# Работа 4. Детектирование области документа на кадрах видео

автор: Парчиев Р.Б.
дата: 2022-04-11T13:41:26

url: https://github.com/J0hnArren/Image-Processing-with-OpenCV/tree/main/prj.labs/lab04

## Задание

0. текст, иллюстрации и подписи отчета придумываем самостоятельно
1. самостоятельно снимаем видео смартфоном

- объект съемки - купюры (рубли разного номинала), расправленные и лежащие на поверхности (проективно искаженны прямоугольник)
- количество роликов - от 5 шт.
- длительность - 5-7 сек
- условия съемки разные

2. извлекаем по 3 кадра из каждого ролика (делим кол-во кадров на 5 и берем каждый с индеком 2/5,3/5,4/5)
3. цветоредуцируем изображения
4. бинаризуем изображения
5. морфологически обрабатываем изображения
6. выделяем основную компоненту связности
7. руками изготавливаем маски (идеальная зона купюры)
8. оцениваем качество выделение зоны и анализируем ошибки

## Результаты

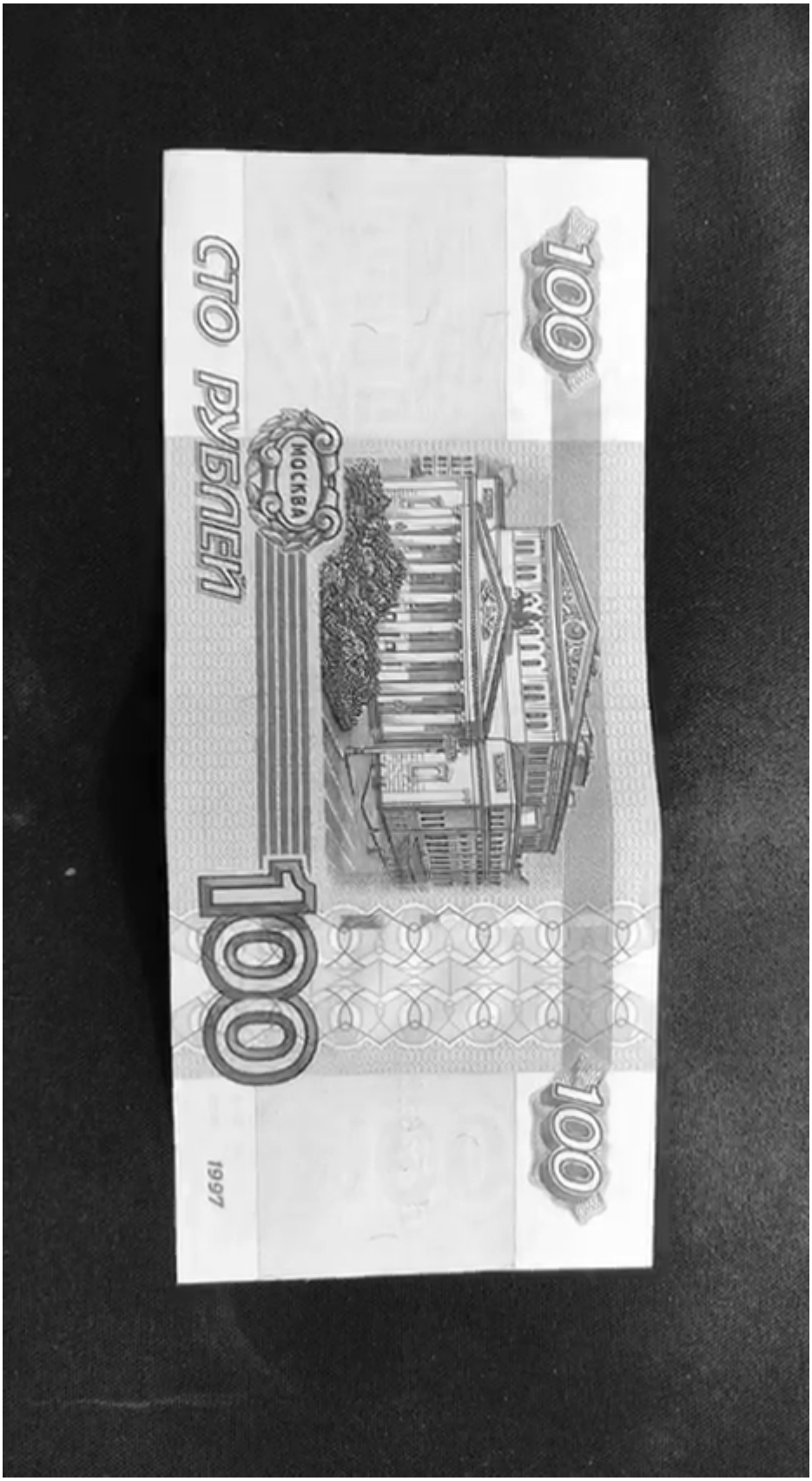
В результате извлечения кадров указанным выше образом из 5 отснятых видео было извлечено по 3 изображения (рис.1):

СТО РУБЛЕЙ
МОСКВА
100
100
100
1997

БИЛЕТ БАНКА РОССИИ
2000
ДВЕ ТЫСЯЧИ РУБЛЕЙ
ПОДДЕЛКА БИЛЕТОВ БАНКА РОССИИ ПРЕСЛЕДУЕТСЯ ПО ЗАКОНУ
2000


Рис. 1. Примеры исходных кадров из видео.

Далее было применено цветоредуцирование (рис. 2):



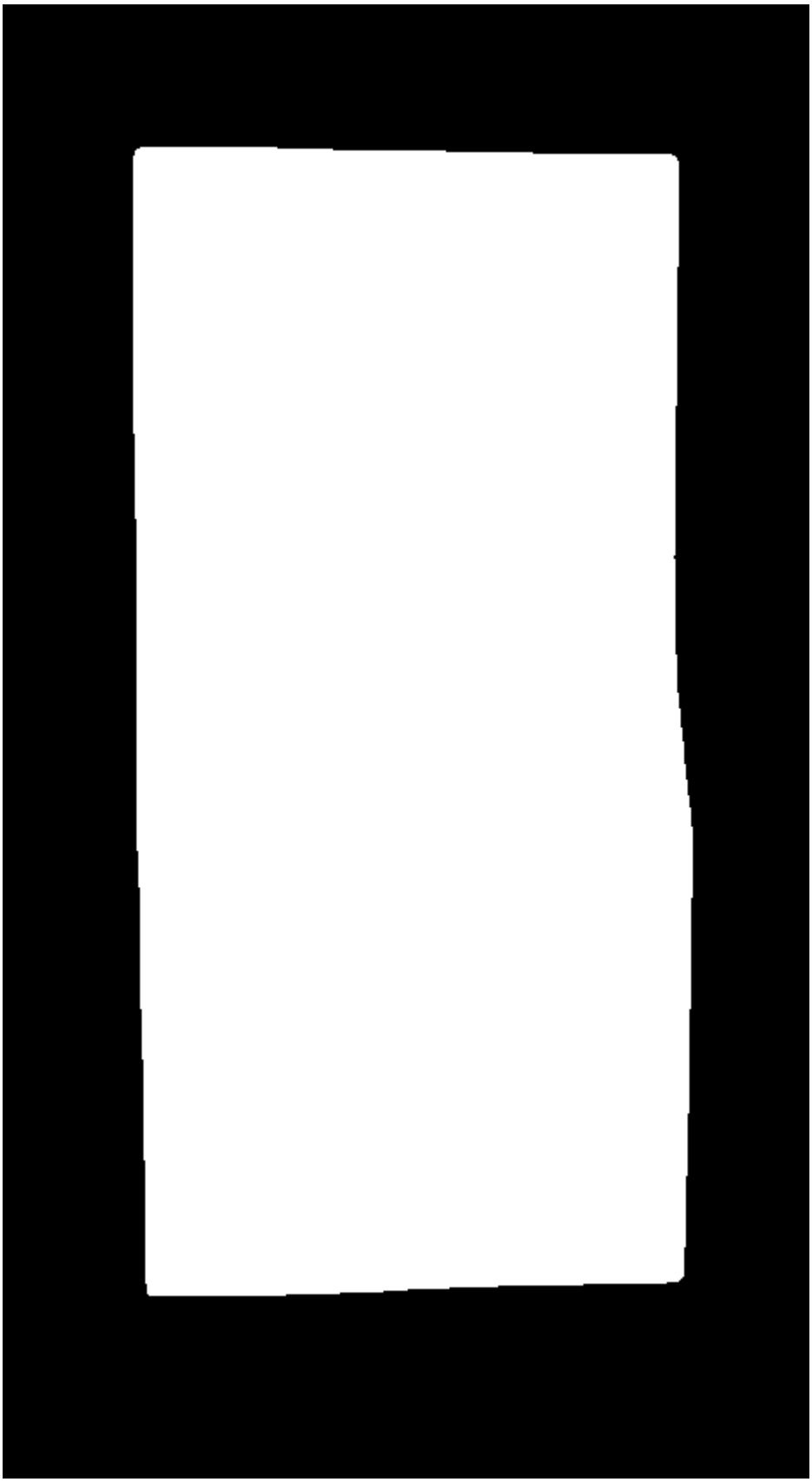
БИЛЕТ БАНКА РОССИИ
2000
2000
ДВЕ ТЫСЯЧИ РУБЛЕЙ
ПОДДЕЛКА БИЛЕТОВ БАНКА РОССИИ ПРЕСЛЕДУЕТСЯ ПО ЗАКОНУ

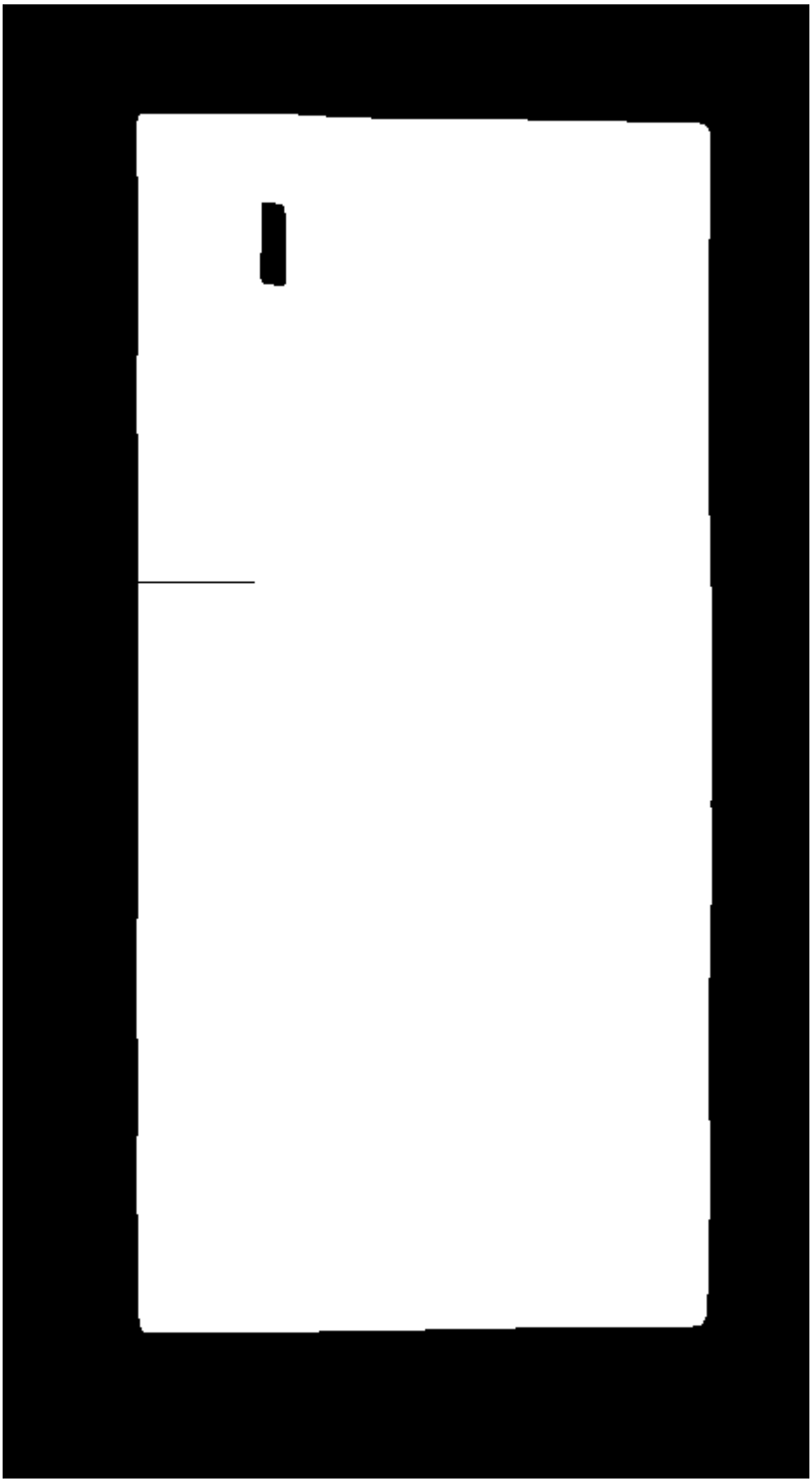

Рис. 2. Цветоредуцированные изображения.

Затем цветоредуцированные изображения бинаризуются при помощи глобальной бинаризации OTSU (рис. 3):

Рис. 3. Бинаризированные изображения.

В результате, к полученным изображениям применяются морфологические преобразования для удаления на них "шумов" и создания более гладких и правильных границ (Рис. 4).
Используемые операции: close, opene и dilate.





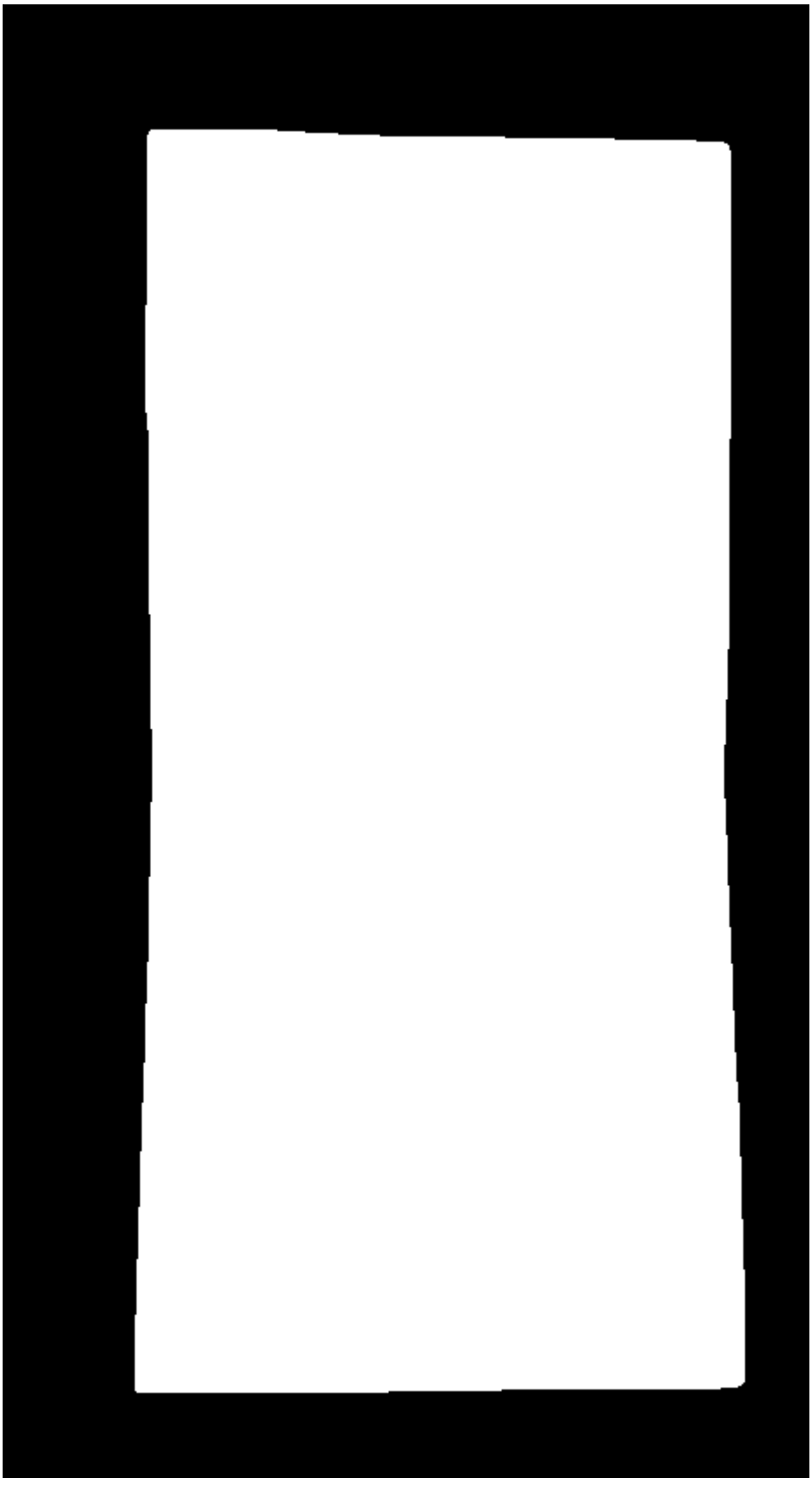
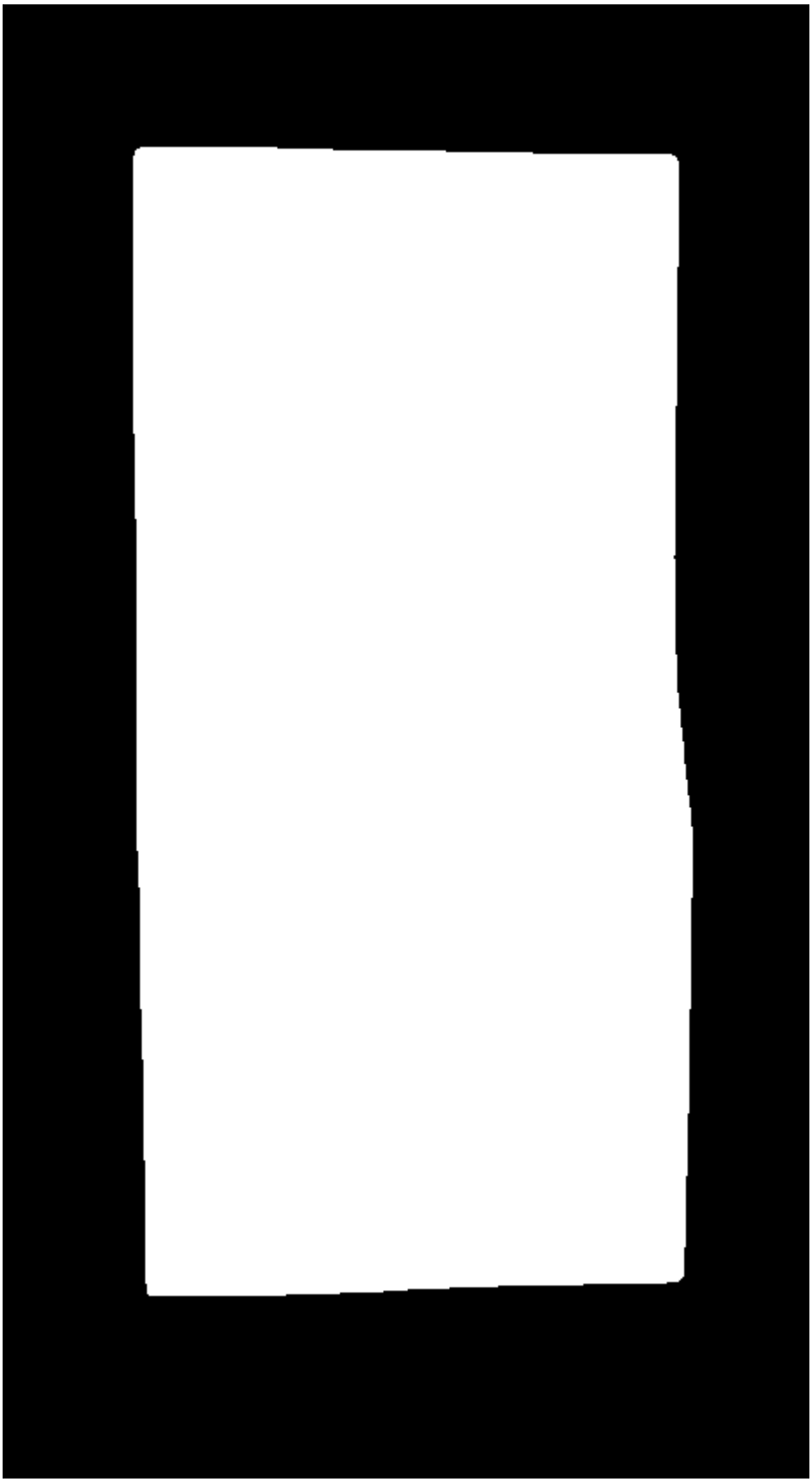
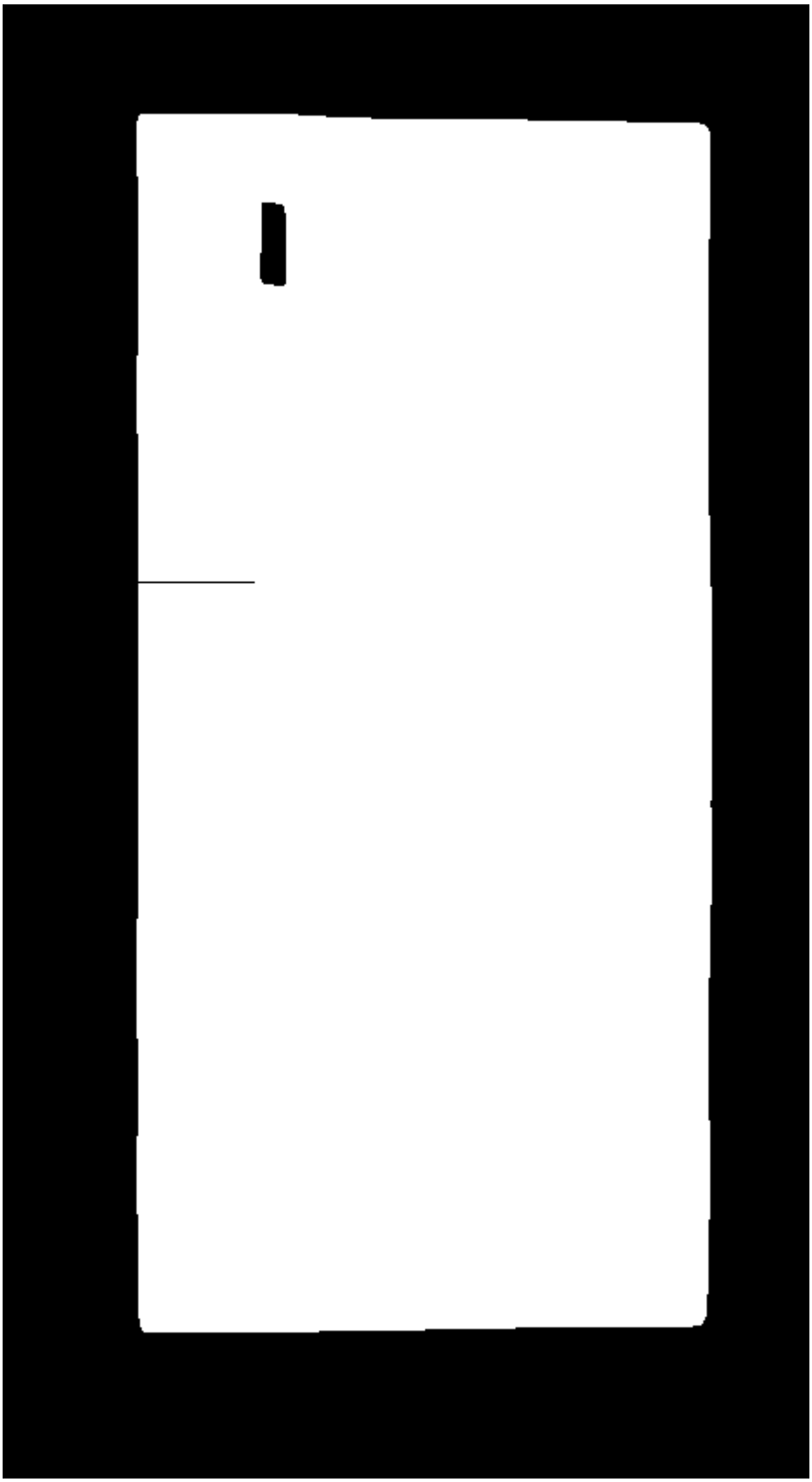


Рис. 4. Морфологически обработанные изображения.

Далее для получения маски производтся нахождение наибольшей по площади компоненты связности (КС), а остальные КС при этом закрашиваются в черный. Для закрашивания в белый цвет внутренних пикселей маски по белому изображению в горизонтальном направлении наносятся черные пиксели с двух сторон до момента встречи первого белого пикселя на маске(рис. 5):





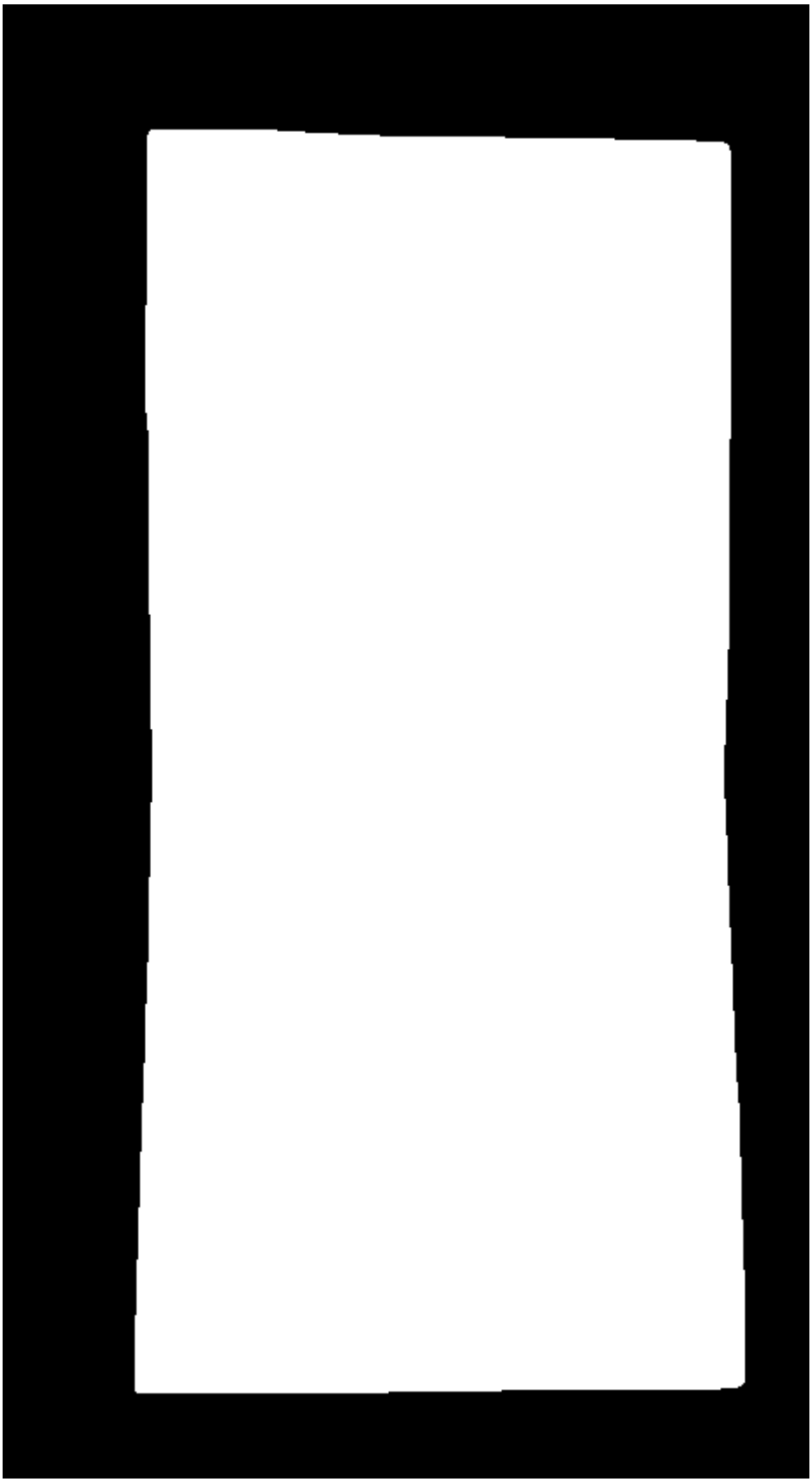
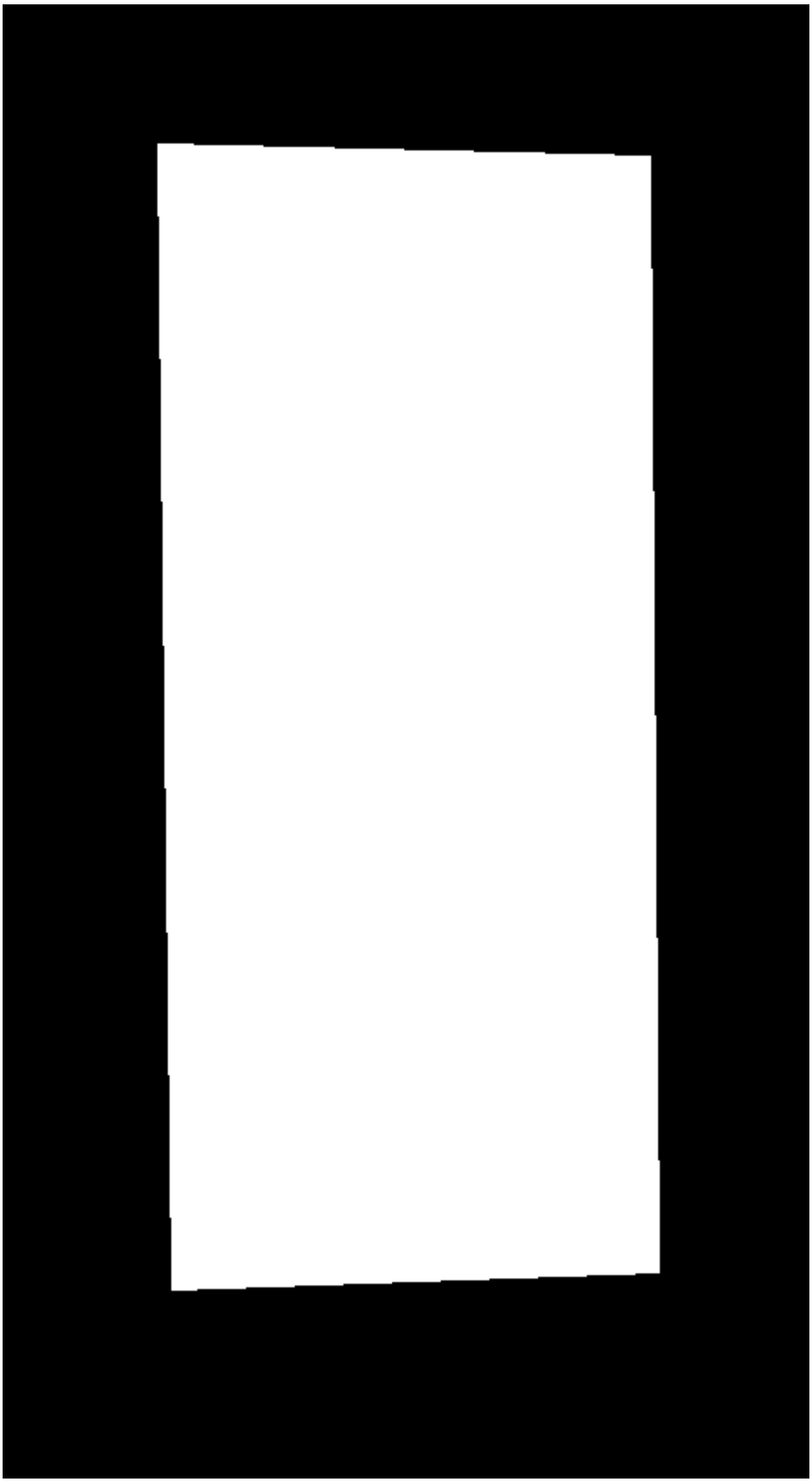
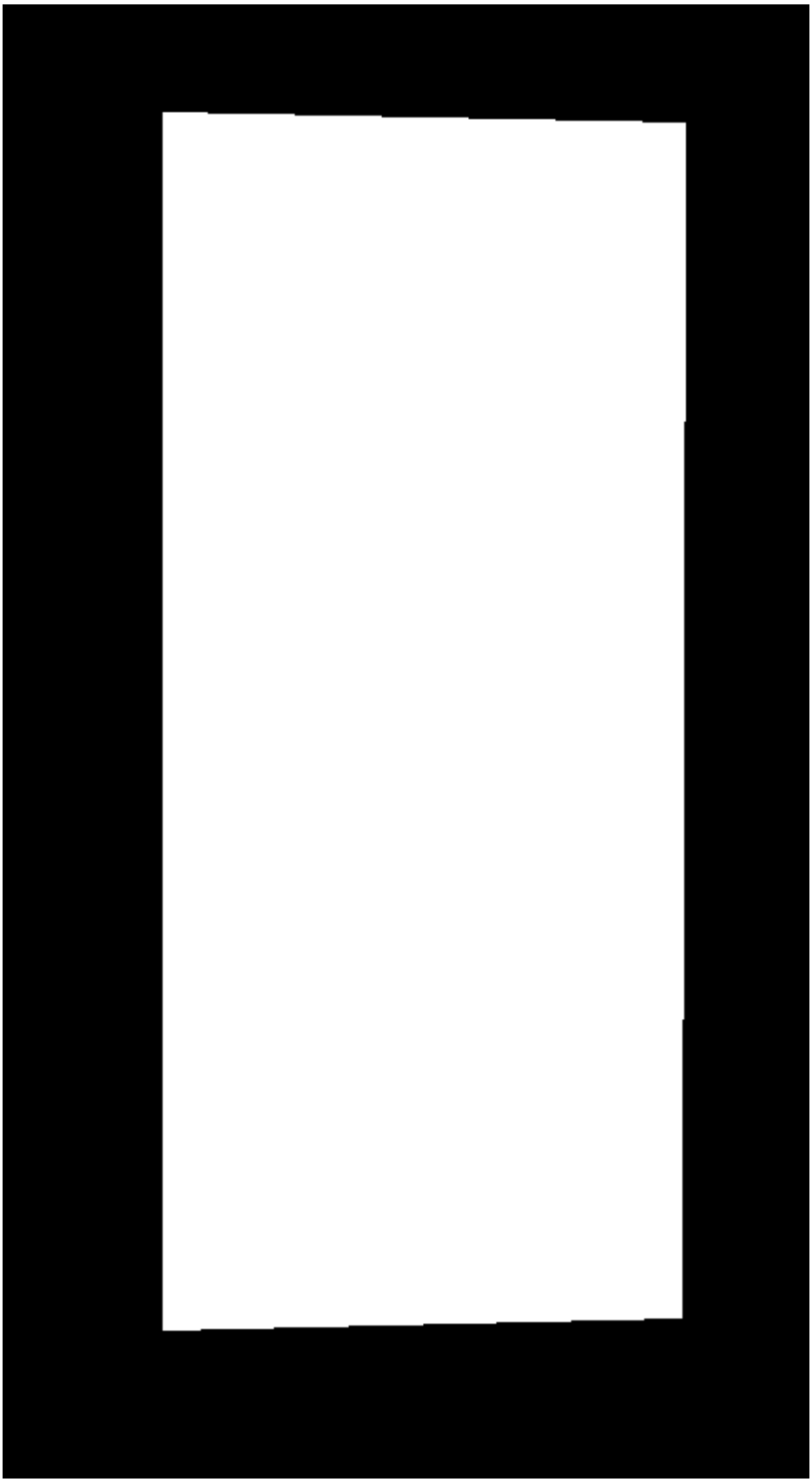


Рис. 5. Полученные маски.

Для проверки качества полученных созданных масок вручную задаются координаты эталонных маскок (рис. 6):





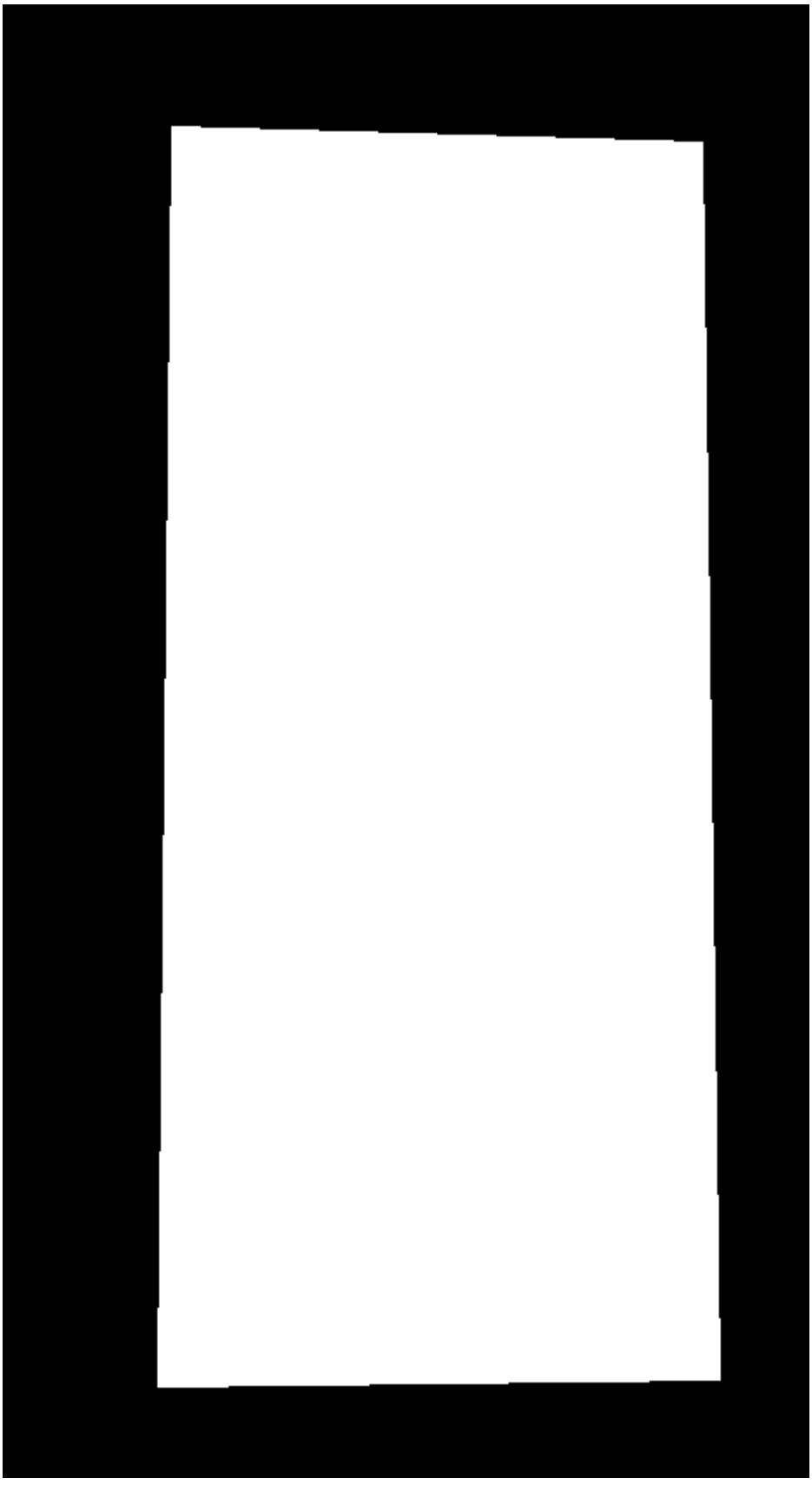


Рис. 6. Эталонные маски.

Для визуализации отличий маски накладываются на исходные кадры в разных цветовых каналах: (рис. 7):

- Красный - программная маска
- Зеленый - эталонная маска

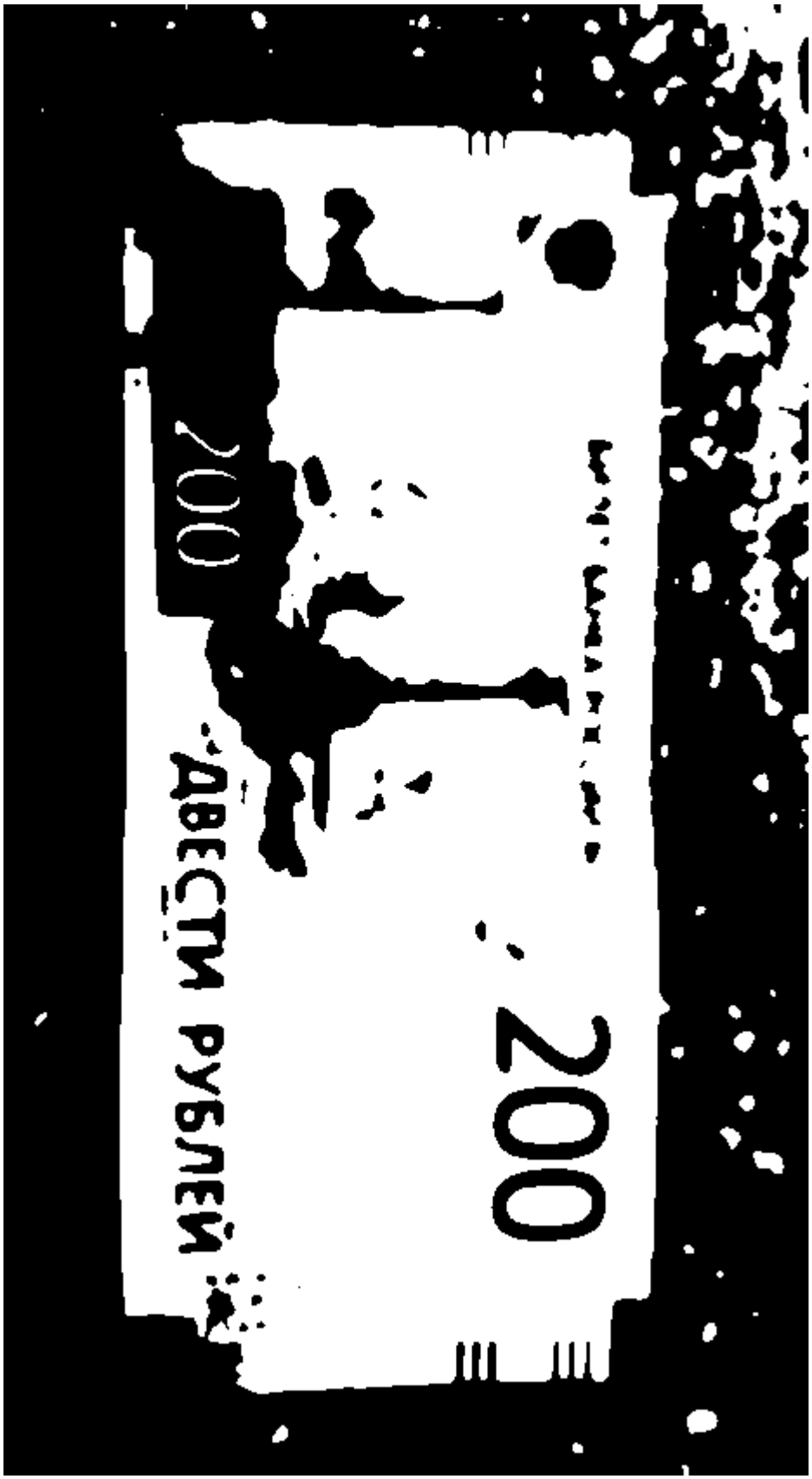
- Желтый - пересечение масок


БИЛЕТ БАНКА РОССИИ
2000
ДВЕ ТЫСЯЧИ РУБЛЕЙ
2000


Рис. 7. Визуализация отличий трех масок.

Для примеров рисунков использовались изображения под номером 2, 8 и 15 соответственно. Подводя итог, можно легко заметить неточность во втором изображении, так как на нем был более сильный свет, однако на пером и третьем изображениях маски определены довольно точно.

В результате опытов, было замечено, что фоны со светлыми оттенками сильно понижают точность масок (рис. 8), а черный фон, напротив, делает купюру более различимой даже при интесивном

освещении. 

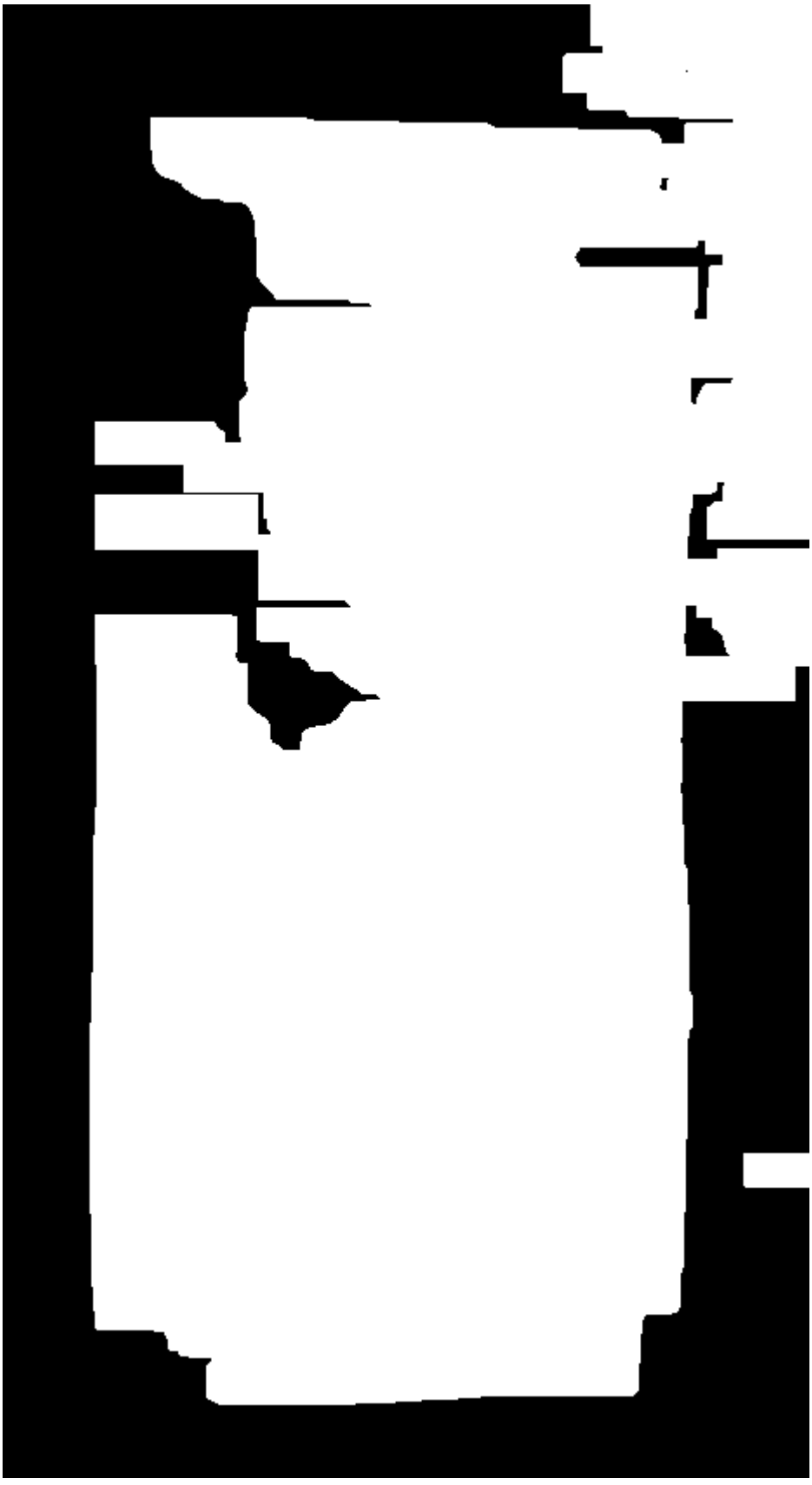


Рис. 8. Фрейм 5 с неудачным светлым фоном.

## Точность программных масок

Точностью программной маски относительно эталонной считается как отношение пересечения пикселей масок на их объединение. Данная оценка показывает коэффициент соответствия от 0 до 1, где близость к 1 показывается большую идентичность масок.

Ниже приведены точности для 15 изображений (по 3 кадра из 5 видео), где эталонные маски были вручную заданы под полученные изображения.

| Название файла | Точность |
|---|---|
| frame_1.png | 0.753731 |
| frame_2.png | 0.89671 |
| frame_3.png | 0.894071 |
| frame_4.png | 0.765661 |
| frame_5.png | 0.742 |
| frame_6.png | 0.744401 |
| frame_7.png | 0.903 |
| frame_8.png | 0.884048 |
| frame_9.png | 0.882077 |
| frame_10.png | 0.908172 |
| frame_11.png | 0.903704 |
| frame_12.png | 0.904756 |
| frame_13.png | 0.888107 |
| frame_14.png | 0.919185 |
| frame_15.png | 0.926419 |

Итого: благодаря проведенным преобразованиям на черном фоне получилась довольно высокая точность, однако более светлый фон второго видео (4-6 фреймы) заметно пониизил точность.

### Текст программы

```
#include <opencv2/opencv.hpp>
#include <iostream>
#include <fstream>
#include "json.hpp"

using json = nlohmann::json;

const size_t NUMBER_OF_VIDEOS = 5;
const std::string VID_NAMES[NUMBER_OF_VIDEOS] = { "100", "200", "2000", "1000",
"5000" };
const std::string VIDS_FORMAT = ".mp4";
const std::string IMAGE_FORMAT = ".png";
const std::string RESULTS_PATH = "./results/";
const std::string DATA_PATH = "./data/";
const std::string MASKS_COORDS = DATA_PATH + "masks_coords.json";
```

```
void extracting_frames() {
    for (int video = 0; video < NUMBER_OF_VIDEOS; ++video) {
        cv::VideoCapture capture(DATA_PATH + VID_NAMES[video] + VIDS_FORMAT);
        if (!capture.isOpened()) {
            std::cerr << "Unable to load the video from path:" + DATA_PATH +
VID_NAMES[video] + "\n";
            exit(1);
        }
        std::vector<cv::Mat> result;
        cv::Mat frame;
        result.reserve(300);
        for (;;) {
            capture >> frame;
            if (frame.empty()) {
                break;
            }
            result.push_back(frame.clone());
        }
        auto total = result.size();
        int img_total = 3;
        for (int i = 0; i < img_total; ++i) {
            int img_num = img_total * video + i + 1;
            int index = (i + 2) * total / 5;
            cv::imwrite(RESULTS_PATH + "frame_" + std::to_string(img_num) +
IMAGE_FORMAT, result[index]);
        }
    }
}

std::vector<cv::Mat> read_frames(const size_t& frames_num) {
    std::vector<cv::Mat> result;
    for (size_t i = 0; i < frames_num; ++i) {
        auto img = cv::imread(RESULTS_PATH + "frame_" + std::to_string(i + 1) +
IMAGE_FORMAT, cv::IMREAD_COLOR);
        result.push_back(img.clone());
    }
    return result;
}

std::vector<cv::Mat>& binarize(std::vector<cv::Mat>& frames) {
    int image_num = 1;
    for (auto& frame : frames) {
        cv::cvtColor(frame, frame, cv::COLOR_BGRA2GRAY, 0);
        cv::imwrite(RESULTS_PATH + "grayscale_frame_" + std::to_string(image_num)
+ IMAGE_FORMAT, frame);
        cv::GaussianBlur(frame, frame, cv::Size(15, 15), 0);
        cv::threshold(frame, frame, 145, 255, cv::THRESH_BINARY +
cv::THRESH_OTSU);
        cv::imwrite(RESULTS_PATH + "bin_frame_" + std::to_string(image_num) +
IMAGE_FORMAT, frame);
        ++image_num;
    }
```

```
    return frames;
}

std::vector<cv::Mat>& morph_processing(std::vector<cv::Mat>& frames) {
    int image_num = 1;
    auto kernel = cv::getStructuringElement(cv::MORPH_RECT, cv::Size(35, 10));
    for (auto& frame : frames) {
        cv::morphologyEx(frame, frame, cv::MORPH_CLOSE, kernel);
        cv::morphologyEx(frame, frame, cv::MORPH_OPEN, kernel);
        cv::dilate(frame, frame, kernel);
        cv::imwrite(RESULTS_PATH + "morph_frame_" + std::to_string(image_num) +
IMAGE_FORMAT, frame);
        ++image_num;
    }
    return frames;
}

const bool intersect(const cv::Rect& rect1, const cv::Rect& rect2) noexcept {
    return (rect1.contains(cv::Point(rect2.x, rect2.y)) ||
        rect1.contains(cv::Point(rect2.x + rect2.width - 1,
            rect2.y + rect2.height - 1))
        );
}

void create_mask(cv::Mat& mask, const cv::Mat& image) {
    for (int i = 0; i < mask.rows; ++i) {
        int last_updated_ind = 0;
        for (int j = 0; j < mask.cols; ++j) {
            if (image.at<uchar>(i, j) == 255) break;
            else {
                last_updated_ind++;
                mask.at<uchar>(i, j) = 0;
            }
        }
        for (int k = mask.cols - 1; k > last_updated_ind; --k) {
            if (image.at<uchar>(i, k) == 255) break;
            else {
                mask.at<uchar>(i, k) = 0;
            }
        }
    }
}

std::vector<cv::Mat>& get_mask(std::vector<cv::Mat>& frames) {
    int image_num = 1;
    for (const auto& frame : frames) {
        cv::Mat labels, stats, centroids;
        cv::Mat selected_img_part(frame.rows, frame.cols, frame.type(),
cv::Scalar(0, 0, 0));
        cv::Mat mask(frame.rows, frame.cols, frame.type(), 255);
        int max_comp_ind = 0, max_comp_stat = 0;

        cv::connectedComponentsWithStats(frame, labels, stats, centroids);
```

```
        for (int i = 1; i < stats.rows; ++i) {
            int square = stats.at<int>(i, 4);
            if (square > max_comp_stat) {
                max_comp_ind = i;
                max_comp_stat = square;
            }
        }

        cv::Rect2i max_component = { stats.at<int>(max_comp_ind, 0),
                                     stats.at<int>(max_comp_ind, 1),
                                     stats.at<int>(max_comp_ind, 2),
                                     stats.at<int>(max_comp_ind, 3) };

        for (int i = max_component.x; i < max_component.x + max_component.width;
++i) {
            for (int j = max_component.y; j < max_component.y +
max_component.height; j++) {
                selected_img_part.at<uchar>(j, i) = frame.at<uchar>(j, i);
                mask.at<uchar>(j, i) = 255;
            }
        }

        for (int i = 1; i < stats.rows; ++i) {
            cv::Rect2i component = { stats.at<int>(i, 0), stats.at<int>(i, 1),
stats.at<int>(i, 2), stats.at<int>(i, 3) };
            if (i != max_comp_ind && intersect(max_component, component)) {
                cv::rectangle(selected_img_part, component, 255, cv::FILLED);
            }
        }

        create_mask(mask, selected_img_part);
        cv::imwrite(RESULTS_PATH + "modify_morph_" + std::to_string(image_num) +
IMAGE_FORMAT, frame);
        ++image_num;
    }
    return frames;
}

std::vector<cv::Mat> create_etalon_mask(const std::vector<std::array<cv::Point,
4>>& points,
                                        const std::vector<cv::Mat>& images) {

    int last_index = 0;
    std::vector<cv::Mat> result;
    for (const auto& vertexes : points) {
        cv::Point pts[1][4];
        pts[0][0] = vertexes[0];
        pts[0][1] = vertexes[1];
        pts[0][2] = vertexes[2];
        pts[0][3] = vertexes[3];
        const cv::Point* ppt[1] = { pts[0] };
        int npt[] = { 4 };
        auto img = images[last_index].clone();
        img = 0;
```

```
        cv::fillPoly(img, ppt, npt, 1, cv::Scalar(255, 255, 255));
        cv::imwrite(RESULTS_PATH + "mask_" + std::to_string(last_index + 1) +
IMAGE_FORMAT, img);
        result.push_back(img);
        ++last_index;
    }
    return result;
}

std::vector<std::array<cv::Point, 4>> read_mask_points() {
    std::ifstream input(MASKS_COORDS);
    std::vector<std::array<cv::Point, 4>> result;
    auto jf = json::parse(input);
    for (auto& p : jf) {
        std::array<cv::Point, 4> points;
        points[0] = cv::Point(p["bottom-left"].get<std::array<int, 2>>()[0],
                              p["bottom-left"].get<std::array<int, 2>>()[1]);
        points[1] = cv::Point(p["top-left"].get<std::array<int, 2>>()[0],
                              p["top-left"].get<std::array<int, 2>>()[1]);
        points[2] = cv::Point(p["top-right"].get<std::array<int, 2>>()[0],
                              p["top-right"].get<std::array<int, 2>>()[1]);
        points[3] = cv::Point(p["bottom-right"].get<std::array<int, 2>>()[0],
                              p["bottom-right"].get<std::array<int, 2>>()[1]);
        result.push_back(points);
    }
    return result;
}

std::vector<cv::Mat> create_concatenated_masks(const std::vector<cv::Mat>&
original_frames,
                                               const std::vector<cv::Mat>& mask_frames,
                                               const std::vector<cv::Mat>& created_frames)
{
    auto result(original_frames);
    for (size_t i = 0; i < original_frames.size(); ++i) {
        cv::Mat rgb_image_channels[3];

        cv::split(original_frames[i], rgb_image_channels);
        cv::max(rgb_image_channels[2], mask_frames[i], rgb_image_channels[2]);
        cv::max(rgb_image_channels[1], created_frames[i], rgb_image_channels[1]);
        cv::merge(rgb_image_channels, 3, result[i]);
        cv::imwrite(RESULTS_PATH + "concat_frame_" + std::to_string(i + 1) +
IMAGE_FORMAT, result[i]);
    }
    return result;
}

std::vector<int> process_intersect_masks(const std::vector<cv::Mat>& mask_frames,
                                         const std::vector<cv::Mat>&
created_frames) {
    int right_value = 255;
    auto result = std::vector<int>(mask_frames.size());
    for (int ind = 0; ind < mask_frames.size(); ++ind) {
        for (int i = 0; i < mask_frames[ind].rows; ++i) {
```

```
            for (int j = 0; j < mask_frames[ind].cols; ++j) {
                if (mask_frames[ind].at<uchar>(i, j) == right_value &&
                    mask_frames[ind].at<uchar>(i, j) ==
created_frames[ind].at<uchar>(i, j)) {
                    ++result[ind];
                }
            }
        }
    }
    return result;
}

std::vector<int> process_union_masks(const std::vector<cv::Mat>& mask_frames,
                                     const std::vector<cv::Mat>& created_frames) {
    int right_value = 255;
    auto result = std::vector<int>(mask_frames.size());
    for (int ind = 0; ind < mask_frames.size(); ++ind) {
        for (int i = 0; i < mask_frames[ind].rows; ++i) {
            for (int j = 0; j < mask_frames[ind].cols; ++j) {
                if (mask_frames[ind].at<uchar>(i, j) == right_value ||
                    created_frames[ind].at<uchar>(i, j) == right_value) {
                    result[ind]++;
                }
            }
        }
    }
    return result;
}

double process_precision(const std::vector<cv::Mat>& mask_frames,
                         const std::vector<cv::Mat>& created_masks, const int& i)
{
    auto int_masks = process_intersect_masks(mask_frames, created_masks);
    auto union_masks = process_union_masks(mask_frames, created_masks);
    return (int_masks[i] * 1.0) / union_masks[i];;
}

std::vector<double> create_precision(const std::vector<cv::Mat>& mask_frames,
                                     const std::vector<cv::Mat>& created_masks) {
    std::ofstream out(RESULTS_PATH + "precision_values.txt", std::ios_base::out);
    auto result = std::vector<double>(mask_frames.size());

    for (int i = 0; i < mask_frames.size(); ++i) {
        result[i] = process_precision(mask_frames, created_masks, i);
        out << i + 1 << " - " << result[i] << std::endl;
    }
    return result;
}

int main() {
    extracting_frames();
    std::vector<cv::Mat> frames = read_frames(15);
    std::vector<cv::Mat> original_frames = frames;
    binarize(frames);
```

```
    morph_processing(frames);
    get_mask(frames);
    std::vector<cv::Mat> created_masks = create_etalon_mask(read_mask_points(),
frames);
    create_concatenated_masks(original_frames, frames, created_masks);
    create_precision(frames, created_masks);

    return EXIT_SUCCESS;
}
```