ Modyfikujemy wyrażenie błędu: $e = e + dx = -3 + 6 = 3$ Zapalamy punkt (x_1,y_1), czyli (4,3).
6.		Na osi X przesuwamy się o 1 i wyliczamy e $x_1 = x_1 + 1 = 4 + 1 = 5$ $e = e - dy = 3 - 4 = -1$. Ponieważ e jest ujemne, wykonujemy ruch w kierunku wolnym Y: $y_1 = y_1 + 1 = 3 + 1 = 4$ Modyfikujemy wyrażenie błędu: $e = e + dx = -1 + 6 = 5$ Zapalamy punkt (x_1,y_1), czyli (5,4).
7.		Na osi X przesuwamy się o 1 i wyliczamy nowe e: $x_1 = x_1 + 1 = 5 + 1 = 6$ $e = e - dy = 5 - 4 = 1$ Ponieważ e nie jest ujemne , nie przesuwamy się na osi Y. Zapalamy punkt (6,4).
8.		Ostatni krok algorytmu: $x_1 = x_1 + 1 = 6 + 1 = 7$ $e = e - dy = 1 - 4 = -3$. Ponieważ e jest ujemne, wykonujemy ruch w kierunku wolnym Y: $y_1 = y_1 + 1 = 4 + 1 = 5$ $e = e + dx = -3 + 6 = 3$ Zapalamy punkt (x_1,y_1), czyli (7,5). Jest to końcowy punkt odcinka. Zadanie wykonane.

Przedstawiony powyżej algorytm Bresenhama działa poprawnie jedynie dla przypadku, gdy odcinek tworzy z osią OX kąt mniejszy lub równy 45°. Pokażemy teraz, jak rozwiązać pozostałe przypadki.

Ogólny algorytm Bresenhama

Poniżej przedstawiamy pełny algorytm Bresenhama, który rysuje odcinki pomiędzy dowolnymi punktami na powierzchni graficznej. Inne przypadki nachyleń są sprowadzane do przypadku podstawowego przez odpowiednią podmianę współrzędnych punktów odcinka.

Wejście

x_1,y_1 - całkowite współrzędne punktu startowego

x_2,y_2 - całkowite współrzędne punktu końcowego

Wyjście

Rysunek rastrowy odcinka łączącego punkt x_1,y_1 z punktem x_2,y_2. Współrzędne są dowolne, całkowite.

Dane pomocnicze

dx - odległość współrzędnych x_1 i x_2

dy - odległość współrzędnych y_1, y_2

kx - krok po osi X, 1 lub -1

ky - krok po osi y, 1 lub -1

e - wartość wyrażenia błędu

Lista kroków

K01: Jeżeli $x_1 \leq x_2$, to kx ← 1, inaczej kx ← -1 *; określamy krok X od x1 do x2*
K02: Jeżeli $y_1 \leq y_2$, to ky ← 1, inaczej ky ← -1 *; określamy krok Y od y1 do y2*
K03: dx ← $|x_2 - x_1|$ *; odległość pomiędzy x1 i x2*
K04: dy ← $|y_2 - y_1|$ *; odległość pomiędzy y1 i y2*
K05: Zapal piksel x_1,y_1 *; pierwszy piksel odcinka*
K06: Jeżeli dx < dy, idź do kroku K16 *; dla kątów > 45° wykonujemy wersję algorytmu z przestawionymi współrzędnymi*
K07: e ← dx / 2 *; obliczamy wartość początkową wyrażenia błędu*
K08: Powtarzaj dx razy kroki K09...K14 *; rysujemy pozostałe piksele w pętli*
K09: x_1 ← x_1 + kx *; przesuwamy się w odpowiednią stronę w kierunku szybkim*
K10: e ← e - dy *; obliczamy wyrażenie błędu*
K11: Jeżeli e ≥ 0, idź do kroku K14 *; jeśli wyrażenie błędu jest nieujemne, pomijamy ruch w kierunku wolnym*
K12: y_1 ← y_1 + ky *; przesuwamy się w odpowiednią stronę w kierunku wolnym*
K13: e ← e + dx *; obliczamy nowe wyrażenie błędu*
K14: Zapal piksel x_1.y_1 *; kolejny piksel odcinka*
K15: Zakończ
K16: e ← dy / 2 *; wersja algorytmu Bresenhama ze zamienionymi współrzędnymi x i y*
K17: Powtarzaj dy razy kroki K18...K23
K18: y_1 ← y_1 + ky
K19: e ← e - dx
K20: Jeżeli e ≥ 0, idź do kroku K23
K21: x_1 ← x_1 + kx
K22: e ← e + dy

K23: Zapal piksel x_1,y_1

K24: Zakończ

Na podstawie podanego powyżej algorytmu tworzymy funkcję biblioteczną gfxLine(), rysującą dowolny odcinek w obszarze graficznym. Funkcja jest już zoptymalizowana w celu przyspieszenia obliczeń adresów pikseli. Jednakże realizuje ona dokładnie algorytm Bresenhama:

UWAGA: Poniższy kod funkcji gfxLine() dopisz do końca pliku SDL_gfx.cpp.

```
// gfxLine() rysuje odcinek pomiędzy zadanymi
punktami
// screen - wskaźnik struktury SDL_Surface
// x1,y1  - współrzędne punktu startowego
// x2,y2  - współrzędne punktu końcowego
// color  - kolor odcinka
//-----------------------------------------------
-----------------

void gfxLine(SDL_Surface * screen, Sint32 x1,
Sint32 y1, Sint32 x2, Sint32 y2, Uint32 color)
{
  Uint8 * p;
  int dx, dy, kx, ky, e;

  kx = (x1 <= x2) ? 4 : -4;
  ky = (y1 <= y2) ? screen->pitch : -screen->pitch;

  dx = x2 - x1; if(dx < 0) dx = -dx;
  dy = y2 - y1; if(dy < 0) dy = -dy;

  p = (Uint8 *)screen->pixels + y1 * screen->pitch
+ (x1 << 2);
  * (Uint32 *) p = color;

  if(dx >= dy)
  {
    e = dx >> 1;
    for(int i = 0; i < dx; i++)
    {
      p += kx;  e -= dy;
      if(e < 0)
      {
        p += ky; e += dx;
```

```
        }
        * (Uint32 *) p = color;
      }
    }
    else
    {
      e = dy >> 1;
      for(int i = 0; i < dy; i++)
      {
        p += ky; e -= dx;
        if(e < 0)
        {
          p += kx; e += dy;
        }
        * (Uint32 *) p = color;
      }
    }
}
```

UWAGA: Na końcu pliku nagłówkowego SDL_gfx.h dopisz:

```
void gfxLine(SDL_Surface * screen, Sint32 x1,
Sint32 y1, Sint32 x2, Sint32 y2, Uint32 color);
```

Funkcja gfxLine() posiada następujące parametry:

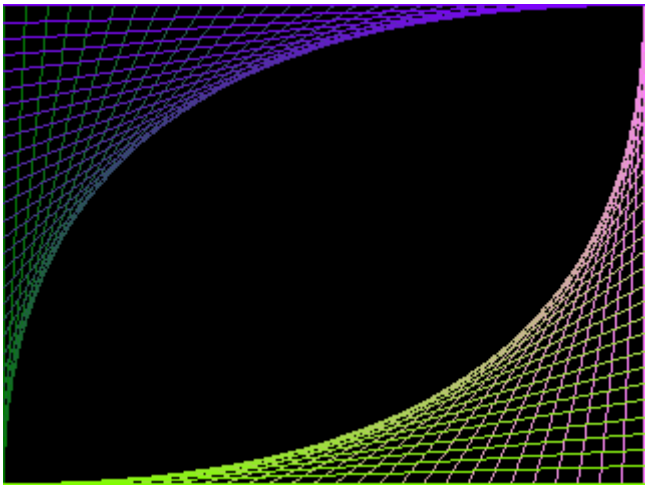
screen - wskaźnik do zainicjowanej struktury typu SDL_Surface. Utworzona wcześniej powierzchnia musi zawierać piksele 32 bitowe oraz być zablokowana
x1, y1 - współrzędne punktu startowego odcinka - muszą się zawierać w obszarze graficznym
x2, y2 - współrzędne punktu końcowego odcinka - muszą się zawierać w obszarze graficznym
color - 32 bitowy kod koloru odcinka

Rysowanie linii otwiera przed nami świat grafiki komputerowej, ograniczony jedynie naszą wyobraźnią i umiejętnościami. Poniższy program, oprócz testowania nowej funkcji bibliotecznej gfxLine(), rysuje całkiem fajny obrazek.

```
//
// P008 - test linii
//------------------

#include <windows.h>
#include <SDL/SDL_gfx.h>

const int SCRX = 320; // stałe określające
szerokość i wysokość
```

```
const int SCRY = 240; // ekranu w pikselach

int main(int argc, char *argv[])
{

  if(SDL_Init(SDL_INIT_VIDEO))
  {
    MessageBox(0, SDL_GetError(), "Błąd
inicjalizacji SDL", MB_OK);
    exit(-1);
  }

  atexit(SDL_Quit);

  SDL_Surface * screen;

  if(!(screen = SDL_SetVideoMode(SCRX, SCRY, 32,
SDL_HWSURFACE)))
  {
    MessageBox(0, SDL_GetError(), "Błąd tworzenia
bufora obrazowego", MB_OK);
    exit(-1);
  }

  if(SDL_MUSTLOCK(screen))
    if(SDL_LockSurface(screen) < 0)
    {
      MessageBox(0, SDL_GetError(), "Błąd blokady
bufora obrazowego", MB_OK);
      exit(-1);
    }

  const int MAXP = 30;  // określa liczbę
rysowanych linii
  int x,y;              // do wyliczania
współrzędnych
  Uint32 c,r,g,b;       // kolor

// Rysujemy MAXP linii.

  for(int i = 0; i < MAXP; i++)
  {
```

```
// Obliczamy położenia punktów równomiernie
rozłożonych na brzegach ekranu

    x = ((screen->w - 1) * i) / (MAXP - 1);
    y = ((screen->h - 1) * i) / (MAXP - 1);

// Teraz wyznaczamy składowe koloru RGB na
podstawie numeru linii

    r = (127 * i) / (MAXP - 1);
    b = (255 * i) / (MAXP - 1);
    g = 128 - r;

// Tworzymy z nich kod koloru piksela

    c = (r << 16) | (g << 8) | b;

// Rysujemy linie

    gfxLine(screen, x, 0, 0, screen->h - y - 1, c);
    gfxLine(screen, screen->w - x - 1, screen->h -
1, screen->w - 1, y, c ^ 0xffffff);

  }

  if(SDL_MUSTLOCK(screen))
SDL_UnlockSurface(screen);

  SDL_UpdateRect(screen, 0, 0, 0, 0);

  MessageBox(0, "Kliknij przycisk OK", "Koniec",
MB_OK);
  return 0;
}
```

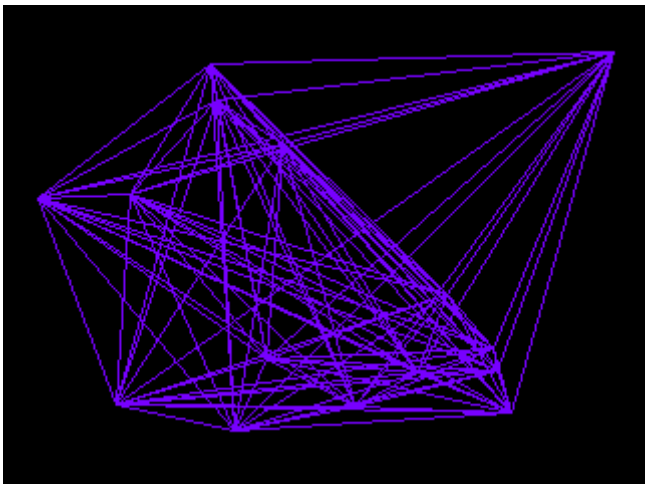


Kolejny program generuje n punktów o losowych współrzędnych, a następnie łączy je ze sobą liniami - powstaje graf zupełny. Wygenerowane punkty zapamiętujemy w tablicy. Wybieramy z tablicy kolejne punkty od 0 do n - 2 (czyli do przedostatniego), i-ty punkt łączymy ze wszystkimi następnymi punktami w tablicy. W ten sposób każda para punktów zostanie połączona odcinkiem.

```
//
// P009 - n wierzchołkowy graf zupełny
//-------------------------------------

#include <windows.h>
#include <SDL/SDL_gfx.h>
#include <time.h>

const int SCRX = 320; // stałe określające
szerokość i wysokość
const int SCRY = 240; // ekranu w pikselach

int main(int argc, char *argv[])
{

  if(SDL_Init(SDL_INIT_VIDEO))
  {
    MessageBox(0, SDL_GetError(), "Błąd
inicjalizacji SDL", MB_OK);
    exit(-1);
  }

  atexit(SDL_Quit);

  SDL_Surface * screen;
```

```
  if(!(screen = SDL_SetVideoMode(SCRX, SCRY, 32,
SDL_HWSURFACE)))
  {
    MessageBox(0, SDL_GetError(), "Błąd tworzenia
bufora obrazowego", MB_OK);
    exit(-1);
  }

  if(SDL_MUSTLOCK(screen))
    if(SDL_LockSurface(screen) < 0)
    {
      MessageBox(0, SDL_GetError(), "Błąd blokady
bufora obrazowego", MB_OK);
      exit(-1);
    }

  const int N = 15;              // liczba
wierzchołków w grafie;
  int i,j,x[N], y[N];

  srand((unsigned)time(NULL));  // inicjujemy
generator liczb pseudolosowych

// generujemy N  punktów o losowych współrzędnych

  for(i = 0; i < N; i++)
  {
    x[i] = rand() % screen->w;
    y[i] = rand() % screen->h;
  }

// rysujemy krawędzie grafu

  for(i = 0; i < N - 1; i++)
    for(j = i + 1; j < N; j++)
      gfxLine(screen, x[i], y[i], x[j], y[j],
0x7700ff);



  if(SDL_MUSTLOCK(screen))
SDL_UnlockSurface(screen);
```

```
  SDL_UpdateRect(screen, 0, 0, 0, 0);

  MessageBox(0, "Kliknij przycisk OK", "Koniec",
MB_OK);
  return 0;
}
```



Przy rysowaniu wielu figur zbudowanych z łamanych niewygodne jest każdorazowe przesyłanie do funkcji współrzędnych początku i końca odcinka. Przydała by się nam funkcja rysująca odcinek, która zapamiętuje koniec poprzedniego odcinka, a nowy odcinek rysuje od zapamiętanego punktu do punktu podanego w parametrach wywołania. Punkt podany w parametrach zostaje zapamiętany, a przy kolejnym wywołaniu funkcji stanie się punktem początku nowego odcinka. Poniżej definiujemy dwie takie funkcje:

gfxMoveTo() - ustala punkt początkowy, od którego rozpoczyna się rysowanie.

gfxLineTo() - rysuje odcinek od końca poprzedniego do podanych współrzędnych, które stają się końcem odcinka dla następnego wywołania.

UWAGA: Poniższy kod funkcji gfxMoveTo() oraz gfxLineTo() dopisz do końca pliku SDL_gfx.cpp.

```
// współrzędne początku odcinka dla gfxLineTo
//-------------------------------------------------
-----------------

Sint32 gfx_x_coord = 0;
Sint32 gfx_y_coord = 0;

// Funkcja gfxMoveTo() ustawia współrzędne początku
odcinka
```

```
// x,y - współrzędne dla gfx_x_coord i gfx_y_coord
//-------------------------------------------------
-----------------

void gfxMoveTo(Sint32 x, Sint32 y)
{
     gfx_x_coord = x; gfx_y_coord = y;
}

// Funkcja gfxLineTo() rysuje odcinek od
zapamiętanych współrzędnych
// do nowych współrzędnych, które po operacji są
zapamiętywane.
// screen - wskaźnik struktury SDL_Surface
// x,y    - współrzędne końca odcinka
// color  - kod koloru odcinka
//-------------------------------------------------
-----------------

void gfxLineTo(SDL_Surface * screen, Sint32 x,
Sint32 y, Uint32 color)
{
   gfxLine(screen, gfx_x_coord, gfx_y_coord, x, y,
color);
   gfx_x_coord = x; gfx_y_coord = y;
}
```

UWAGA: Na końcu pliku nagłówkowego SDL_gfx.h dopisz:

```
void gfxMoveTo(Sint32 x, Sint32 y);

void gfxLineTo(SDL_Surface * screen, Sint32 x,
Sint32 y, Uint32 color);
```

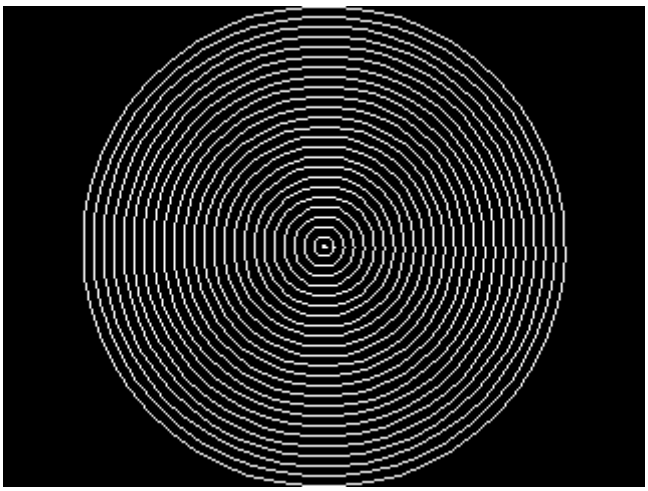
Poniższy program wykorzystuje funkcje **gfxMoveTo()** i **gfxLineTo()** do narysowania okręgów - okręgi rysujemy jako wielokąty foremne o odpowiednio dużej liczbie boków, np. 100. Wyliczamy kolejne współrzędne wierzchołków wielokąta, które następnie łączymy odcinkami - powstaje koło. Wzory są następujące:

$\left.\begin{aligned} x_i &= x_s + r \cdot \cos\left(\frac{2\pi i}{n}\right) \\ y_i &= y_s + r \cdot \sin\left(\frac{2\pi i}{n}\right) \end{aligned}\right\} dla\ i = 0,1,\ldots,n-1$	n - liczba wierzchołków i - numer wierzchołka x_i, y_i - współrzędne i-tego wierzchołka x_s, y_s - współrzędne środka okręgu r - promień okręgu

Okręgi powstałe tą metodą nie są najładniejsze - później poznamy lepsze algorytmy rysowania ładnych, gładkich okręgów, które zupełnie nie wymagają stosowania uciążliwych funkcji trygonometrycznych. Na razie zadowolimy się tym, co mamy.

```
//
// P010 - rysowanie okręgów
//-------------------------

#include <windows.h>
#include <SDL/SDL_gfx.h>
#include <math.h>

const int SCRX = 320; // stałe określające
szerokość i wysokość
const int SCRY = 240; // ekranu w pikselach

int main(int argc, char *argv[])
{

  if(SDL_Init(SDL_INIT_VIDEO))
  {
    MessageBox(0, SDL_GetError(), "Błąd
inicjalizacji SDL", MB_OK);
    exit(-1);
  }

  atexit(SDL_Quit);

  SDL_Surface * screen;

  if(!(screen = SDL_SetVideoMode(SCRX, SCRY, 32,
SDL_HWSURFACE)))
  {
    MessageBox(0, SDL_GetError(), "Błąd tworzenia
bufora obrazowego", MB_OK);
    exit(-1);
  }

  if(SDL_MUSTLOCK(screen))
    if(SDL_LockSurface(screen) < 0)
    {
      MessageBox(0, SDL_GetError(), "Błąd blokady
bufora obrazowego", MB_OK);
```

```
    exit(-1);
  }

  const int N  = 50;        // liczba wierzchołków
  const int LO = 25;        // liczba okręgów
  int xs = screen->w / 2;
  int ys = screen->h / 2;
  int rs  = ys - 1;

  for(int k = 0; k < LO; k++)
  {
    int r = 1 + (rs * k) / (LO - 1);
    gfxMoveTo(xs + r, ys);
    for(int i = 1; i <= N; i++)
      gfxLineTo(screen, xs + r * cos(6.283185 * i / N), ys + r * sin(6.283185 * i / N), 0xffffff);
  }

  if(SDL_MUSTLOCK(screen))
SDL_UnlockSurface(screen);

  SDL_UpdateRect(screen, 0, 0, 0, 0);

  MessageBox(0, "Kliknij przycisk OK", "Koniec",
MB_OK);

  return 0;
}
```



Ćwiczenia

Napisz programy tworzące następujące rysunki: