画像処理及び演習
～総復習～

塚田・森本・澤野

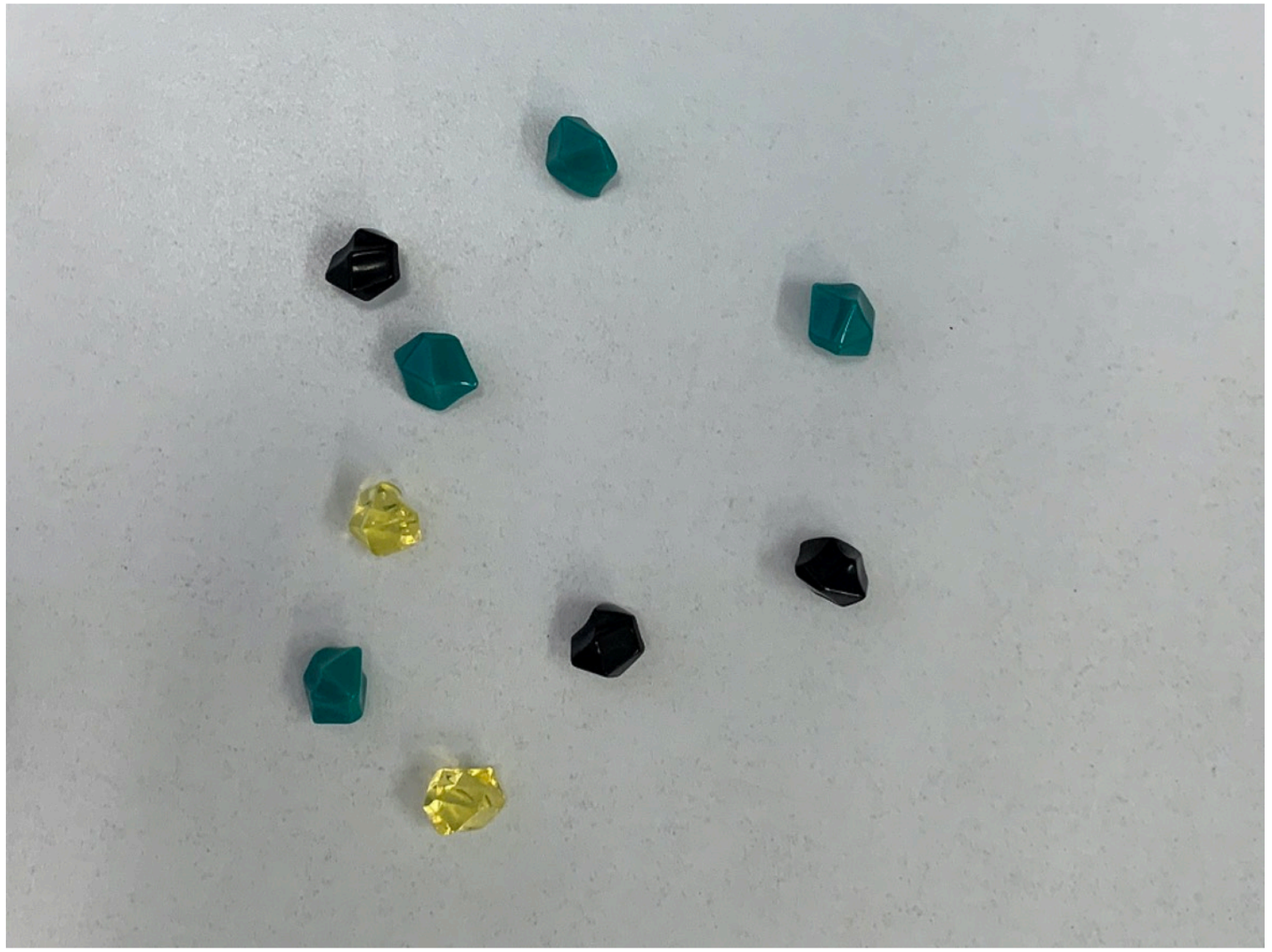
目次

- 第14回：これまでの手法を用いた演習（総復習）
    1. エッジ検出
    2. ヒストグラム
    3. 二値化
    4. カラー画像処理
    5. HSV色変換
    6. 膨張収縮処理
    7. 輪郭追跡

✦課題

対象画像と目標

- ボードゲーム「インカの黄金」
  - 宝石（gem）
    - ✓ トルコ石（緑色）
    - ✓ 黒曜石（黒色）
    - ✓ 黄金
      （半透明の黄色）



入力画像 gem1.jpg

➡それぞれの個数を求める

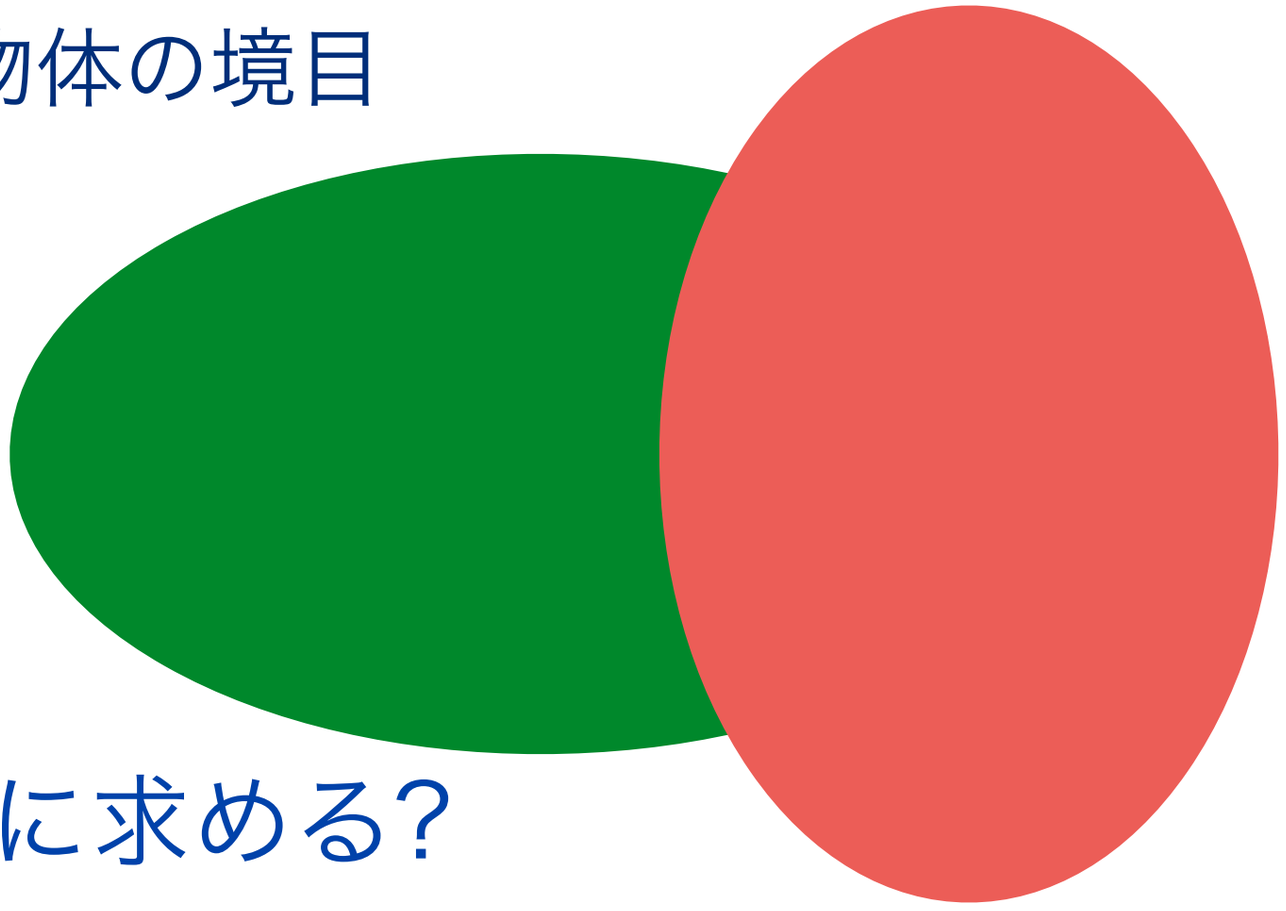
➡まずは画像内からそれぞれを検出

復習

エッジ検出

(教科書p.47)

- 領域と領域の境界を求めたい
  - 物体と背景、物体と物体の境目



- 画像処理でどのように求める?
  - エッジ：輝度値が急激に変化する境目
    - ✓ 微分フィルタ、プリューウィト、ソーベル、ラプラシアン

ノート

Cannyオペレータ

(教科書p.56)

- 現在エッジ検出でよく用いられる手法

1. ガウシアンフィルタによる平滑化
2. 一次微分により勾配を求める
3. 勾配の最大値を検出し細線化
4. ２つのしきい値を用いてエッジを判定

メモ

Cannyオペレータ関数

```
cv::Canny (gray_img, edge_img, double th1, double th2);
```

- gray_img: 入力画像　edge_img: 出力画像
- th1, th2: 勾配の大きさに対するしきい値
  （th1 < th2とする）

✓ th2: エッジ初期検出のためのしきい値

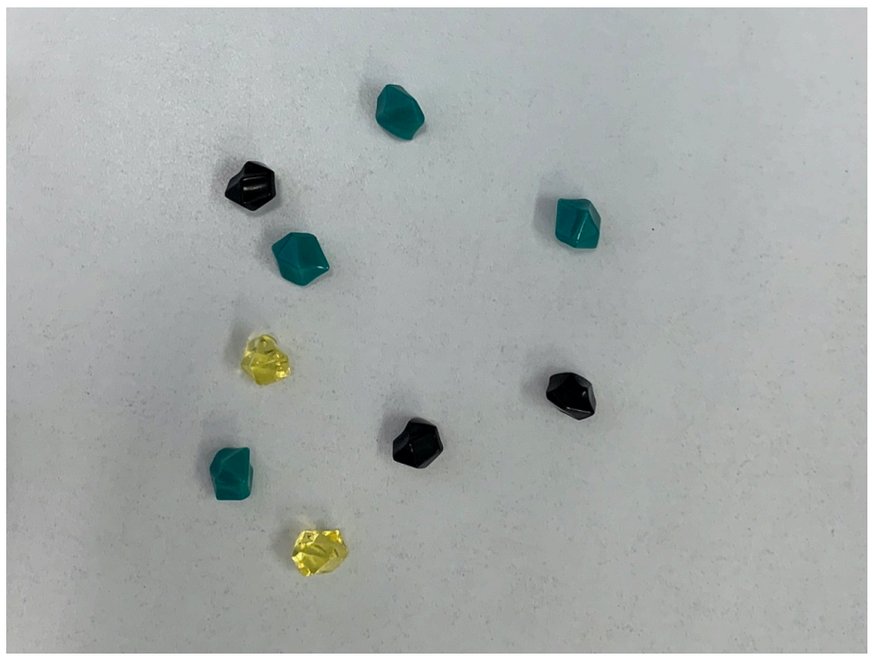
✓ th1: このしきい値以上で、エッジに接続されている画素はエッジと見なす

演習: Cannyオペレータ

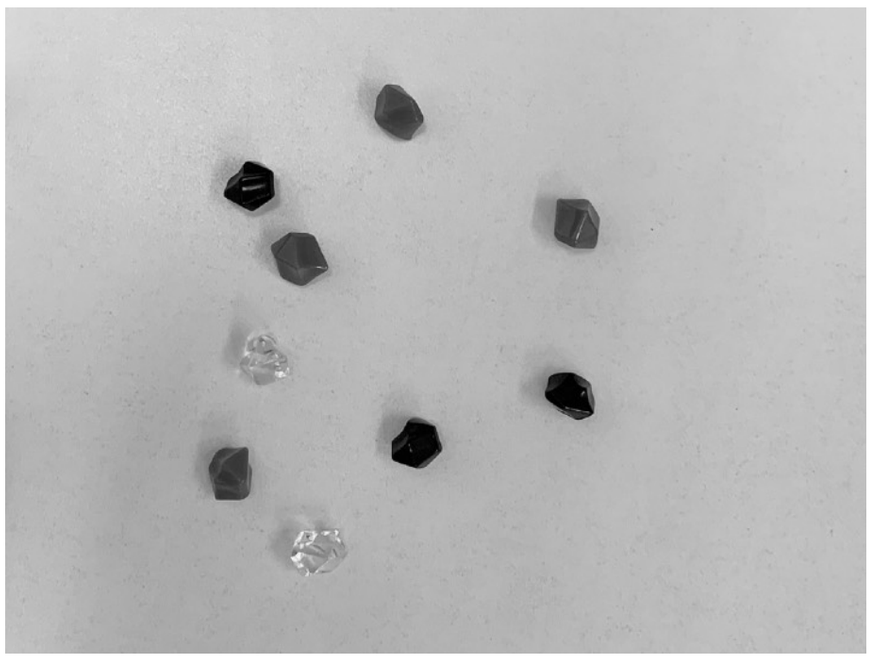
- プロジェクト名: gem_edge
- 入力画像からエッジ検出を行う
  - 入力画像: gem1.jpg
  - OpenCVのcannyオペレータを利用
    - ✓ しきい値：th1=100, th2=200
      （通常th2はth1の2〜3倍）

処理の流れ

1. 入力画像をカラー画像として入力
2. グレースケール画像に変換
3. **エッジ検出（cannyオペレータ）**
4. 表示



入力画像 gem1.jpg



グレースケール画像

エッジ
検出

プログラムのポイント

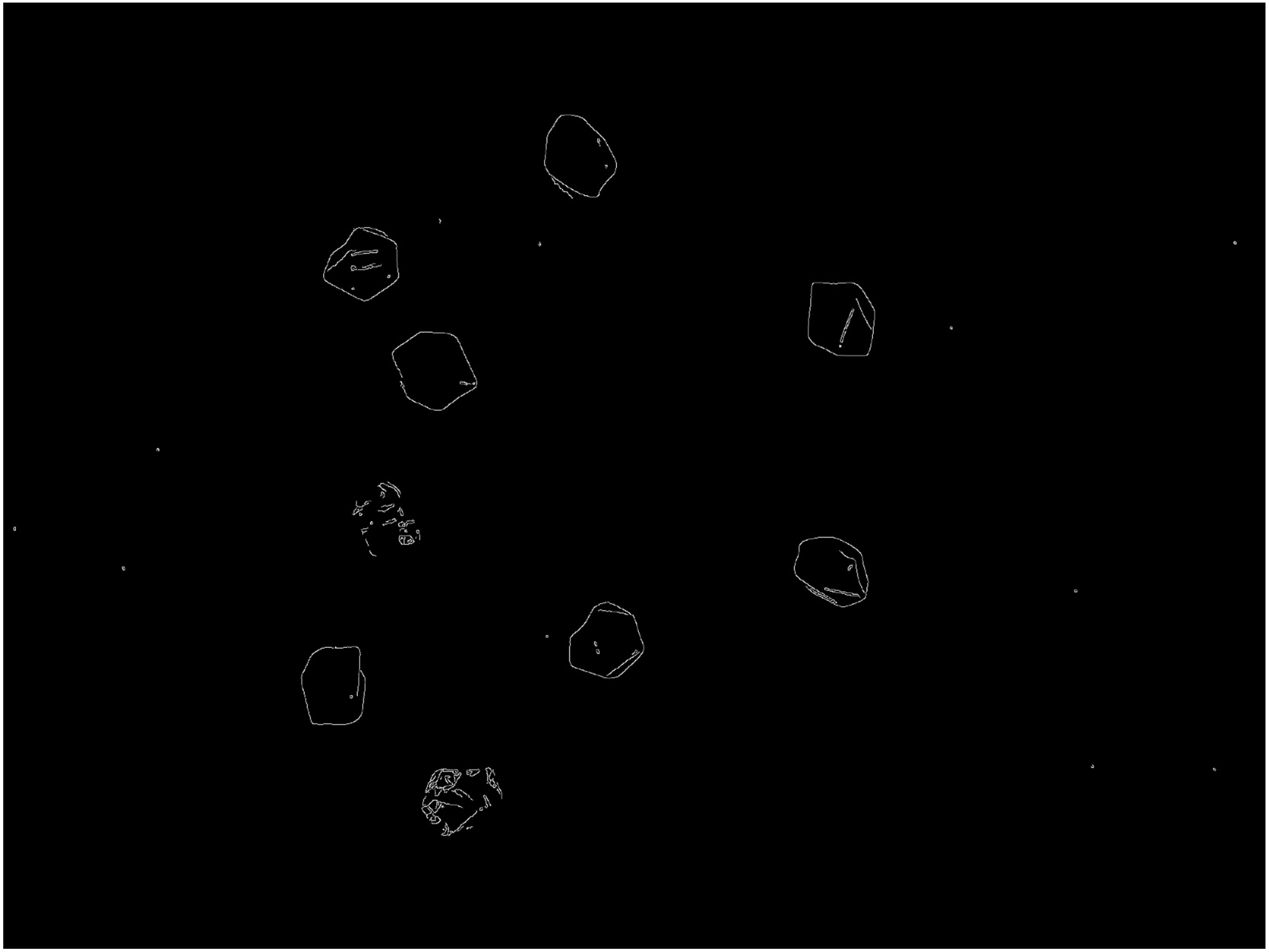
```
//画像変数の宣言
cv::Mat src_img, gray_img, dst_img;

//1. 入力画像をカラーで入力
src_img = cv::imread(FILE_NAME, cv::IMREAD_COLOR);
if (src_img.empty()) { //入力失敗の場合
    fprintf(stderr, "Cannot read image file: %s.\n", FILE_NAME);
    return (-1);
}

//2. グレースケール画像
cv::cvtColor(src_img, gray_img, cv::COLOR_BGR2GRAY);

//3. エッジ検出（cannyオペレータ）
cv::Canny(gray_img, dst_img, 100, 200);
```

カラー画像（省略可能）

濃淡変換を指示

今回表示のための加工（convertScaleAbs）は不要
（cannyオペレータ内で加工している）

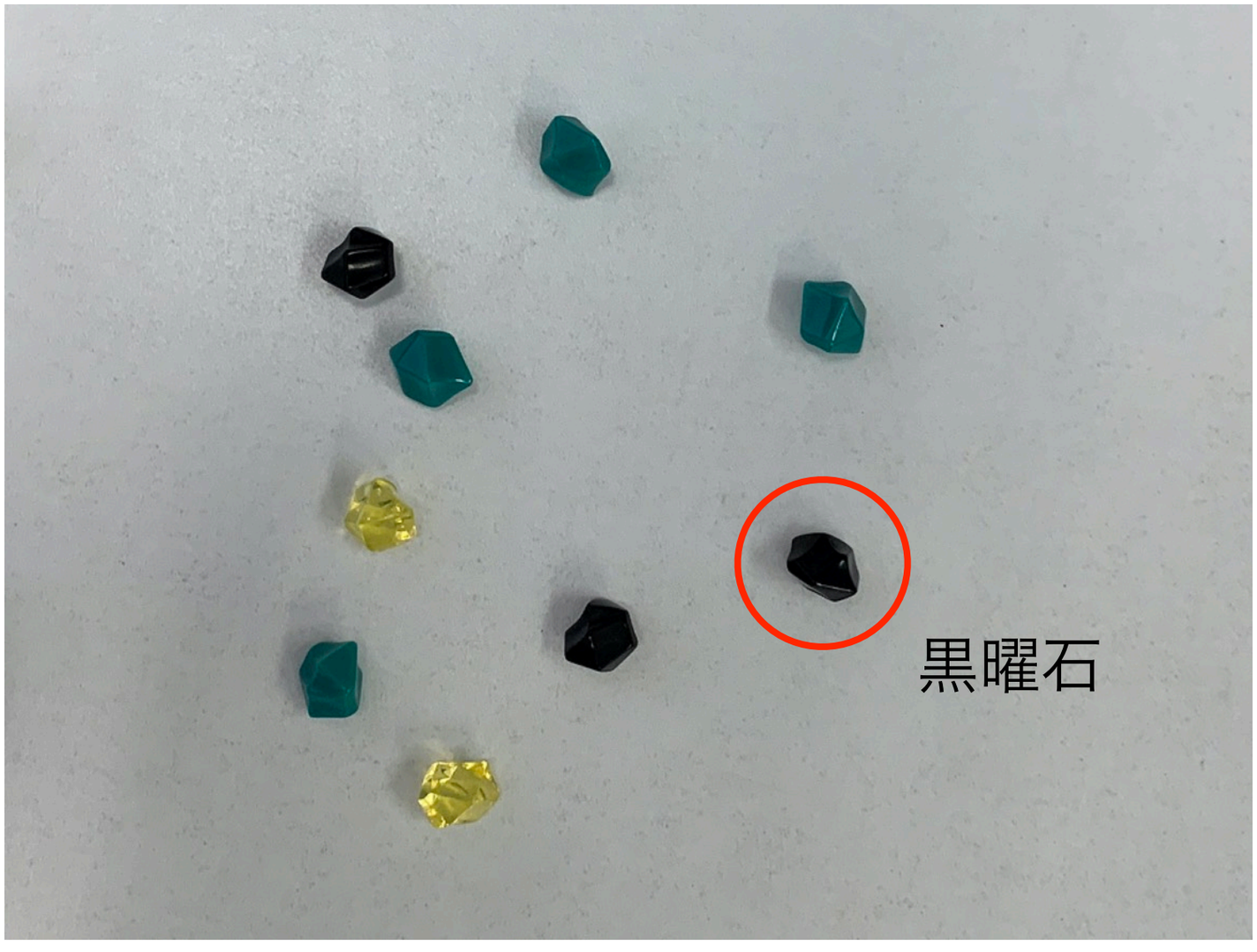
処理結果



しきい値を変更してみよう

輝度値による宝石検出

- エッジだと黄金検出は難しそう
- 輝度値（グレースケール画像）は？
  - 黒曜石（黒色）は取り出せるかも　→　要分析

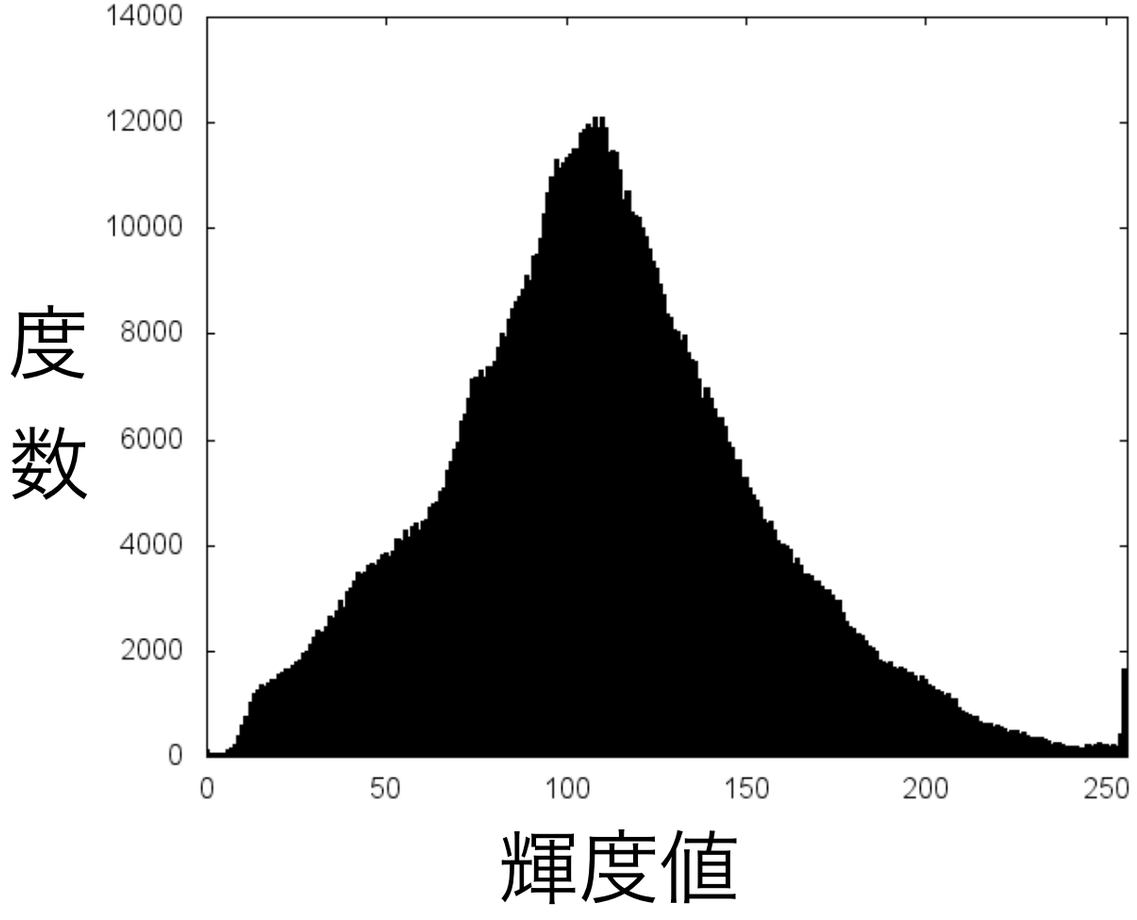


復習

ヒストグラムとは

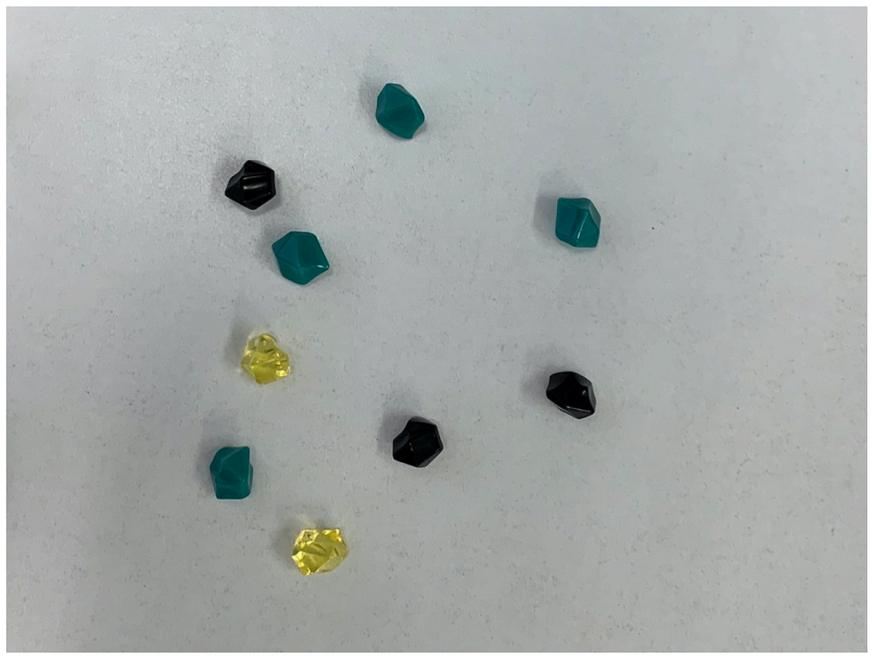
(教科書p.22)

- 画像特徴を可視化する手段
- 輝度ヒストグラム：画像の輝度値分布を表現
  - 縦軸に度数，横軸に輝度値

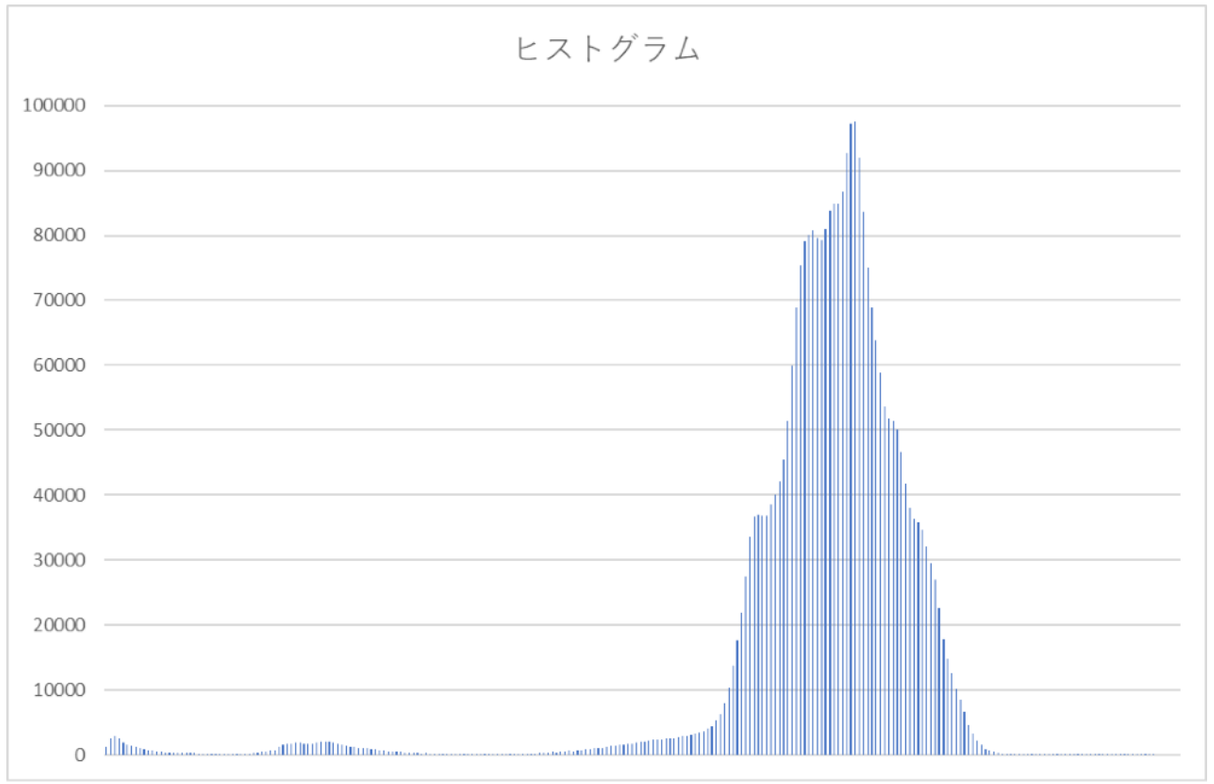


演習: ヒストグラム作成

- プロジェクト名: gem_hist
- 入力画像の輝度値からヒストグラムを作成
  - 入力画像: gem1.jpg



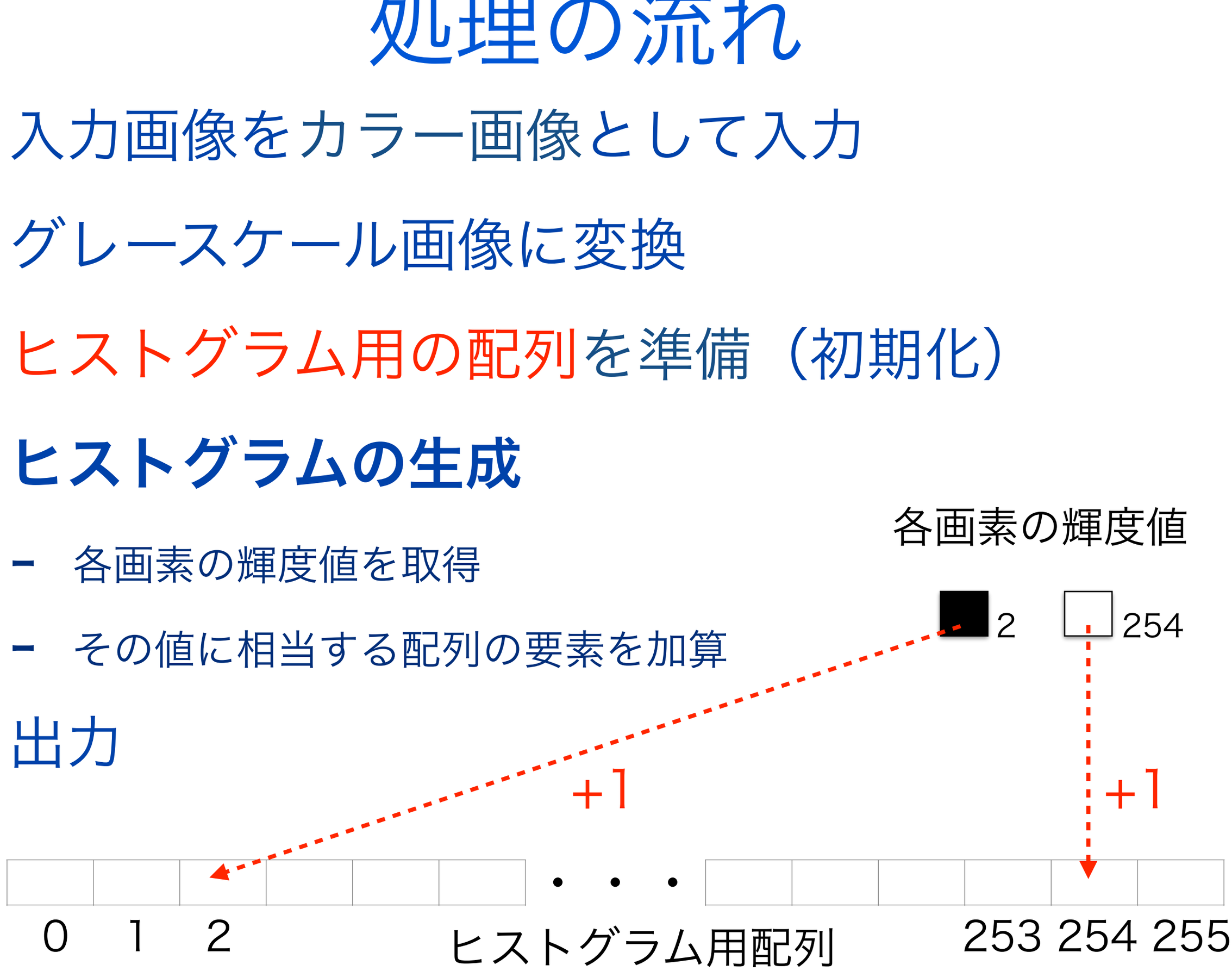
入力画像 gem1.jpg



輝度値

処理の流れ

1. 入力画像をカラー画像として入力
2. グレースケール画像に変換
3. ヒストグラム用の配列を準備（初期化）
4. **ヒストグラムの生成**
   - 各画素の輝度値を取得
   - その値に相当する配列の要素を加算
5. 出力



プログラムのポイント

```
#define COLOR_MAX (256)
        中略
    int hist[COLOR_MAX];
        中略
    //3. ヒストグラム用配列の初期化
    for (i=0; i<COLOR_MAX; i++) {
        hist[i] = 0;
    }
    //4. ヒストグラムの生成
    for (y=0; y<gray_img.rows; y++) {
        for (x=0; x<gray_img.cols; x++) {
            uchar s = gray_img.at<uchar>(y,x);  //画素値の取得
            hist[(int)s]++; //画素値に相当する配列の要素を加算
        }
    }
    //5. コンソールに出力
    for (i=0; i<COLOR_MAX; i++) {
        printf("%d\n", hist[i]);
    }
```

ヒストグラム用配列（サイズは輝度値の範囲）

グレースケール画像の画素は
uchar（もしくはunsigned char）

出力からヒストグラムを作成

```
27
38
40
38
35
24
31
42
40
40
35
42
27
29
20
12
3
0
0
0
0
0
0
Program ended with exit code: 0
```

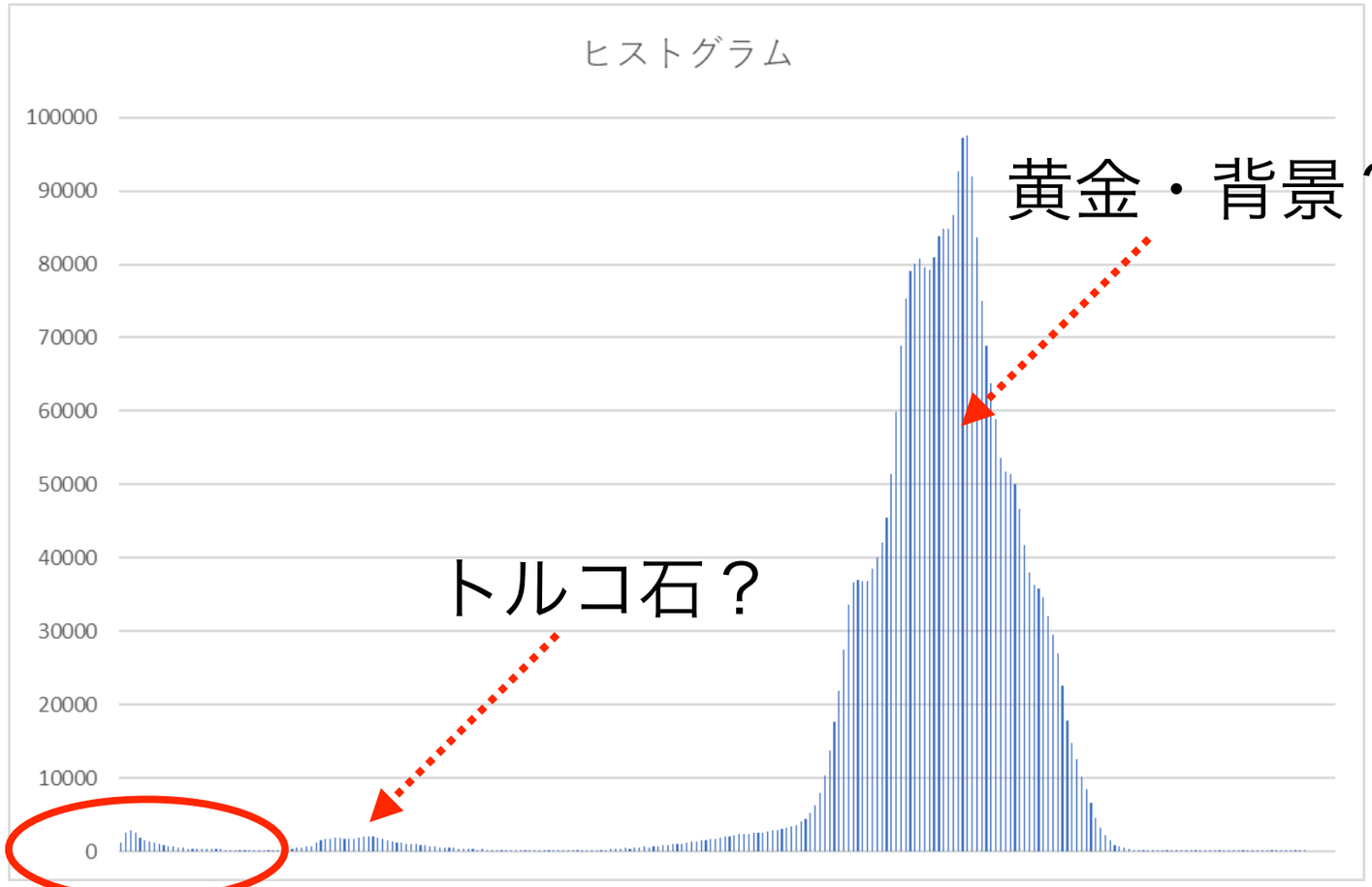
コンソールの出力（256行）

コピー

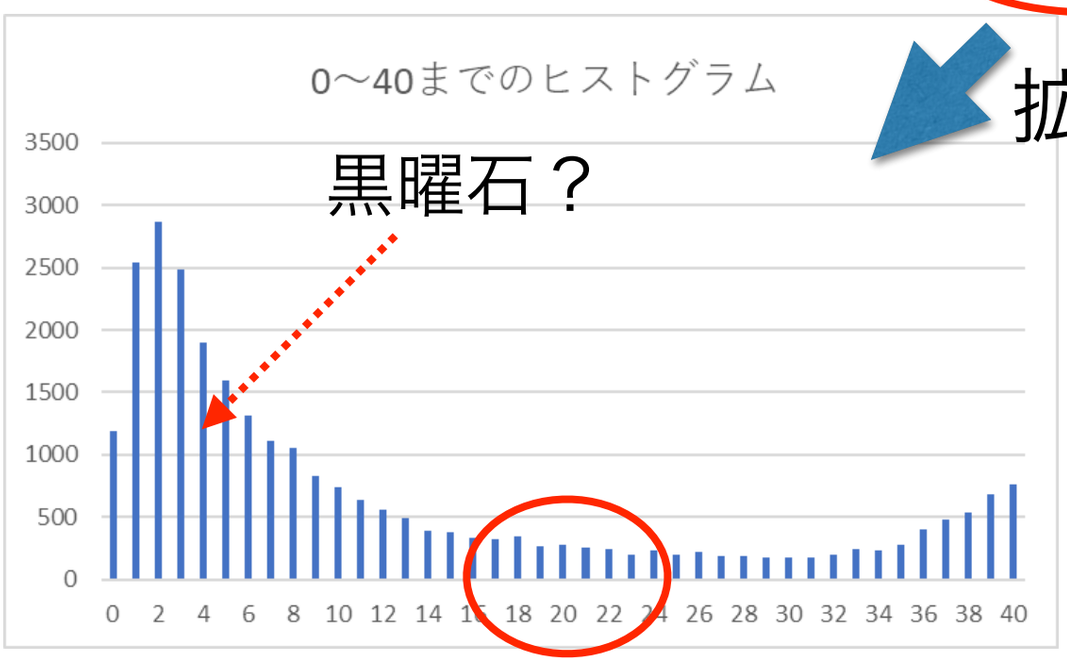
	A	B
1	1186	
2	2545	
3	2862	
4	2481	
5	1895	
6	1594	
7	1312	
8	1116	
9	1058	
10	834	
11	735	
12	635	
13	564	
14	497	
15	395	
16	383	
17	335	

excelやnumbersへ貼り付け
グラフ機能を用いる

ヒストグラム結果



拡大



輝度値20あたりをしきい値にすれば
黒曜石が検出できそう

復習

二値化処理

(教科書p.60)

・画像の画素値を二つの値だけにする処理

- 各画素を明るい画素と暗い画素のどちらかに分類

➡白（255）もしくは黒（0）に変換

カラー画像　濃淡画像　二値画像

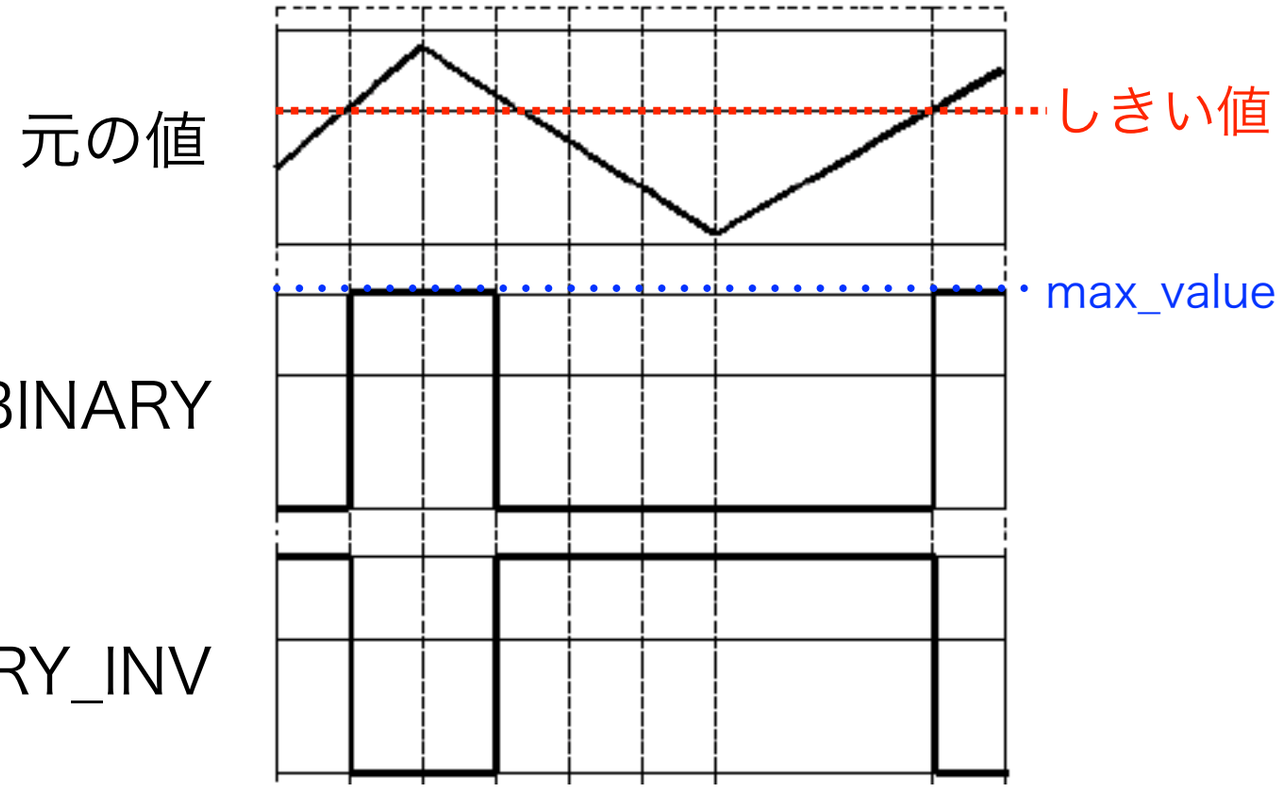
復習

しきい値処理関数

- 関数紹介

```
cv::threshold(入力画像, 出力画像, しきい値, max_value, オプション);
```

  - オプション
    - ✓ cv::THRESH_BINARY: しきい値以上をmax_valueに設定
    - ✓ cv::THRESH_BINARY_INV：しきい値以下を max_valueに設定

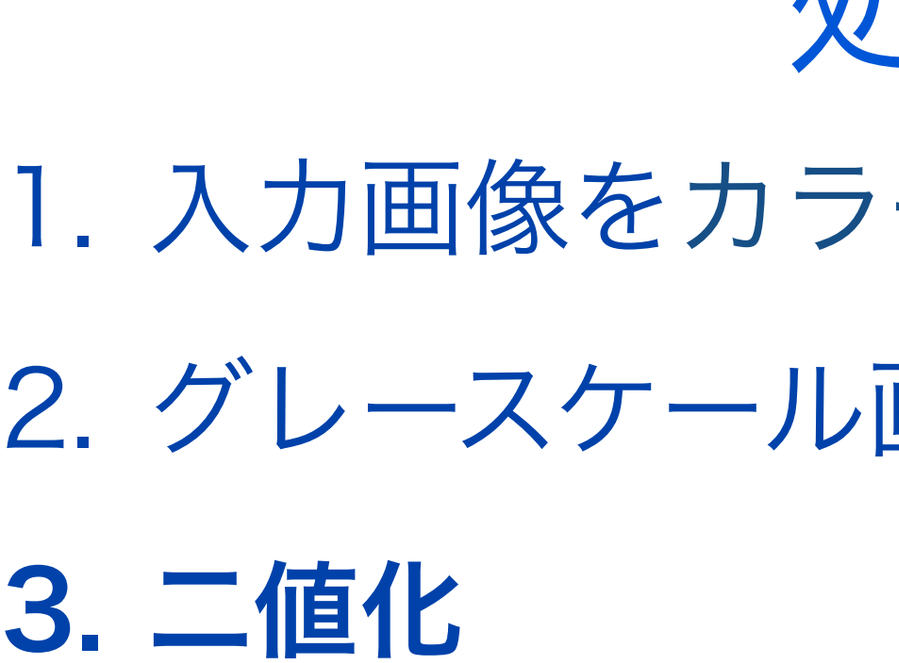


演習: 二値化による黒曜石検出

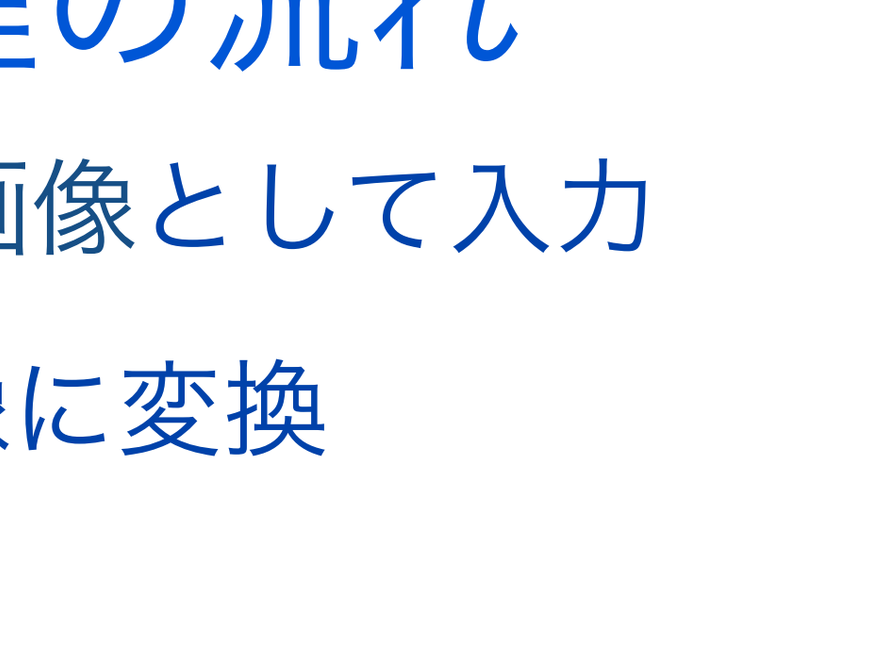
- プロジェクト名: gem_binarize
- 入力画像から二値化を行う
  - 入力画像: gem1.jpg
  - OpenCVのthresholdオペレータを利用
    - ✓ しきい値：TH=20　（ヒストグラムでの分析より）
    - ✓ THRESH_BINARY_INVを用いて、しきい値以下の領域を MAX_VAL（白画素）にする
      （黒曜石だけを白画素にする）

処理の流れ

1. 入力画像をカラー画像として入力
2. グレースケール画像に変換
3. **二値化**
4. 表示



入力画像 gem1.jpg



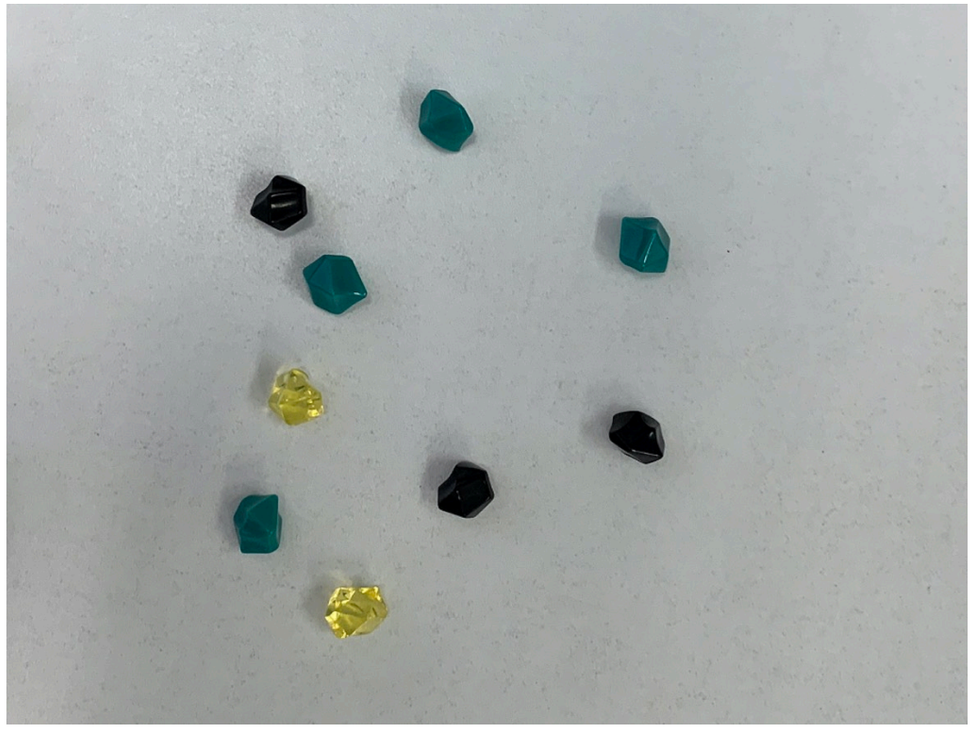
グレースケール画像

二値化

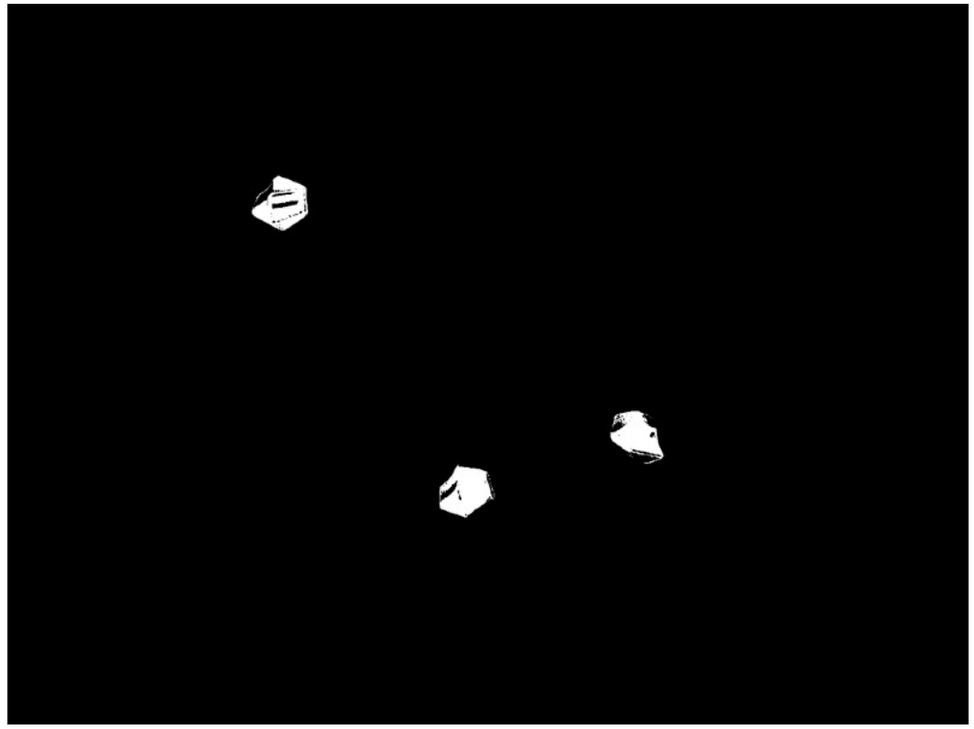
プログラムのポイント

```
#define TH (20)
#define MAX_VAL (255)

                中略

    cv::Mat gray_img, dst_img;

    //1. 入力画像をカラーで入力
    cv::Mat src_img = cv::imread(FILE_NAME, cv::IMREAD_COLOR);
    if (src_img.empty()) { //入力失敗の場合
        fprintf(stderr, "Cannot read image file: %s.\n", FILE_NAME);
        return (-1);
    }

    //2. グレースケール画像
    cv::cvtColor(src_img, gray_img, cv::COLOR_BGR2GRAY);

    //3. 二値化
    cv::threshold(gray_img, dst_img, TH, MAX_VAL, cv::THRESH_BINARY_INV);
```

処理結果



入力画像 gem1.jpg



二値画像

黒曜石は全て検出できている

石中の一部が欠けている

しきい値THを変更するとできるかも？

色情報による宝石検出

- トルコ石（緑色）はどうする？
  - 輝度値でできるかも知れないが…
  - 「緑」色で取り出せないか？　→ 要分析

メモ

カラー画像のチャンネルを分割

```
cv::Mat planes[3];
cv::split(src_img, planes);
cv::imshow("Blue", planes[0]);
cv::imshow("Green", planes[1]);
cv::imshow("Red", planes[2]);
```

- split（入力画像, 出力画像配列）
  - チャンネル毎の画像に分割する
    →各チャンネルの傾向を見ることができる
    （ヒストグラムも作成可能）

Gチャンネルの画像

- 黒曜石よりは少し明るいかな、という程度
  →他の条件も加えてみる

演習: RGBによるトルコ石検出

- プロジェクト名: gem_color
- 入力画像からRGB値による領域検出を行う
  - 入力画像: gem1.jpg
  - 各画素（RGB値）に対して条件を適用
    - ✓ Gチャンネル：50〜100
    - ✓ 補足条件　Rチャンネル：0〜50
  - 二値画像を出力

処理の流れ

1. 入力画像をカラー画像として入力
2. 出力二値画像の領域を確保（初期値ゼロ）
3. **しきい値処理**
   - 各画素のRGB値を取得
   - しきい値条件が適合したら対応する二値画像の画素位置に白画素を書き込む
4. 表示

プログラムのポイント

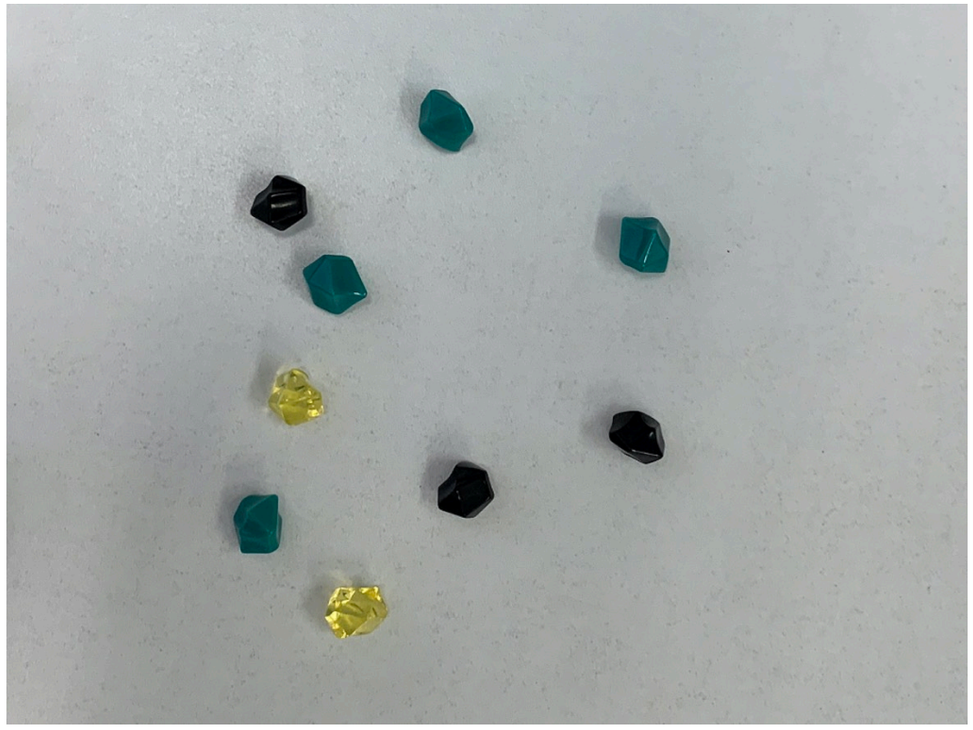
```
#define R_MIN (0) //Rの最小値
#define R_MAX (50) //Rの最大値
#define G_MIN (50) //Gの最小値
#define G_MAX (100) //Gの最大値

                    中略

    cv::Vec3b p; //色値

                    中略

    // 2. 出力二値画像の領域を確保（初期値ゼロ：黒）
    dst_img = cv::Mat::zeros(src_img.size(), CV_8UC1);

    // 3. しきい値処理
    for (y=0; y<src_img.rows; y++) {
        for (x=0; x<src_img.cols; x++) {
            p = src_img.at<cv::Vec3b>(y, x); //各画素のBGR値を取得
            // RGB色空間で緑色を検出
            if (p[1] >= G_MIN && p[1] <= G_MAX &&
                p[2] >= R_MIN && p[2] <=R_MAX) {
                dst_img.at<uchar>(y,x) = MAX_VAL; // しきい値条件が適合したら白画素
            }
        }
    }
```

入力画像と同じサイズ
１チャンネル、初期値０

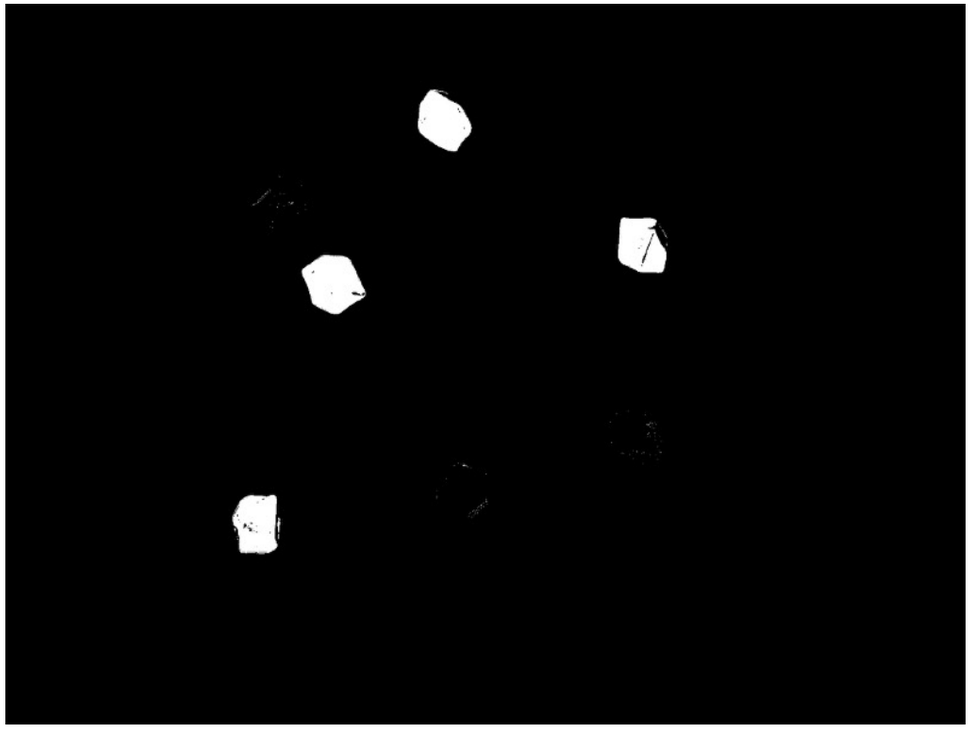
カラー画像の画素はVec3b

出力画像の画素はuchar

処理結果



入力画像 gem1.jpg



検出画像

トルコ石は全て検出できている

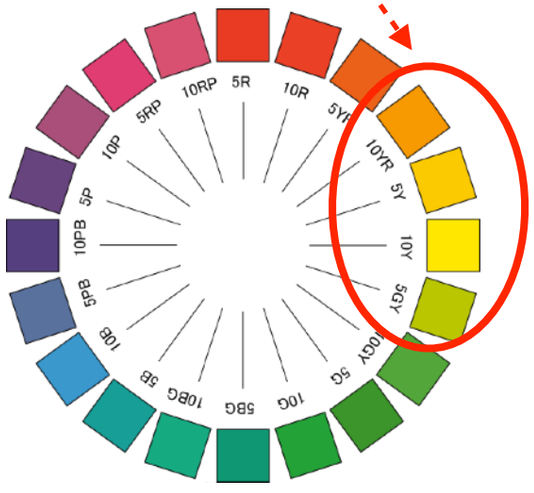
石中の一部が欠けている

こちらも試行錯誤するともっとできるかも？

黄金検出

- 黄金（半透明の黄色）はどうする？
  - 輝度値やRGBでは難しそう…
  - HSV色空間でできないか？

この辺



マンセルの
色相環

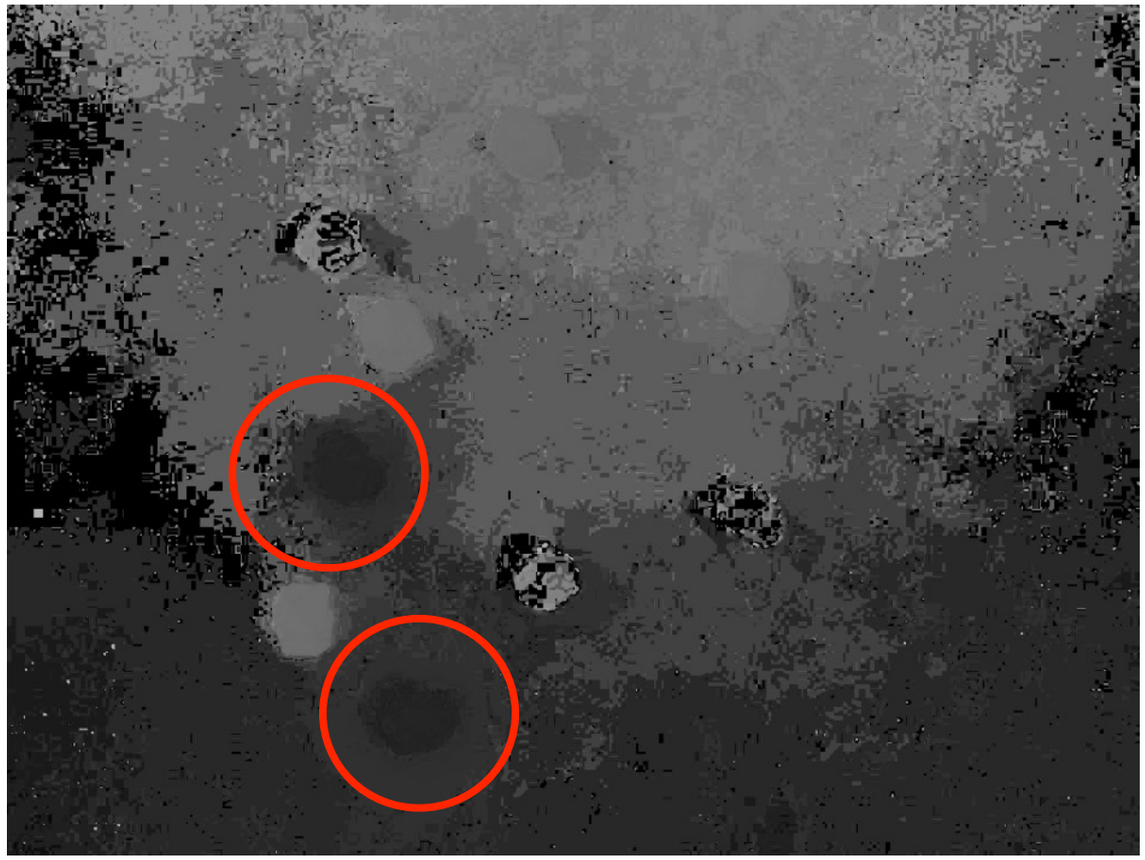
HSV色変換関数

```
//HSV変換 (RGB => HSV)
cv::cvtColor(src_img, dst_img, cv::COLOR_BGR2HSV);
```

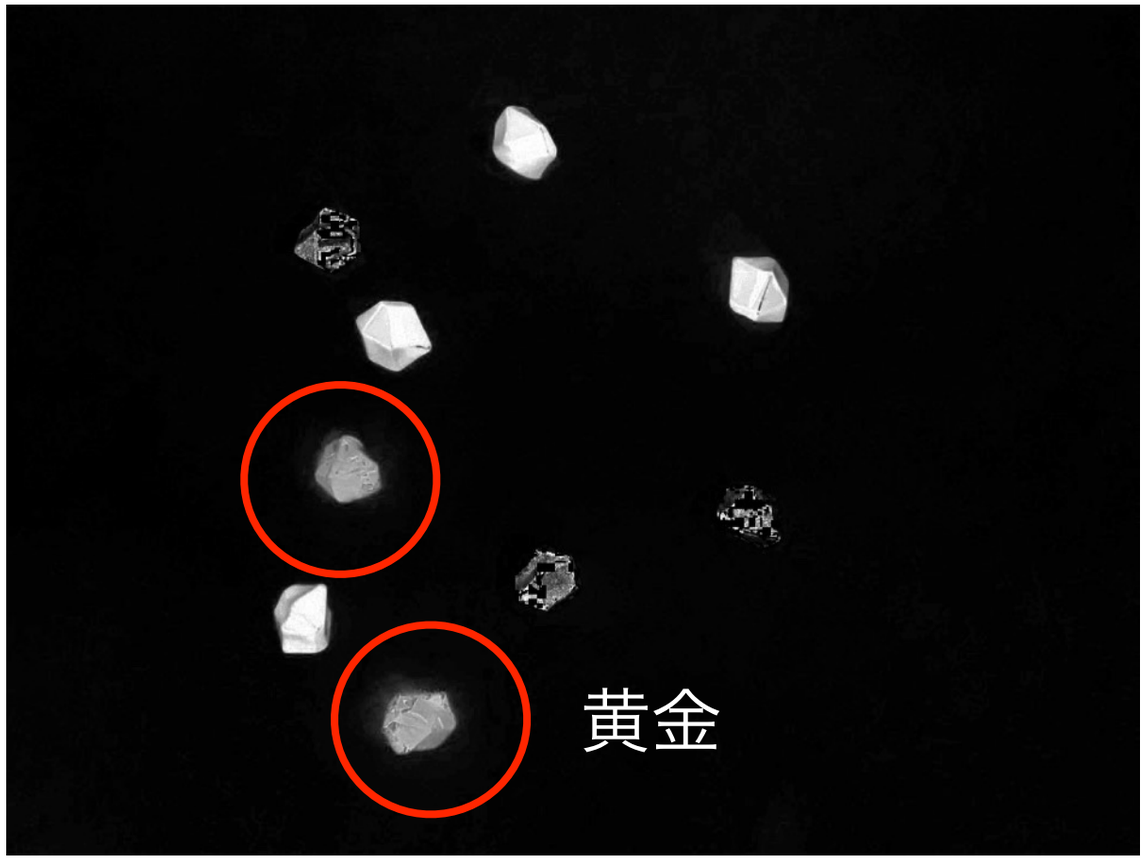
- 引数: 入力画像, 出力画像, コード
- コード (変換命令)
  - **cv::COLOR_BGR2HSV**
    - **◉ HSV変換　RGB→HSV**
  - **cv::COLOR_HSV2BGR**
    - **◉ 逆HSV変換　HSV→RGB**

(教科書p.28)

色相(H)／彩度(S)の画像



色相画像



彩度画像

- 色相だけでは難しそう（低い値が多い）
- 彩度と一緒になら使えそう

演習: HSVによる黄金検出

- プロジェクト名: gem_hsv
- 入力画像からHSV値による領域検出を行う
  - 入力画像: gem1.jpg
  - 各画素のHSV値に対して条件を適用
    - ✓ Hue：20～60 （色相の値域は0から179）
    - ✓ Saturation：100～200 （彩度の値域は0から255）
  - 二値画像を出力

処理の流れ

1. 入力画像をカラー画像として入力
2. 出力二値画像の領域を確保（初期値ゼロ）
3. **色変換(RGB→HSV)**
4. しきい値処理
   - 各画素のHSV値を取得
   - しきい値条件が適合したら対応する二値画像の画素位置に白画素を書き込む
5. 表示

プログラムのポイント

```
#define HUE_MIN (20.0) // Hの最小値
#define HUE_MAX (60.0) // Hの最大値
#define SAT_MIN (100.0) // Sの最小値
#define SAT_MAX (200.0) // Sの最大値
```

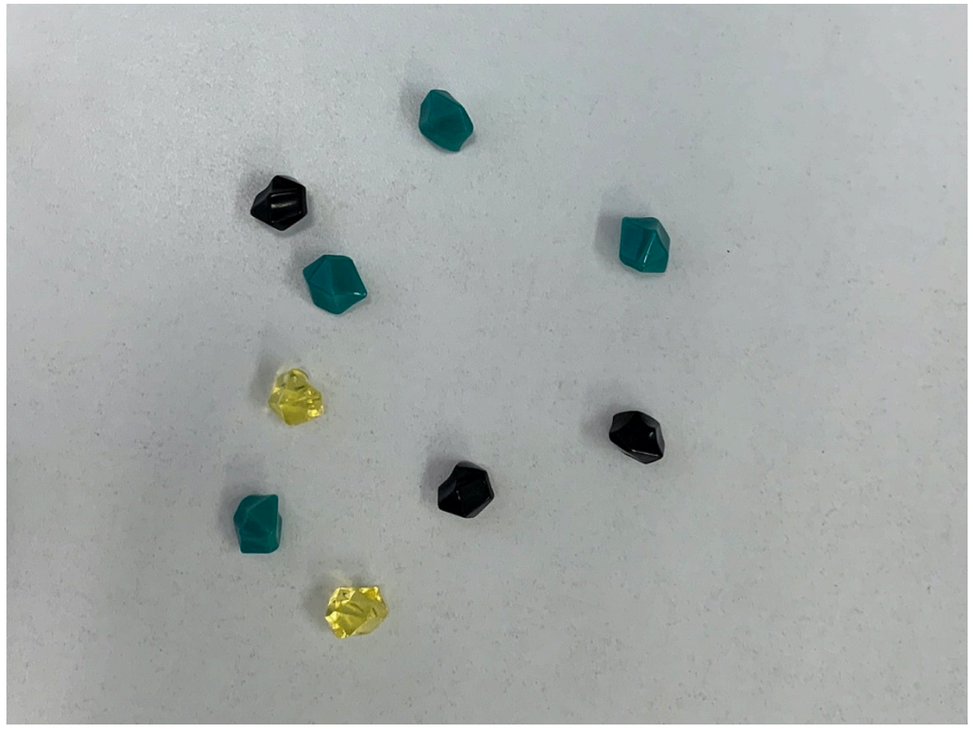
中略

HSV色空間に変換する

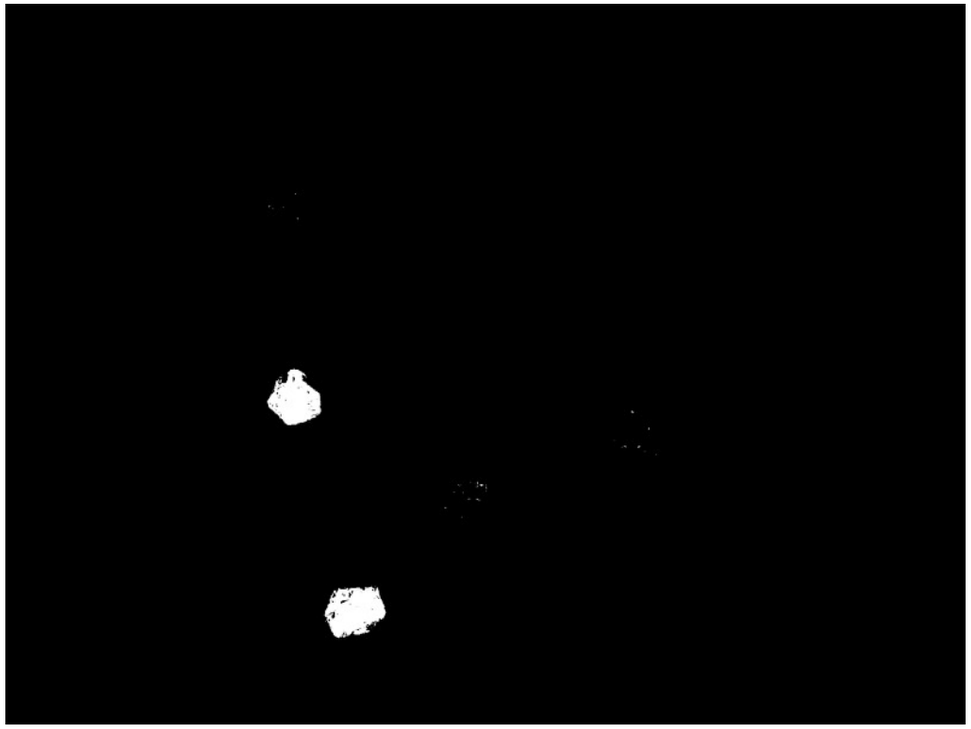
```
    // 3. 色変換 (BGR → HSV)
    cv::cvtColor(src_img, hsv_img, cv::COLOR_BGR2HSV);

    // 4. しきい値処理
    for (y=0; y<hsv_img.rows; y++) {
        for (x=0; x<hsv_img.cols; x++) {
            p = hsv_img.at<cv::Vec3b>(y, x); // 各画素のHSV値を取得
            // HSV色空間で色を検出
            if (p[0] >= HUE_MIN && p[0] <=HUE_MAX &&
                                   p[1] >= SAT_MIN && p[1] <=SAT_MAX) {
                dst_img.at<uchar>(y,x) = MAX_VAL; // しきい値条件が適合したら白画素
            }
        }
```

gem_colorとやり方は一緒

処理結果



入力画像 gem1.jpg



検出画像

黄金は全て検出できている

石中の一部が欠けている

Sだけでもできる？Vも入れたほうが良い？

石を数えよう

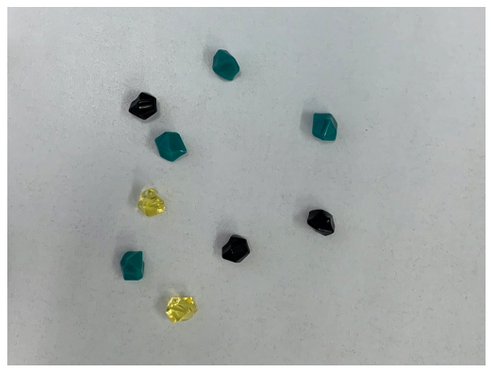
- 処理方法
  - それぞれの石が検出できた（二値画像）
  - その二値画像から輪郭追跡により領域を求める

    ➡ 輪郭数＝領域数＝石の個数
  - 懸案

    ✓ 余計な輪郭を取り出さないか

    ✓ 石ひとつに複数の輪郭ができないか

    ➡ 膨張収縮処理



入力画像 gem1.jpg

黒曜石の二値画像

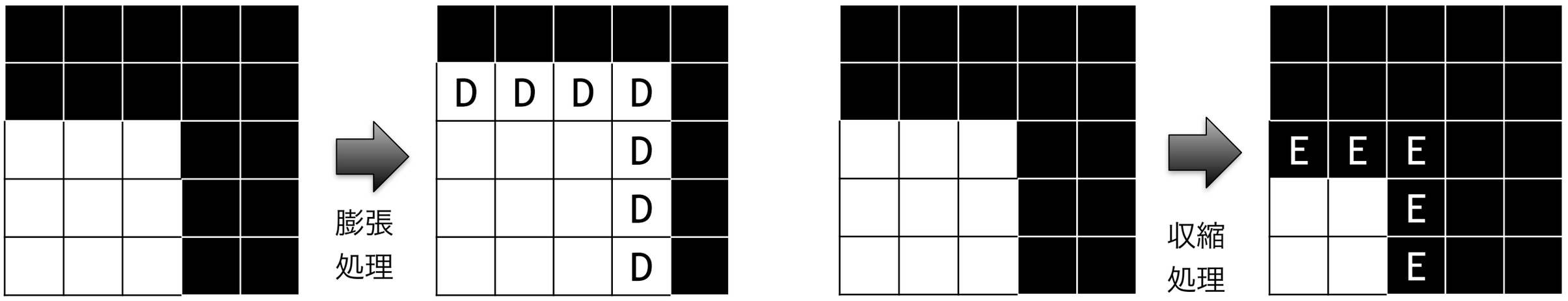
石の個数

復習

膨張収縮処理

(教科書p.65)

- 膨張処理: 背景または穴に接する対象の画素に，白画素を一回り加える処理
  - 白画素領域を膨張
- 収縮処理: 一回り削る処理
  - 白画素領域を収縮　（＝黒画素領域を膨張）



膨張収縮処理関数

- 膨張処理関数

```
cv::dilate(bin_img, dst_img, cv::Mat(), cv::Point(-1, -1), 回数);
```

- 収縮処理関数

```
cv::erode(bin_img, dst_img, cv::Mat(), cv::Point(-1, -1), 回数);
```

  - bin_img：入力画像　dst_img：出力画像
  - 回数：何回膨張収縮処理を繰り返すか

復習 クロージングとオープニング

- クロージング
  - 同じ回数だけ膨張して収縮
  - 穴を取り除く
- オープニング
  - 同じ回数だけ収縮して膨張
  - ノイズを取り除く
- 膨張・収縮の回数は、どの程度の大きさの穴とノイズを削除したいかに応じて決める

演習: 黒曜石を数える　その１

- プロジェクト名: gem_count_black
- 入力画像から黒曜石を検出し膨張収縮処理を行う
  - 入力画像: gem1.jpg
  - 前半はgem_binarizeと同じ
  - 後半でクロージング・オープニングを行う
    - ✓ 回数：COUNT=15
    - ✓ まずクロージングで穴を埋めて、次にオープニングでノイズ除去

処理の流れ

1. 入力画像をカラー画像として入力
2. グレースケール画像に変換
3. 二値化
4. クロージング・オープニング
5. 表示

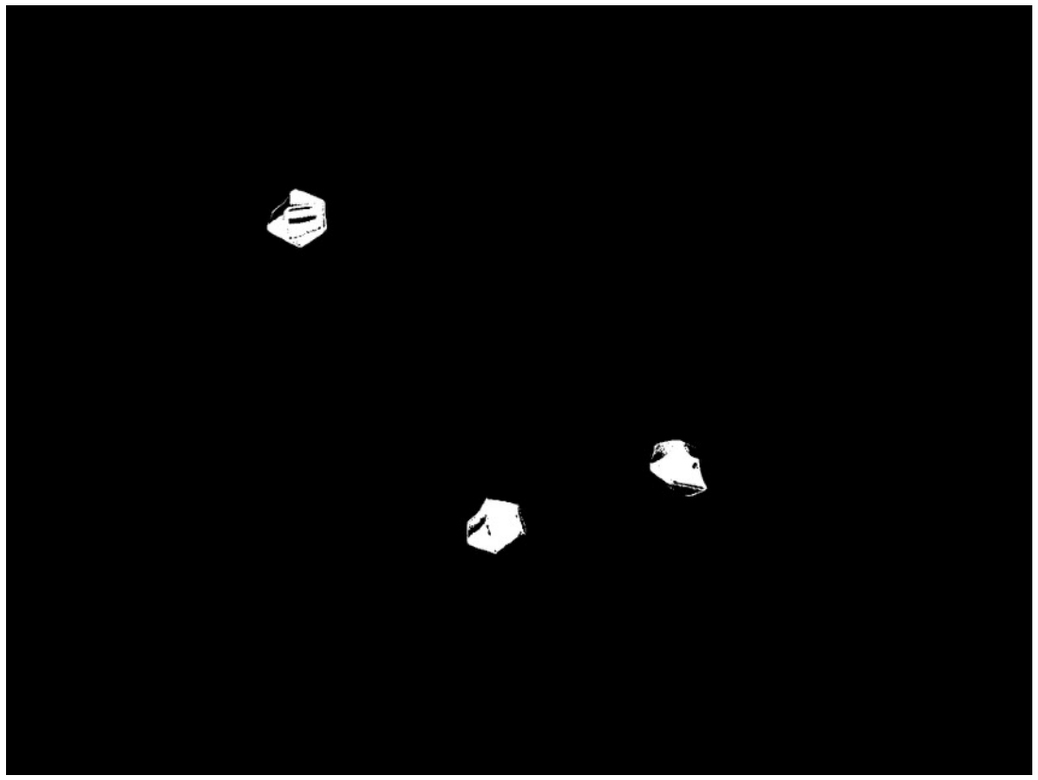
プログラムのポイント

```
#define COUNT (15)
                中略
    //2. グレースケール画像
    cv::cvtColor(src_img, gray_img, cv::COLOR_BGR2GRAY);

    //3. 二値化
    cv::threshold(gray_img, bin_img, TH, MAX_VAL, cv::THRESH_BINARY_INV);

    //4. クロージング・オープニング
    cv::dilate(bin_img, bin_img, cv::Mat(), cv::Point(-1,-1), COUNT);
    cv::erode(bin_img, bin_img, cv::Mat(), cv::Point(-1,-1), COUNT*2);
    cv::dilate(bin_img, dst_img, cv::Mat(), cv::Point(-1,-1), COUNT);
```

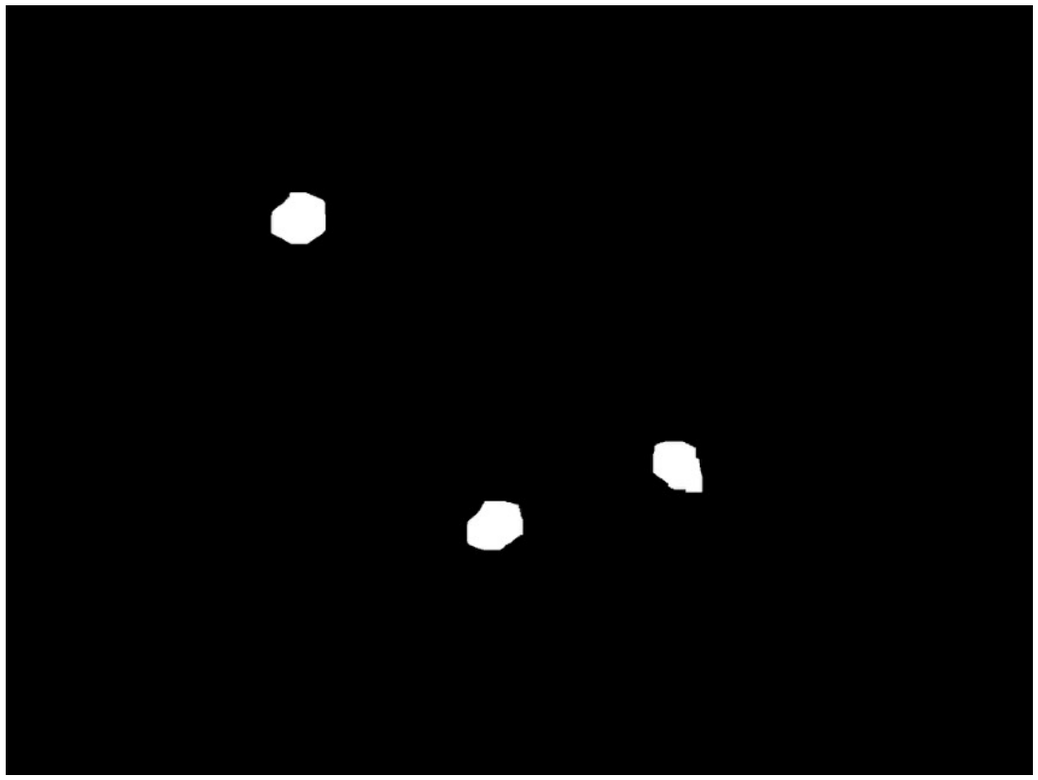
前半はgem_binarizeと一緒

- クロージング：dilate→erode、オープニング：erode→dilate なのでerodeを結合している（COUNT＊2）
- 二値画像bin_imgを使い回しして（出力画像として上書き→次の入力画像）、最後のみdst_imgにしている

処理結果



処理前の二値画像



処理後の二値画像

黒曜石中の欠けがなくなっている

復習

輪郭追跡

(教科書p.74)

・輪郭を求める目的

➡ 図形領域の位置・形状がわかる

➡ 図形領域の特徴がわかる（領域特徴量）

➡ 図形領域を変形できる・数えることができる　等

画像内の図形領域ごとに処理を行うことができる

・輪郭追跡

－ 輪郭を構成する画素列を求める処理

復習

輪郭追跡関数

- findContours：入力二値画像の輪郭を追跡する

```
cv::findContours(二値画像, 輪郭, 追跡モード, 輪郭近似手法);
```

- 今回の記載例

```
//輪郭追跡
cv::findContours(tmp_img, contours,
   cv::RETR_EXTERNAL, cv::CHAIN_APPROX_NONE);
```

- cv::RETR_EXTERNAL: 外輪郭のみ追跡
- cv::CHAIN_APPROX_NONE: 8近傍、近似なし

輪郭追跡結果がcontoursに格納される

輪郭の座標リストの宣言

```
std::vector< std::vector< cv::Point > > contours;
```

- std::vector: 動的配列
  - 配列の大きさが固定されない（可変長）
- cv::Point: 座標
  - 二次元の座標
- **画素位置の 並び（＝輪郭）**をリスト化
  (cv::Point) の (std::vector) の (std::vector)

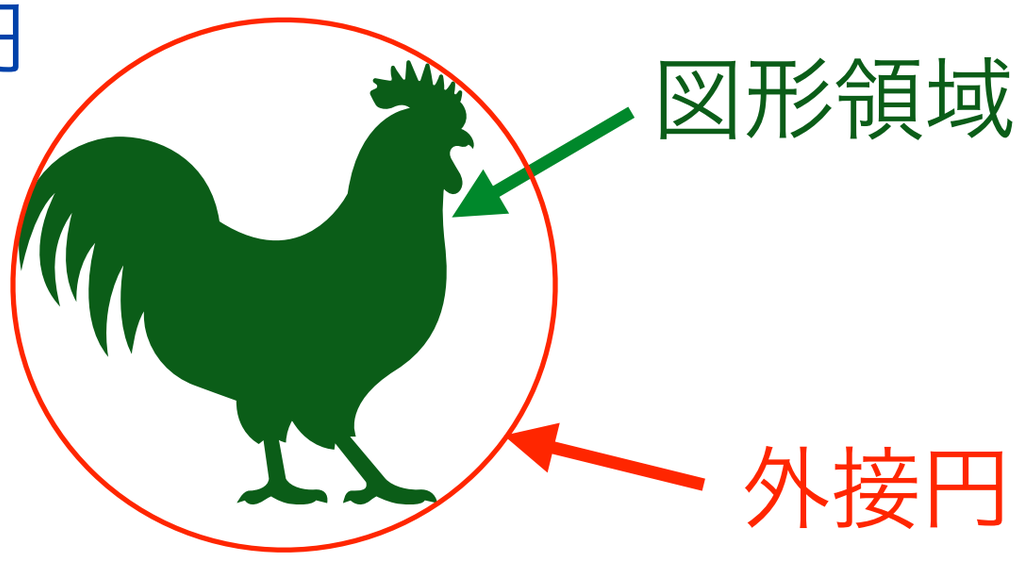
個数と領域の出力

- 輪郭数（＝領域数＝石の個数）
  - contours.size() で得られる
  - 今回はstd::coutでコンソールに出力
- 領域
  - 今回は石の外接円を描画

メモ

外接円

・外接円：図形領域に接する最小の円



・検出関数：minEnclosingCircle

- 入力：輪郭　出力：外接円の中心座標と半径

・円の描画関数：circle（画像, 中心, 半径, 色, 線の太さ）

・例

```
for (int i=0; i<contours.size(); i++) {
    float radius;        // 半径
    cv::Point2f center; // 中心座標
    cv::minEnclosingCircle(contours[i], center, radius);
    cv::circle(dst_img, center, (int)radius, CV_RGB(255, 0, 0), 2); // 赤色2画素
}
```

演習: 黒曜石を数える　その２

- プロジェクト名: gem_count_black　（前と同じ）
- 入力画像から黒曜石を検出し輪郭追跡を行い出力
  - 入力画像: gem1.jpg
  - 先ほどの演習プロジェクトの続きを作成

✓ オープニング・クロージングを行った二値画像で輪郭追跡

✓ 輪郭ごとに外接円を求めて円を赤色で描画

✓ 輪郭数を黒曜石の個数としてコンソールに出力

処理の流れ

1. 入力画像をカラー画像として入力
2. グレースケール画像に変換
3. 二値化
4. クロージング・オープニング
5. 輪郭追跡
6. 外接円を描画
7. 個数を出力
8. 表示

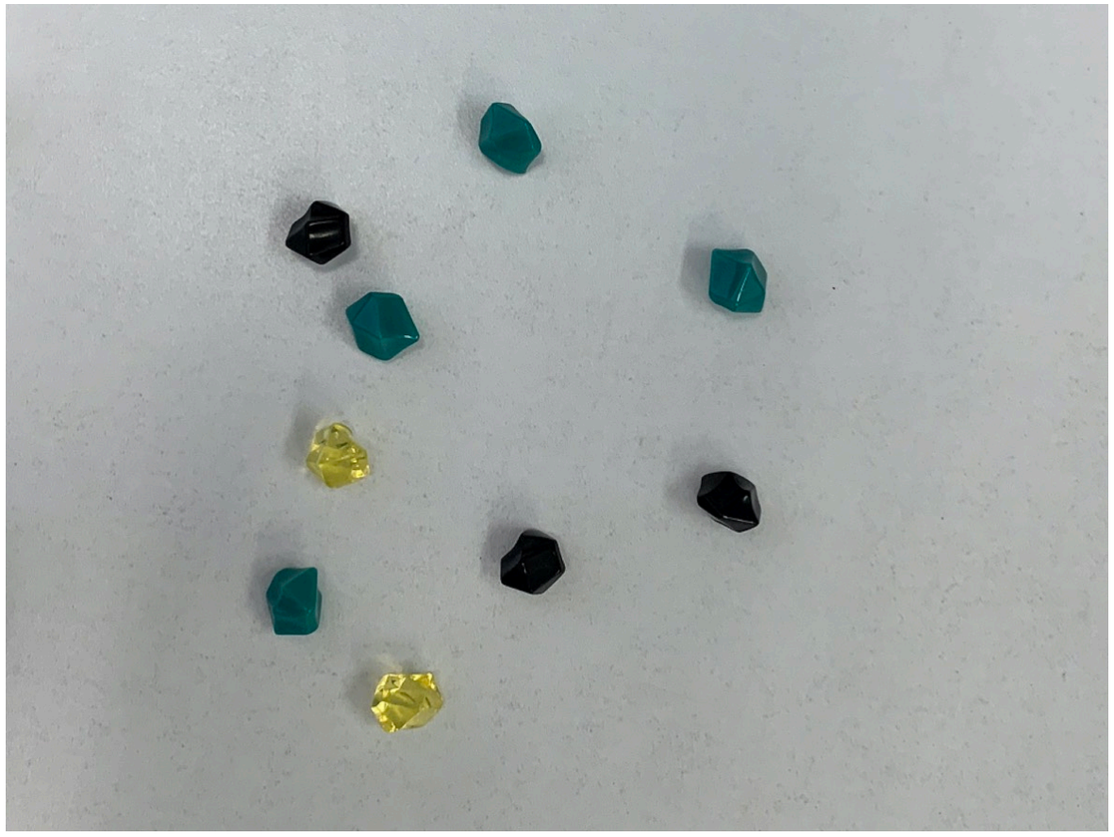
プログラムのポイント

```
cv::dilate(bin_img, bin_img, cv::Mat(), cv::Point(-1,-1), COUNT);

//5. 輪郭追跡による領域検出
tmp_img = bin_img.clone();  //二値画像をコピー
cv::findContours(tmp_img, contours, cv::RETR_EXTERNAL,
                                    cv::CHAIN_APPROX_NONE);  // 外輪郭のみ

//6. 外接円を描画
dst_img = src_img.clone();  //入力画像を出力画像にコピー
for (int i=0; i<contours.size(); i++) {
    float radius;
    cv::Point2f center;
    cv::minEnclosingCircle(contours[i], center, radius);    //外接円を求める
    cv::circle(dst_img, center, (int)radius, CV_RGB(255, 0, 0), 2); // 赤色
}

//7. 個数を出力
std::cout << "Black gem = " << contours.size() << std::endl;
```

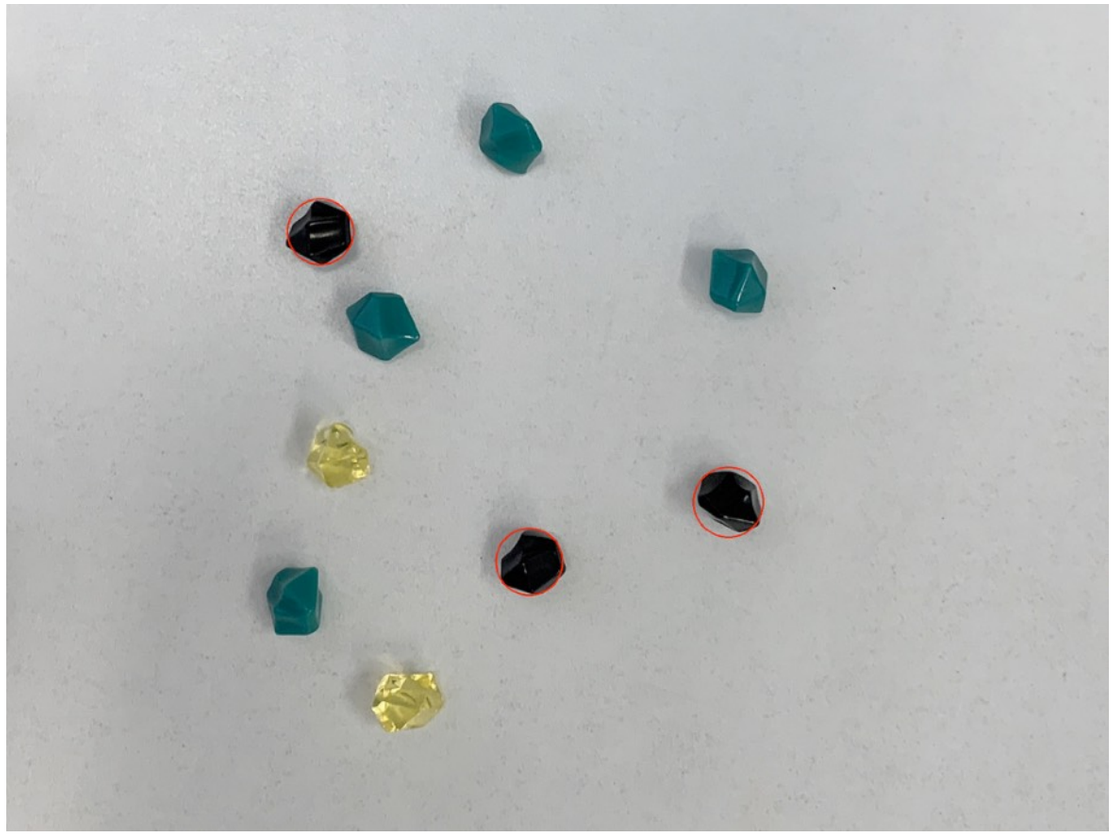
bin_imgに修正

用いる画像や輪郭の変数を適切に宣言すること

処理結果



入力画像 gem1.jpg



出力画像

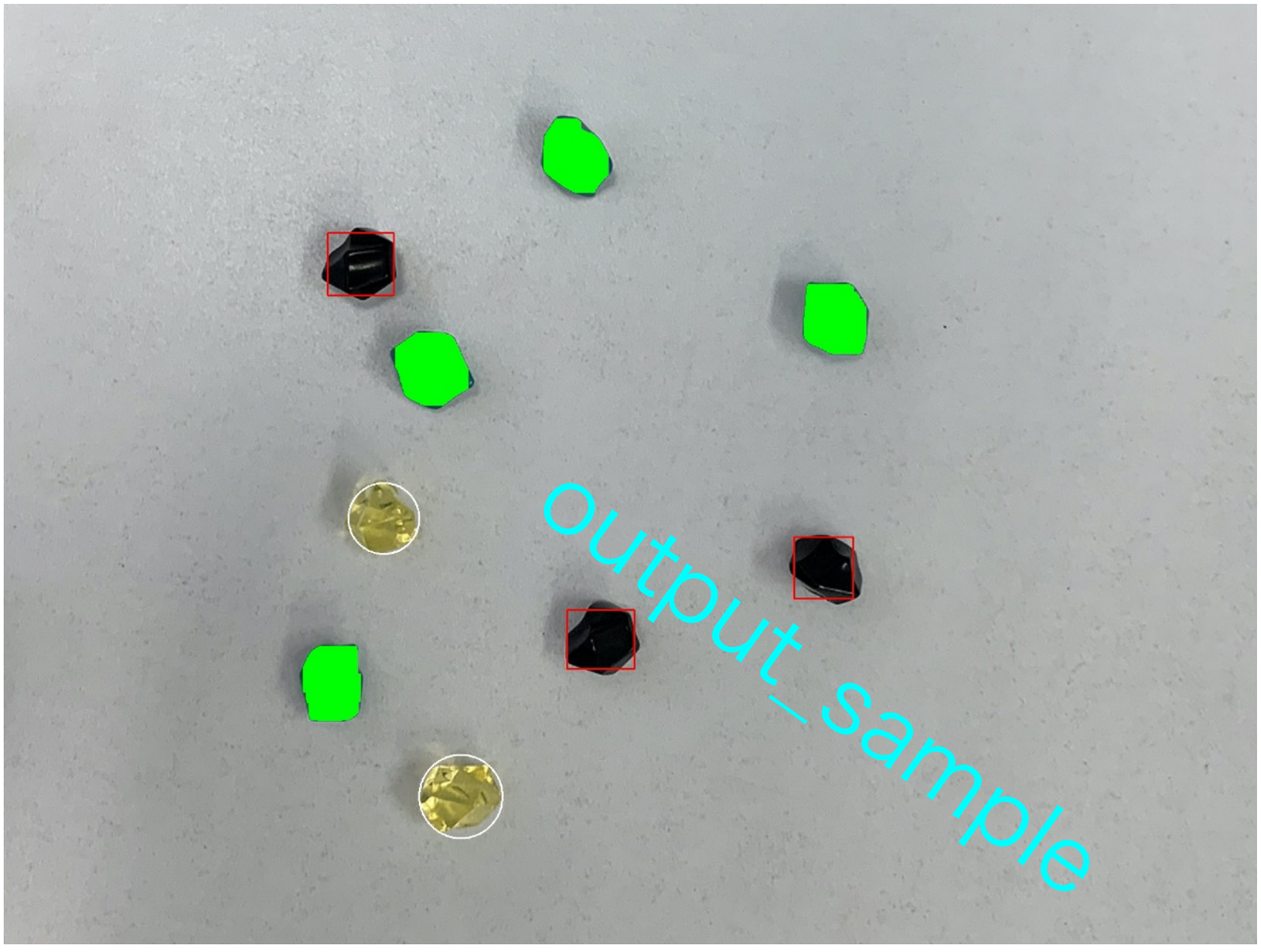
```
Black gem = 3
```

コンソール出力

課題

- 入力画像から各宝石の個数を求め、その位置を入力画像に上書き描画して出力する1つのプログラムを作成せよ
  - 入力画像は gem1.jpg
  - 黒曜石・トルコ石・黄金それぞれの個数をコンソールに出力
  - 同時にそれぞれの画像位置に描画を行う
    - ✓ 黒曜石（Black gem）は赤で外接長方形を
    - ✓ トルコ石（Green gem）は緑で塗りつぶし
    - ✓ 黄金（Golden gem）は白で外接円を
- 提出ファイル
  - ソースプログラム 14_01_学籍番号.cpp
  - 出力画像：コンソール出力の画面キャプチャと描画画像
    （ファイル名は任意　一緒に画面キャプチャしても良い）

出力例



出力画像

```
Black gem = 3
Green gem = 4
Golden gem = 2
```

コンソール出力

ヒント

- 処理の流れ
  - 二値化・カラー画像処理・HSV変換で各宝石を検出
  - それぞれの二値画像に対し、膨張収縮・輪郭追跡を行い、個数出力と描画を行う
  - 検出→描画を3回繰り返しても良いし、全て検出→全て描画の順でも良い
- 変数の使い方
  - 二値画像は各宝石の検出処理ごとに用意するのが無難
  - 輪郭変数（contours）は共用してもよいが、追跡処理のたびに一旦内容をクリアする必要がある
    - ✓ contours.clear()　で可能
  - 描画先（出力画像）は共通にする
- その他
  - 同じ処理は関数化するとスマート（だが必須ではない）

感想と要望

- 感想や要望をお知らせください

- 「授業フィードバックアンケート」にも
ご回答ください