イメージの制御

ピクセルを操作する

Bitmap クラスは、ビットマップが持つピクセルの情報に簡単にアクセスできるように
座標を指定することで、ピクセルのカラーを取得/設定するための手段を提供しています
これを用いれば、イメージに何らかのフィルタ処理を施したりすることができます

ピクセルのカラー情報を得るには **Bitmap.GetPixel()** メソッドを使います

public Color GetPixel(int x , int y);

x には X 座標を、y には Y 座標を表す数値を指定します
これによって、指定した座標のピクセルの色を表す Color オブジェクトを返します
ピクセルの色を設定するには **Bitmap.SetPixel()** メソッドを用います

public void SetPixel(int x , int y , Color color);

x と y には、ピクセルを設定する X 座標と Y 座標を指定します
color は指定した座標の新しい色を表す Color オブジェクトです

これによって、既存のイメージに様々な処理をピクセル単位で行ったり
あるいはイメージのサイズだけを指定した空のイメージを作成して
これにピクセル単位で色を設定することで、新しいイメージを作ることができます

まだ説明していませんでしたが、Color クラスから色の数値表現を得るには
Color.ToArgb() メソッドを使います
また、アルファ値と各色の要素の値を個別に分割したい場合は
A、**R**、**G**、**B** プロパティを使えば、それぞれの要素の値だけを得ることができます

public int ToArgb();
public byte A {get;}
public byte R {get;}
public byte G {get;}
public byte B {get;}

これらから、色の数値情報を取得することができます
この機能を使えば、GetPixel() メソッドで得た Color オブジェクトの数値表現を得
それに適当な演算処理を加えて再び Color オブジェクトにして
そして、SetPixel() メソッドを使って新しい色を設定するという処理ができます

```
using System.Windows.Forms;
using System.Drawing;

class WinMain : Form {
 Bitmap img;
 public static void Main(string[] args) {
  Application.Run(new WinMain());
 }
 public WinMain() {
  img = new Bitmap("test.jpg");
  for (int i = 0 ; i < img.Height ; i++) {
   for (int j = 0 ; j < img.Width ; j++) {
    int px = ~(img.GetPixel(j , i)).ToArgb();
    img.SetPixel(j , i , Color.FromArgb(px));
   }
  }
 }

 override protected void OnPaint(PaintEventArgs e) {
  Graphics g = e.Graphics;
  g.DrawImage(img , 0 , 0);
 }
}
```

このプログラムは、コンストラクタで test.jpg ファイルを読み取り
このイメージの左上角から右下角までの全ピクセルをループで処理してしまいます
GetPixel() メソッドで Color オブジェクトを得て、それを ToArgb() メソッドで数値化します
さらに、数値を ~ 論理演算子で否定し、色を反転させます

最後に、同じピクセル部分に反転した色を設定しています
この処理を繰り返し行い、全てのピクセルを反転してからウィンドウを表示します
結果、上の図のようにオリジナルのイメージの色を反転させたものが表示されるのです

ダブルバッファリング

ビットマップをメモリ上に配置できるようになれば、ある程度自由に静止画を扱えますが
上記のような1ピクセル単位での制御は極めて緻密な作業であるといえます

出力先がディスプレイでもプリンタでもプロッタでも
同一の操作を行って同じ図形を出力できるべきであるというのが GDI の思想です
抽象化することで、ハードウェアごとに描画プログラムを書きなおす必要をなくすのです

メモリ上に展開されているビットマップにも、この考え方は当てはまります
.NET で具体的に考えれば、ビットマップに対する図形の描画にも
Graphics クラスの機能を使えるべきだと、GDI をよく知るプログラマなら誰もが思うでしょう

予想通り、**Graphics.FromImage()** という静的メソッドが用意されています
このメソッドは、Image オブジェクトの DC をパックした Graphics オブジェクトを返します

public static Graphics FromImage(Image image);

image には Graphics オブジェクトの描画対象となる Image オブジェクトを指定します
メソッドは、image のデバイスコンテキストを表す Graphics オブジェクトを返します

この機能を用いれば、複雑な描画処理をウィンドウに行うのではなく
メモリ上のビットマップに行うことで、その描画過程をユーザーから隠蔽できます
とくに、画面クリア時に発生するちらつきの防止などでこの手法が使われます
このようなバッファを使った処理を**ダブルバッファリング**と呼びます

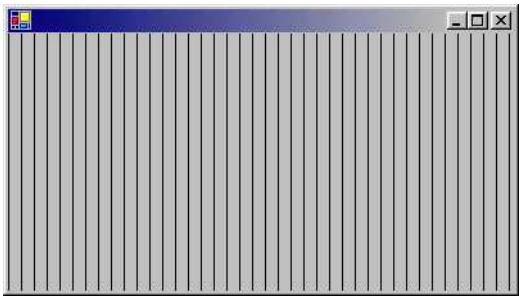
```
using System.Windows.Forms;
using System.Drawing;

class WinMain : Form {
 public static void Main(string[] args) {
  Application.Run(new WinMain());
 }

 override protected void OnPaint(PaintEventArgs e) {
  Graphics g = e.Graphics;
  Image img = new Bitmap(400 , 200);
  Graphics buf = Graphics.FromImage(img);

  for (int i = 0 ; i < 400 ; i += 10)
   buf.DrawLine(Pens.Black , i , 0 , i , 200);

  g.DrawImage(img , 0 , 0);
 }
}
```



DrawLine() メソッドで連続した縦線を引いているのは img オブジェクトです
この時点では、まだウィンドウのクライアント領域に描画されていないことに注意してください
イメージに対する描画が全て終わった後に、DrawImage() を使って
作成した img オブジェクトをウィンドウのクライアント領域に描画しています

前のページへ　戻る　次のページへ