

Graphics with Processing



2016-04 色彩とピクセル処理

<http://vilab.org>

塩澤秀和

4.1 色彩

色のデータ形式

□ 色の指定方法

- 1つの数値(グレースケール)
- 3つの数値の組(カラー)
初期モードは RGB 各0~255
- 16進数カラーコード #rrggbb
- color型の変数

□ color型

- 色を表すデータ型(実態はint)
- color関数で合成できる
color(成分1, 成分2, 成分3)
- 例) color c = color(r, g, b);

□ 成分の取得

- red(c), green(c), blue(c),
hue(c), saturation(c),
brightness(c), alpha(c)

半透明の表現

□ アルファ値(p.286)

- 色の第4成分(透過処理用)
- 重ね塗りでの色の混合率
- 例) c = color(r, g, b, a);
- 例) fill(255, 0, 0, 128);

色モードの設定

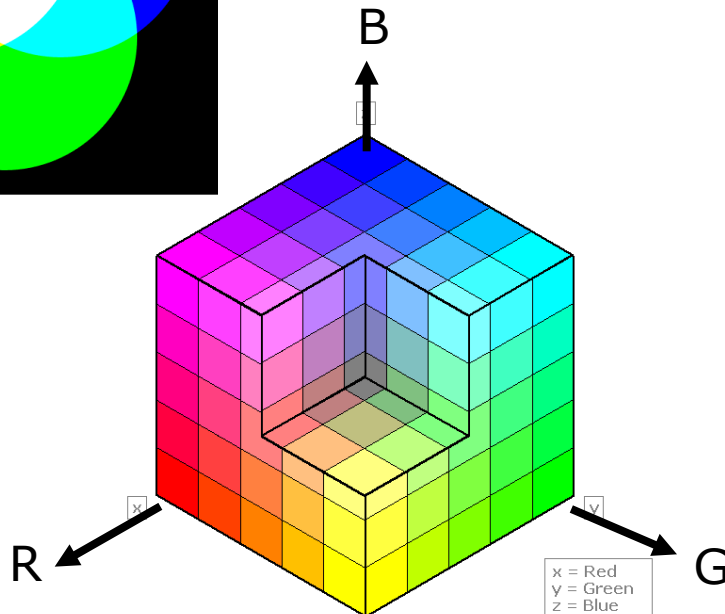
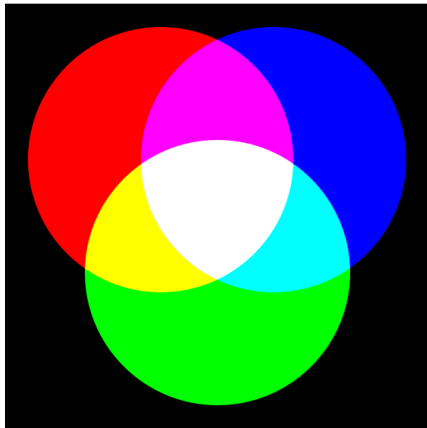
□ colorMode(モード, 値範囲)

- モード: カラーモデル
RGB または HSB
- 値範囲: 成分の上限値
 - colorMode(モード, 範囲1, 範囲2, 範囲3) の形式もある
- 例) colorMode(HSB, 1.0);
- サンプル Basics → Color

4.2 表色系/カラーモデル(p.246)

RGBカラーモデル

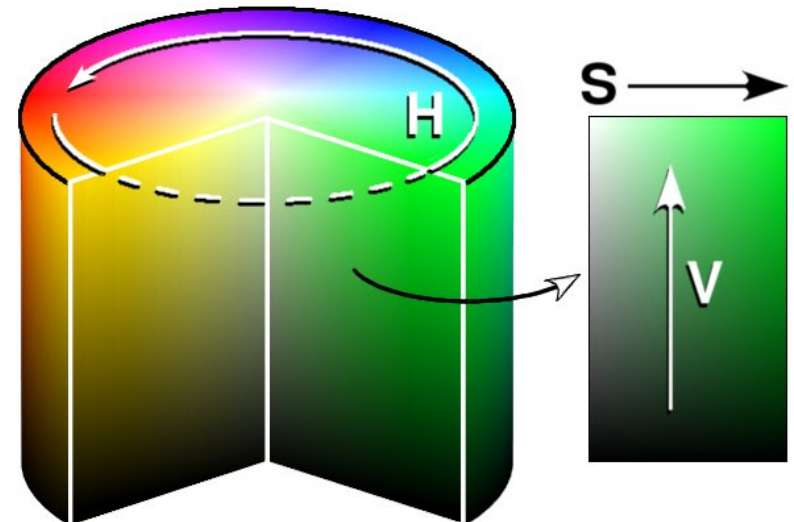
□ 光の三原色(赤, 緑, 青)



HSB(HSV/HSI)カラーモデル

□ 光の三属性

- 色相(H): 色あい
- 彩度(S): あざやかさ
- 明度(B/V/I): 明るさ
- メニュー Tools → Color Selector



4.3 ピクセル処理

ピクセル配列による操作

- ピクセルとは(p.13)
 - 画面を構成する画素1点1点
(pixel ← picture cell)
- ⇒ ラスター表現のグラフィックス
- pixels[]
 - 各画素の色(color型のデータ)を格納する1次元配列
 - 画面座標(x, y)の要素は
pixels[y * width + x]
- loadPixels()
 - ピクセル処理の開始処理
 - 画面の画素ごとの色データを
pixels[]に読み込む
- updatePixels()
 - pixels[]を画面に書き戻す

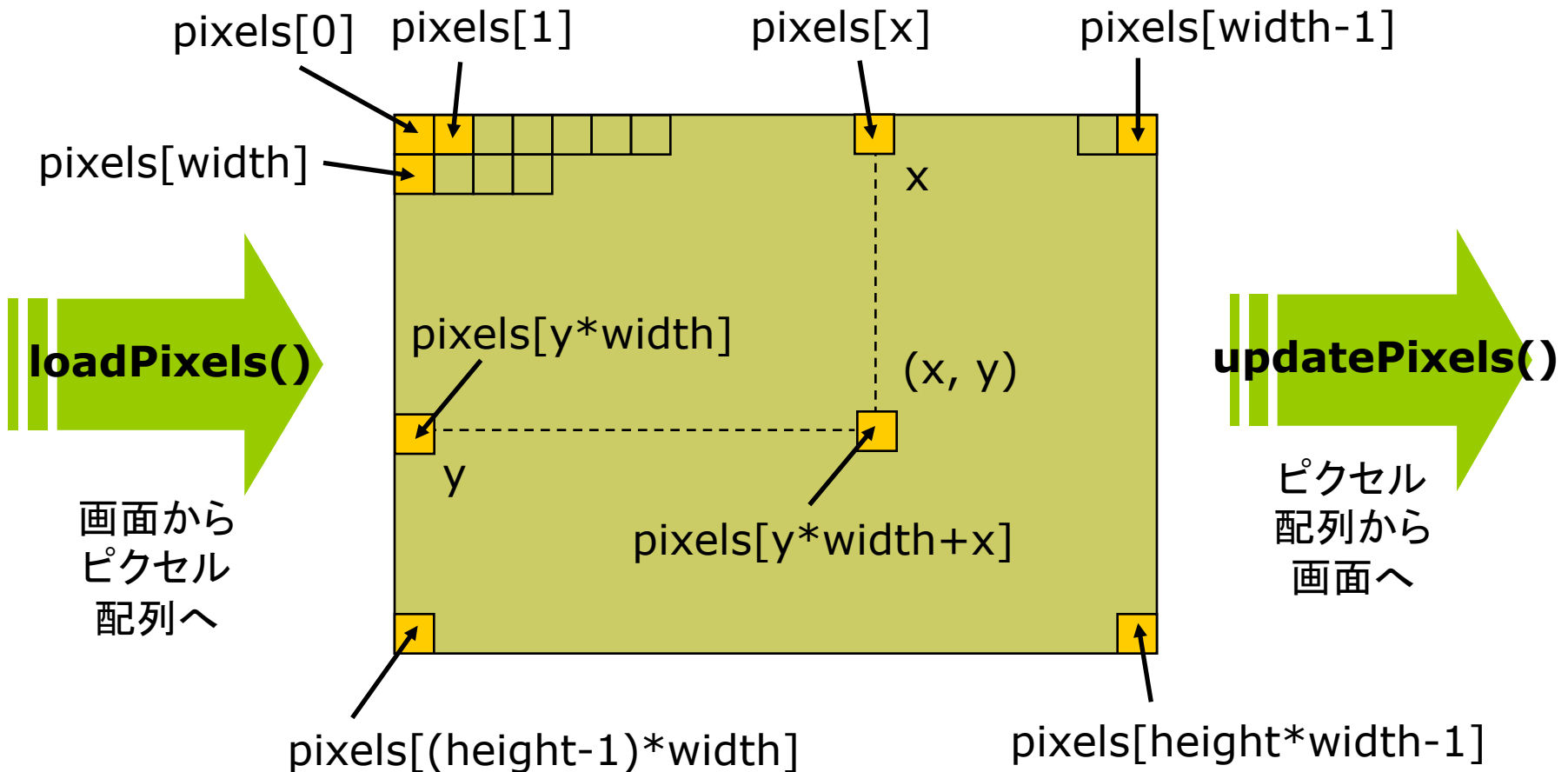
ピクセル配列の操作

- ピクセルの読み出し
 - color c;
 - c = pixels[y * width + x];
- ピクセルの書き込み
 - pixels[y * width + x] = c;

ピクセル配列を使わない操作

- copy(x1, y1, w1, h1, x2, y2, w2, h2)
- copy(画像, x_{画像}, y_{画像}, w_{画像}, h_{画像}, x, y, w, h)
 - 領域や画像からのコピー
- get(), get(x, y, 幅, 高さ)
 - 表示内容を画像として取得

4.4 ピクセル配列

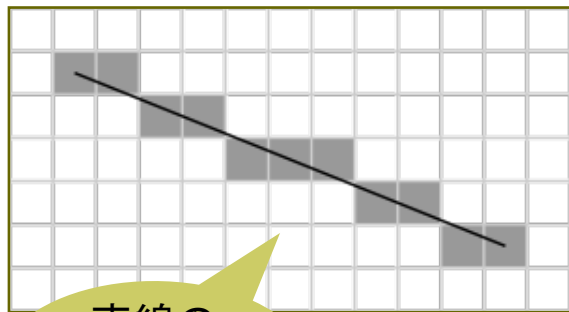


4.5 ラスタライ化

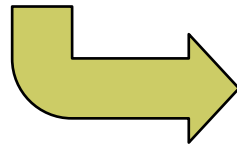
ラスタライ化(p.252)

□ ラスタライ化とは

- ピクセル上への描画処理
- ベクター表現(座標とパラメータ)の図形を画素の集合に変換する



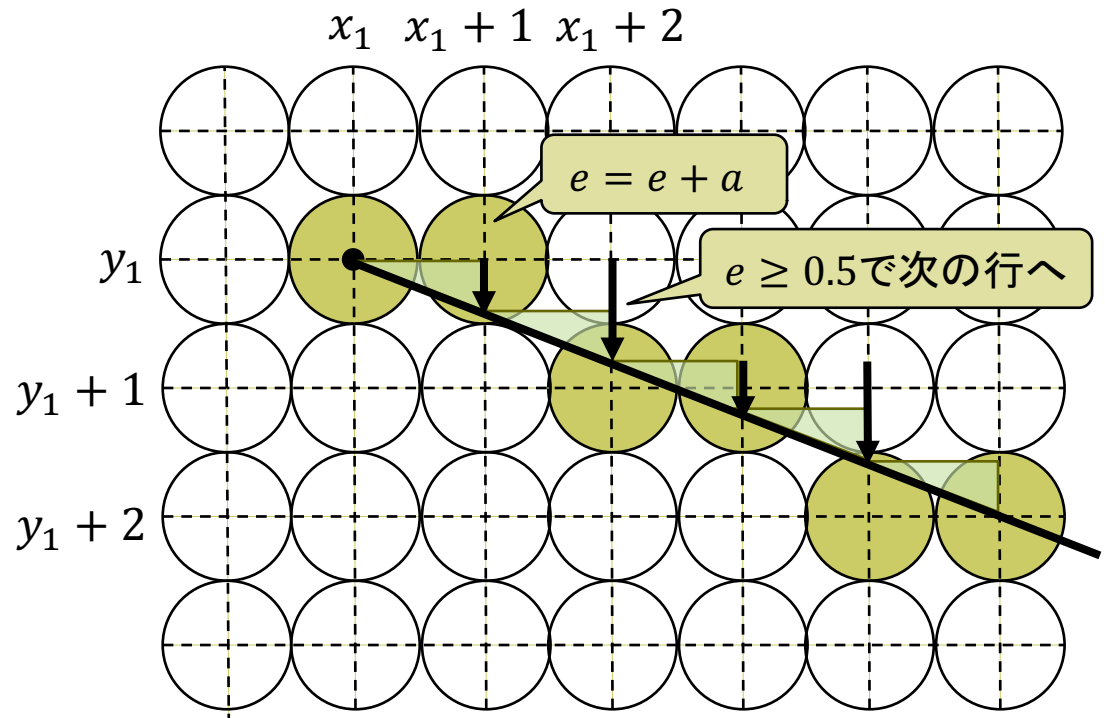
直線の
ラスタライ
化の例



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

□ 直線(線分)のラスタライ化

- x座標(またはy座標)を, 1ずつ変化させながら, 理想の直線に最も近い整数座標のピクセルに色を設定していく



4.6 直線の描画

- ラスタ化のアルゴリズム
 - 直線の傾きで4通りに場合分けして, それぞれ処理する
(この例は $0 \leq \text{傾き} \leq 1$ の処理)
- これをさらに高速化
 - ブレゼンハムのアルゴリズム
 - 式を変形して割り算(と少数)をなくし, すべての演算を整数化
 - 計算の累積誤差も排除される

```
void draw() {
    background(0);
    if (mouseX > mouseY) {
        loadPixels();
        pxline(0, 0, mouseX, mouseY);
        updatePixels();
    }
}
```

```
void pxline(int x1, int y1,
            int x2, int y2)
{
    color c = color(255, 255, 255);

    float a = (float)(y2-y1)/(x2-x1);
    float e = 0.0;

    int x = x1, y = y1;
    while (x <= x2) {
        pixels[y * width + x] = c;

        x++;
        e += a;
        if (e >= 0.5) {
            e -= 1.0;
            y++;
        }
    }
}
```

xが1増えるあたりのyの増分(小数)

本来のy座標との累積誤差

画素設定

xが1増えるごとにyの誤差はa増加

yの誤差が0.5以上になったらyを1増やす

4.7 クリッピング

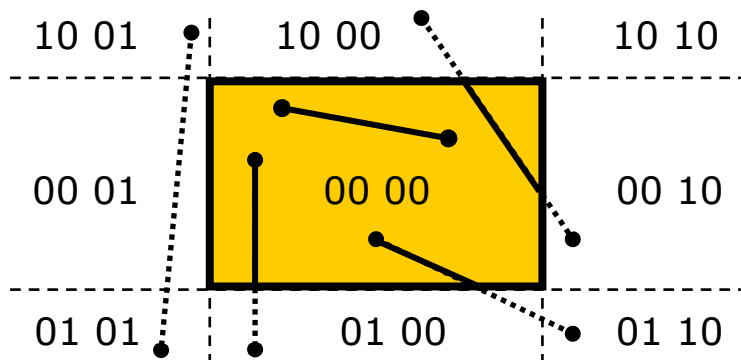
クリッピング(p.52)

□ クリッピングとは

- 描画図形について表示領域(ビューポート)からはみ出した部分は描画しない処理
- 図形の種類ごとに、効率のよい方法が開発されている

□ 線分のクリッピング

- コーエン・サザランドの方法
- ビット演算で直線(線分)が表示領域にかかるか高速に判定



□ アルゴリズム

1. 線分の両端が表示領域の上下右左にはみ出しているかを, 4桁のビットコードで表す
2. 両端点のコードがともに0000なら, 線分の全部が表示領域内にある
3. そうでないなら, 両端点のコードのビットごとの論理積を計算する
例: $1001 \ \& \ 0101 = 0001$
4. 結果が0000以外なら, 線分の全部が表示領域外にある
5. 0000なら, 線分の一部が表示領域にかかっている
6. その場合, ビットコードから線分がどちら側にはみ出しているかが分かるので, 線分と境界線との交点を求め, それを新しい端点として再判定する

4.8 演習課題

課題

- 例のように画像ファイル等からピクセル単位で色を読み出し、色や形を変えて表示させよ
 - 必ずピクセル配列を使うこと
 - カメラ映像を利用したいなら Sketch → Add Library → Libraries → Video で導入

```
import processing.video.*;
Capture v; // キャプチャ画像
```

```
void setup() {
  size(700, 480);
  v = new Capture(this, 80, 60, 30);
  v.start();
  colorMode(HSB, 1.0); // HSBモード
  frameRate(30);
}
```

```
void draw() {
  // キャプチャできない状態なら何もしない
  if (!v.available()) return;
  background(0);
  v.read();          // 1フレームキャプチャ
  v.loadPixels();    // ピクセル配列の設定

  for (int y = 0; y < v.height; y++) {
    for (int x = 0; x < v.width; x++) {
      // ピクセル配列から色を読み出す
      color c = v.pixels[y * v.width + x];

      // 色相・彩度・明度を取得する
      float h = hue(c);
      float s = saturation(c);
      float b = brightness(c);

      // ピクセルの色や形を変形して表示
      fill(h, s, 1.0); noStroke();
      float d = 8 * sqrt(b);
      rect(x * 8 + y, y * 8, d, d);
    }
  }
}
```