CodeGear from Borland
Developer Camp

画像処理アルゴリズムと高速画像処理手法

株式会社シリアルゲームズ　シニアエンジニア　細川淳

CodeGear from Borland

Developer Camp

画像の種類や色空間

画像の種類 - ベクター

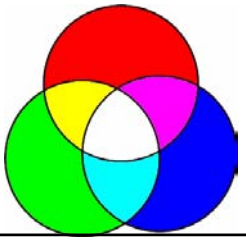
- ベクターグラフィックス
  - データが小さい
  - スケーラブル
  - 画像データが情報を持っている
    - 点と線や塗りの方法など
  - 比較的単純な図形に優れる
    - 単純な図形などの人工的な画像向け
  - Adobe Flash や Adobe Illustrator などで利用

画像の種類 - ラスター

- ラスターグラフィックス
  - データが大きい
  - 多くのデバイスに、そのまま出力できる
    - 光速船などの電子銃を操作するものには不向き
  - 画像データは情報を持たない
    - Exif などは画像データの描画を制限しない
  - 複雑な図形に優れる
    - 自然画や絵画など
  - Adobe Photoshop や画像を扱う大多数のソフト

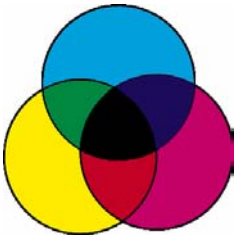
色空間 - RGB

- RGB - Red Green Blue
  - 光の三原色　赤・緑・青　で色を表す
    - 各色8bitの24bitで表している場合 TrueColor と呼ばれる
    - 透明度 α を8bit加えた 32bit TrueColor もある
      - RGBA
  - モニタなどの自身が発色するデバイスで用いられる
    - →大多数のソフトで使用される一般的な色空間
  - 加法混色
  - sRGB規格



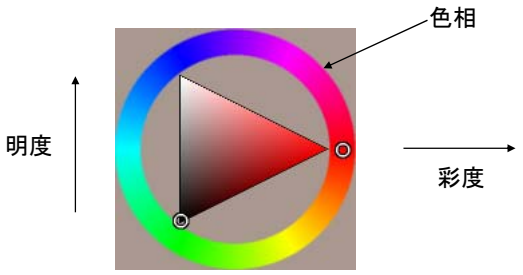
色空間 - CMYK

- CMYK - Cyan Magenta Yellow KeyPlate
  - インクの3原色　シアン・マゼンタ・イエロー　とブラックで色を表す
  - 印刷などで用いられる
  - 減法混色



色空間 - HSV

- HSV - Hue Saturation Value
  - 色相・彩度・明度で色を表す
  - 多くのペイントソフトで用いられる
    - Corel Painter など

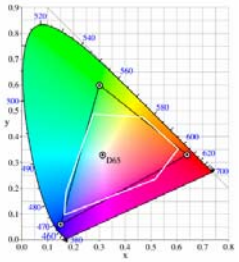


色空間 - HLS

- HLS - Hue Luminance Saturation
  - 色相・輝度・彩度で色を表す
  - HSVと似ているが人間の見た目の色を考慮した輝度によって色を表す

色空間 - その他

- その他にも YIQ などがある
- 表色系
  - CIE表色系のほかにも、マンセル表色系やPCCSといったモノがある



CIE表色系(*1)

Delphi / Windows 上の画像

Delphi 上の画像処理

- Delphi での既存の画像処理方法
  - TImageやTPictureによる画像表示
  - TCanvasでラッピングされている Win32 API
    - Ellipseや、Drawなど
- Delphi に実装されていない Win32API
  - リージョン
  - GDI+

GDI 上の画像処理

- ビット転送処理
  - BitBlt系のビット転送処理
    - ラスタオペレーション
    - 行列演算や座標変換
- ペンとブラシによるプリミティブな図形
- テキストの描画
- メタファイル
- リージョンによるクリッピング処理
- GDI+

DIB

- DIB - Deveice Independent Bitmap
  - デバイス独立ビットマップ
- ビットマップをメモリ上で表現したもの
  - メモリ上には色が、そのまま入っている
    - パレットを使用した場合パレット番号が入る
- 各色 8bit ＋α 8bit の計32bit で表すと Intel 32bit プロセッサで最適なパフォーマンスを得られる

DIB Section

- 通常、GDIベースの描画機能は、DDB でしか使用できない
- DIB Section は、GDIベースの描画機能も使用できる
- 時間が掛かる処理は、自前のルーチンで、面倒だったり、時間が掛からない処理には GDI を使用するなどの切り分けができる

CodeGear from Borland

Developer Camp

画像アルゴリズム

Developer Camp

画像処理の高速化手法

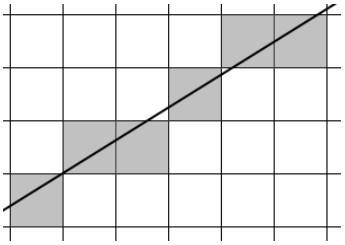
- アルゴリズムの再検討
  - ループ処理を展開してみる
- ラッパーを介さない
  - VCL や Win32API を使わない
- 代替手段を探す
  - デバイスへのアクセス方法
  - DirectXの検討など
- 言語の検討
  - 機械語を使えるか？

画像処理アルゴリズム - DDA

- DDA
  - Digital Differential Analyzer
  - デジタル微分解析器
  - 図形アルゴリズムの基本
  - 線形補間を行う

画像処理アルゴリズム - DDA

- 基本的には線の方程式と同じ
  $ay = bx + c \rightarrow y = (b/a)x + (b/a)c$
- b/aの値を算出する時に、除算しない
- a - b < 0 になったとき、y 値が増加する



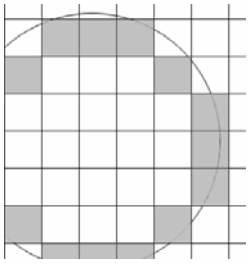
画像処理アルゴリズム - 円

- 円
  - 円の描画には DDA と同様な考え方を使う
  - ブレゼンハム（Bresenham）の円アルゴリズムといわれるアルゴリズムが有名

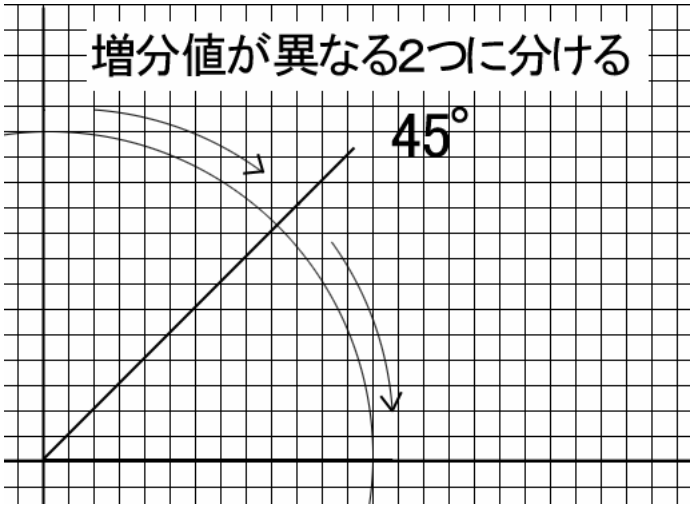
画像処理アルゴリズム - 円

- 円の方程式を、DDAのように考える

  x^2 + y^2 = r^2 → y^2 = r^2 - x^2
- X値を自乗した時、半径rの自乗値と比べ、rの自乗を超えない範囲に点を打つ
- １象限で増分値が異なる2つを分けて描く



画像処理アルゴリズム - 円

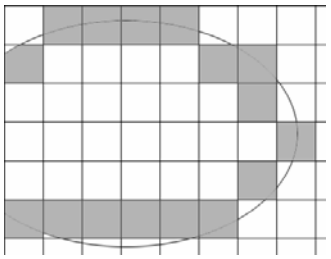


画像処理アルゴリズム - 楕円

- 円と同様に考える

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1$$ → y^2 = b^2 - b^2 * x^2 / a^2

- 円にはない焦点距離があるため別々のアルゴリズムにしたほうが効率的



画像処理アルゴリズム - 矩形

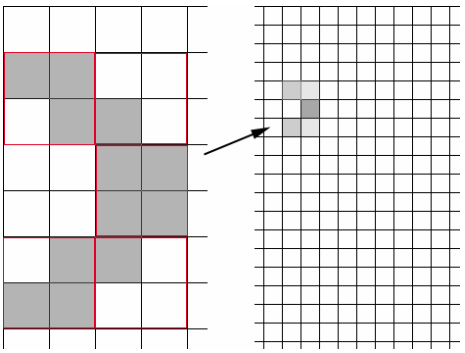
- 矩形は4つの直線の集まりと考えられる
- DDAを用いるのは、もったいない
- メモリを、その方向に向けて埋めてしまう方法を用いる
  - X方向ならば、VCL の FillChar
  - Y方向ならば、幅の分だけ足しながら1ピクセル入れていく

画像処理アルゴリズム - アンチエイリアス

- アンチエイリアスは、画像の縁を滑らかにする技術
- 色々なアルゴリズムが考案されている
- 今回は、最も簡単な縮小時の加算平均をとる方法を紹介

画像処理アルゴリズム - アンチエイリアス

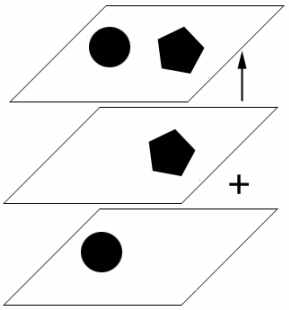
- 元画像の4倍の画像を作り、それを縮小する
- 縮小時、加算平均をとる



赤で囲まれた4つのピクセルの
色の平均値で、1ピクセルを塗る

画像処理アルゴリズム - 加算

- 画像の重ね合わせ方で、色を加算していくこと
- 画像の重ねあわせを高速に実行できればレイヤーを実現できる



全体の見た目

レイヤー1

レイヤー2

MMX

MMXについて

- MMX - Multi Media eXtension
- Pentium MMX から使用できる機械語セット
- FPU（浮動小数点演算ユニット）に備わっているレジスタを使用して128bit演算を行える
- 現在では、MMX に加え、SSE , SSE2 などもある（AMD 系では、3DNow!, 3DNow!2がある）

CPUの判定

- MMXは、使用できる CPU が限られているため CPU を判定する必要がある
  - とはいえ、最近のPCでは未サポートCPUは載っていないだろうが……

CPUの判定 - メーカー判定

```
procedure CheckCPUAbility;
var
  Manifacture: array [0.. 11] of Char;
begin
  try
    asm
      push esi
      push edi
      push ebx
```

CPUの判定 - AMD判定

```
// CPU Manifacture Check
lea esi, Manifacture
mov edi, esi

xor eax, eax
mov ecx, 12
rep stosb

cpuid
```

```
mov [esi + 0], ebx
mov [esi + 4], edx
mov [esi + 8], ecx

// AMD Check
cmp ebx, 'htuA'
jnz @@Start

cmp edx, 'itne'
jnz @@Start

cmp ecx, 'DMAc'
jnz @@Start

// AMD
mov IsAMD, True
```


CPUの判定 - MMX/SSE判定

```
// SSE Check
@@Start:
  mov eax, 1
  cpuid
```

```
@@MMX: // MMX
  test edx, 1 shl 23
  jz @@SSE
  mov MMXEnabled, True

@@SSE: // SSE
  test edx, 1 shl 25
  jz @@SSE2
  mov SSEEnabled, True

@@SSE2: // SSE2
  test edx, 1 shl 26
  jz @@3DNow
  mov SSE2Enabled, True
```

CPUの判定 - 3DNow!判定

```
@@3DNow: // AMD 3D Now!
  cmp IsAMD, True
  jnz @@Exit

  mov eax, $80000001
  cpuid
```

```
  test edx, 1 shl 31
  jz @@3DNow2
  mov AMD3DNowEnabled, True

@@3DNow2: // AMD 3D Now! 2
  test edx, 1 shl 30
  jz @@Exit
  mov AMD3DNow2Enabled, True
```


CPUの判定 - 終了処理

```
@@Exit:
  pop ebx
  pop edi
  pop esi
end;
except
  // not support CPU ID
end;
```

```
  CPUManifacture := String(Manifacture);
end;
```

アルゴリズムの実装

DIBSection の生成

- 全ての画像アルゴリズムで使用する DIBSection の作成方法を紹介する
- DIBSection は、メモリ操作とGDIによる描画関数が両方とも使える便利なビットマップ

DIBSection の生成

```
procedure TLayer.CreateDIB(const vWidth, vHeight: Integer);
var
  Info: PBitmapInfo;
  DC: HDC;
begin
  FWidth := vWidth;
  FHeight := vHeight;

  if (FWidth = 0) or (FHeight = 0) then
    Exit;

  FSize := ((FWidth shl 2) and $fffffffc) * Cardinal(FHeight);
  FDWordSize := FSize shr 2;
```


DIBSection の生成

```
  try
    GetMem(Info, SizeOf(TBitmapInfoHeader) + 0);
    try
      with Info^, bmiHeader do begin
        biSize := SizeOf(bmiHeader);
        biWidth := FWidth;
        biHeight := -FHeight; // top down Bitmap
        biPlanes := 1;
        biBitCount := 32;
        biCompression := 0;
        biSizeImage := 0;
        biXPelsPerMeter := 0;
        biYPelsPerMeter := 0;
        biClrUsed := 0;
        biClrImportant := 0;
      end;
```

DIBSection の生成

```
  DC := CreateDC('DISPLAY', nil, nil, nil);
  try
    FDIB := CreateDIBSection(
      0,
      Info^,
      DIB_RGB_COLORS,
      Pointer(FMemory),
      0,
      0);

    FDC := CreateCompatibleDC(DC);
    FOldSelected := SelectObject(FDC, FDIB);
    ZeroMemory(FMemory, FSize);
  finally
    DeleteDC(DC);
  end;
finally
  FreeMem(Info);
end;
```


DDA の実装

- DDAは単純なため、機械語による実装でも、コンパイラが吐き出すコードでもあまり違いがない
- そのため、保守容易性なども考慮に入れ Delphi 言語で記述する

DDA の実装

```
procedure DDA(
  vX1, vY1, vX2, vY2: Integer;
  vOnBits: TDDAEvent;
  vData: Pointer);
var
  Flag: Integer;
  X, Y: Integer;
  Sign: Integer;
  XSize, YSize: Integer;
begin
  XSize := abs(vX2 - vX1);
  YSize := abs(vY2 - vY1);
```


DDA の実装

```
  if (XSize > YSize) then
  begin
    Flag := XSize shr 1;

    if (vX1 < vX2) then begin
      if (vY1 < vY2) then
        Sign := +1
      else
        Sign := -1;

      Y := vY1;
```

```
      for X := vX1 to vX2 do begin
        vOnBits(X, Y, vData);

        Dec(Flag, YSize);
        if (Flag < 0) then begin
          Inc(Flag, XSize);
          Inc(Y, Sign);
        end;
      end;
    end
```

矩形の実装

- 矩形は、単純なメモリを埋める処理のため機械語で書いたほうが速く、理解も容易

矩形の実装 - FillMem関数

```
procedure TLayerCanvas.FillMem( // eax -> Self
  const vMem: Pointer;          // edx
  const vLen: Integer;          // ecx
  const vValue: Cardinal);
asm
  push edi

  mov edi, edx
  mov eax, vValue

  rep stosd

  pop edi
end;
```

矩形の実装 - YFill

- X方向は、単純にメモリを埋めるだけだが、Y方向は、1ピクセル描くごとに、幅を足してメモリ上の位置をずらす必要がある

矩形の実装 - YFill

```
procedure TLayerCanvas.YFill(vX, vY, vLength: Integer; const vColor:
   Cardinal);
var
  Mem: PCardinal;
  tmpInt: Integer;
  i: Integer;
begin
  if (vX < 0) or (vX >= FLayer.Width) then
    Exit;

  tmpInt := vY;
  Dec(vLength, -tmpInt + Adjust(vY, FLayer.Height));

  if ((vY + vLength) > FLayer.Height) then
    vLength := FLayer.Height - vY;
```

矩形の実装 - YFill

```
  Mem := FLayer.GetMemory(vX, vY);

  if (Mem = nil) or (vLength < 1) then
    Exit;

  tmpInt := FLayer.Width;

  for i := 0 to vLength - 1 do begin
    Mem^ := vColor;
    Inc(Mem, tmpInt);
  end;
end;
```


円の実装

- 円も、やはり、DDAのように加算のみで実装できるので、Delphi言語による実装で速度的に問題ない（そのためのアルゴリズムともいえる）

円の実装

```
procedure Circle(
  const vDiameter: Integer;
  const vCircleSubProc: TCircleSubProc;
  vValue: Pointer);
var
  X, Y: Integer;
  XP, XN, YP, YN: Integer;
  OrdRadius: Integer;
  Radius: Integer;
  Even: Integer;
  Matrix: array of array of Boolean;
```


円の実装

```
  procedure CallSub(vX, vY: Integer);
  var
    tmpX, tmpY: Integer;
  begin
    tmpX := vX + OrdRadius;
    tmpY := vY + OrdRadius;

    if (not Matrix[tmpX, tmpY]) then begin
      Matrix[tmpX, tmpY] := True;
      vCircleSubProc(vX, vY, vValue);
    end;
  end;
```

円の実装

```
begin
  OrdRadius := vDiameter shr 1;
  Radius := OrdRadius;
  Even := Ord(not Odd(vDiameter));

  X := Radius;
  Y := 0;

  if (Radius > 0) then begin
    SetLength(Matrix, vDiameter + 1, vDiameter + 1);
```


円の実装

```
    while (X >= Y) do begin
      XP := +X - Even;
      XN := -X + Even;
      YP := +Y - Even;
      YN := -Y + Even;
```

円の実装

```
      CallSub(XP, +Y);
      CallSub(XP, YN);

      CallSub(-X, +Y);
      CallSub(-X, YN);

      CallSub(YP, +X);
      CallSub(YP, XN);

      CallSub(-Y, +X);
      CallSub(-Y, XN);
```


円の実装

```
      Dec(Radius, Y shl 1 + 1);
      Inc(Y);

      if (Radius < 0) then begin
        Inc(Radius, (X - 1) shl 1);
        Dec(X);
      end;
    end;
  else begin
    vCircleSubProc(0, 0, vValue);
  end;
end;
```

アンチエイリアスの実装

- アンチエイリアスを行うための加算平均関数を紹介する

アンチエイリアスの実装

```
procedure BiLinear(
  const vDest: TLayer;
  const vDestX, vDestY, vXLen, vYLen: Integer;
  const vSrc: TLayer;
  const vSrcX, vSrcY: Integer);
var
  X, Y: Integer;
  SrcX, SrcY: Integer;
  tmpX, tmpY: Integer;
  R, G, B, A: Integer;
  Count: Integer;
```

アンチエイリアスの実装

```
procedure CalcRGBA(const vX, vY: Integer);
var
  Rt, Gt, Bt, At: Integer;
begin
  ToRGBA(vSrc[vX, vY], Rt, Gt, Bt, At);

  if (At > 0) then begin
    Inc(R, Rt);
    Inc(G, Gt);
    Inc(B, Bt);
    Inc(A, At);
    Inc(Count);
  end;
end;
```


アンチエイリアスの実装

```
function Calc(const vValue: Integer): Integer;
begin
  if (Count = 3) then
    Result := vValue div Count
  else
    Result := vValue shr (Count shr 1)
end;
```

アンチエイリアスの実装

```
begin
  for Y := 0 to vYLen do
    for X := 0 to vXLen do begin
      SrcX := vSrcX + (X shl 1);
      SrcY := vSrcX + (Y shl 1);

      tmpX := vDestX + X;
      tmpY := vDestY + Y;

      R := 0;
      G := 0;
      B := 0;
      A := 0;
      Count := 0;
```


アンチエイリアスの実装

```
      CalcRGBA(SrcX + 0, SrcY + 0);
      CalcRGBA(SrcX + 1, SrcY + 0);
      CalcRGBA(SrcX + 0, SrcY + 1);
      CalcRGBA(SrcX + 1, SrcY + 1);

      vDest[tmpX, tmpY] :=
        Mix(
          vDest[tmpX, tmpY],
          RGBA(Calc(R), Calc(G), Calc(B), A shr 2));
    end;
end;
```

混色アルゴリズム

- アンチエイリアスでも使用された mix 関数を実装する
- mix関数は、2つの色を混ぜる関数
- レイヤーを作成する場合にも有効な関数である

混色アルゴリズム

```
function Mix(const vColor1, vColor2: Cardinal): Cardinal;
var
  R1, G1, B1, A1: Cardinal;
  R2, G2, B2, A2: Cardinal;
  Alpha: Cardinal;

  function Calc(const v1, v2: Cardinal): Cardinal;
  begin
    Result := (v1 * A1 + v2 * A2) shr 8;
  end;

begin
```

混色アルゴリズム

```
if (MMXEnabled) then begin
  asm
    // Alpha1 -> ecx
    // Alpha2 -> eax
    mov ecx, vColor1     // Alpha 1
    mov eax, vColor2     // Alpha 2
    and ecx, AMask       // Alpha 1
    and eax, AMask       // Alpha 2
    shr ecx, AShiftCount // Alpha 1
    shr eax, AShiftCount // Alpha 2
```


混色アルゴリズム

```
    add ecx, eax         // Denom
    cmp ecx, OverAValue  // Denom
    jc @@CalcAlpha       // Denom
    mov ecx, MaxAValue   // Denom

    @@CalcAlpha:
    xor edx, edx         // Calc Alpha 2
    shl eax, 8           // Calc Alpha 2
    test ecx, ecx        // Calc Alpha 2
    jz @@NotDiv          // Calc Alpha 2
    div ecx              // Calc Alpha 2
    @@NotDiv:
    mov edx, OverAValue + 1 // Calc Alpha 1
    sub edx, eax            // Calc Alpha 1
```

混色アルゴリズム

```
// Zero -> mm7
// Alpha1 -> mm1
// Alpha2 -> mm2
pxor mm7, mm7       // Zero
movd mm2, eax       // Alpha 2
movd mm1, edx       // Alpha 1
punpcklwd mm2, mm2  // Alpha 2
punpcklwd mm1, mm1  // Alpha 1
punpcklwd mm2, mm2  // Alpha 2
punpcklwd mm1, mm1  // Alpha 1
```


混色アルゴリズム

```
// Calc
movd mm0, vColor1   // Calc 1
movd mm3, vColor2   // Calc 2
punpcklbw mm0, mm7  // Calc 1
punpcklbw mm3, mm7  // Calc 2
pmullw mm0, mm1     // Calc 1
pmullw mm3, mm2     // Calc 2
```

混色アルゴリズム

```
      // Store -> mm0
      // Alpha
      paddusw mm0, mm3     // Store
      cmp ecx, OverAValue  // Alpha
      jc @@Alpha           // Alpha
      mov ecx, MaxAValue   // Alpha
      @@Alpha:             // Alpha
      shl ecx, AShiftCount // Alpha
      psrlw mm0, 8         // Store
      packuswb mm0, mm7    // Store
      movd eax, mm0        // Store

      and eax, NotAMask
      or eax, ecx
      mov Result, eax
```


混色アルゴリズム

```
      // End
      emms
    end;
end
```

混色アルゴリズム

```
  else begin
    ToRGBA(vColor1, R1, G1, B1, A1);
    ToRGBA(vColor2, R2, G2, B2, A2);

    Alpha := A1 + A2;
    Adjust(Alpha, MaxAValue);

    if (Alpha > 0) then begin
      A2 := {OverAValue * A2}(A2 shl 8) div Alpha;
      A1 := OverAValue - A2;
    end;

    Result := RGBA(Calc(R1, R2), Calc(G1, G2), Calc(B1, B2), Alpha);
  end;
end;
```


まとめ

- 画像を扱うプログラムでは基礎を知っているとプログラムが有利になることがある
- DDAや円アルゴリズムなど加算のみで行うような処理には、あえて機械語を使うメリットはあまりない
- 大量の処理が必要で、32bitずつ扱えるようなデータに対しては MMX が有効かどうか検討する

参考文献

- Wikipedia - 色空間
  - http://ja.wikipedia.org/wiki/%E8%89%B2%E7%A9%BA%E9%96%93
    - *1画像の出典元
- **MMXテクノロジ最適化テクニック**
  - 著者：小鷲 英一
  - 出版社：アスキー

Thank you