LAB: Dimension Measurement with 2D camera

Date: 2024-April-30

Author: Gyeonheal An 21900416

Partner: Taegeon Han 21900793

Github: https://github.com/AnGyeonheal/DLIP_GH

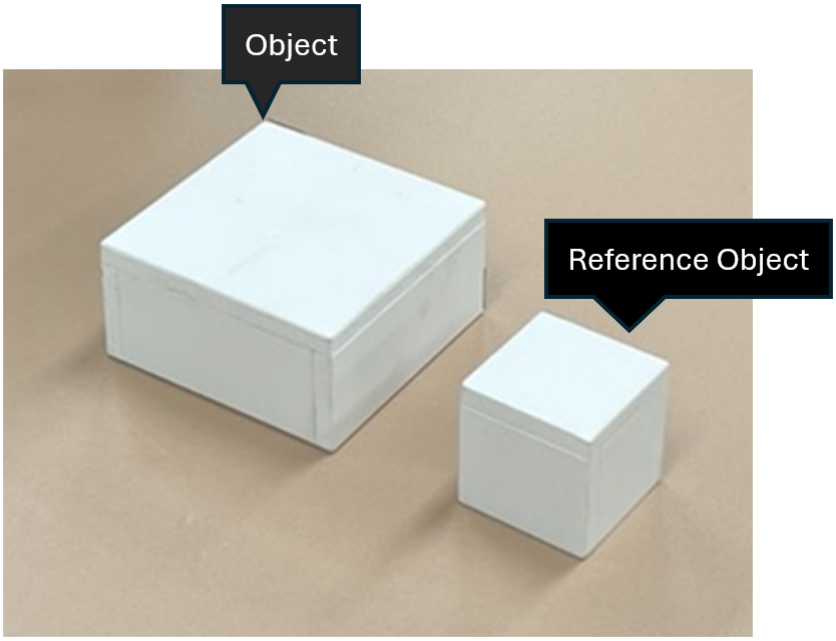
Demo Video: https://www.youtube.com/watch?v=dX_aqF7gaDM

I. Introduction

1. Objective

Goal: To enable 3D measurement (width, length, height) of a target object by taking a photo with a simple 2D camera such as a smartphone.Problem Conditions

- Measure the 3D dimensions (LxWxH) of a small rectangular object
- Assume We know the exact width (**W**) of the target object. We only need to find **L** and **H**.
- The accuracy of the object should be within **3mm**
- We can only use a smartphone (webcam) 2D camera for sensors. No other sensors.
- We cannot know the exact pose of the camera from the object or the table.
- We can use other known dimension objects, such as A4-sized paper, check-board, small square boxes etc
- Try to make the whole measurement process to be as simple, and convenient as possible for the user



2. Preparation

Software Installation

- OpenCV 3.83, Clion JetBrain

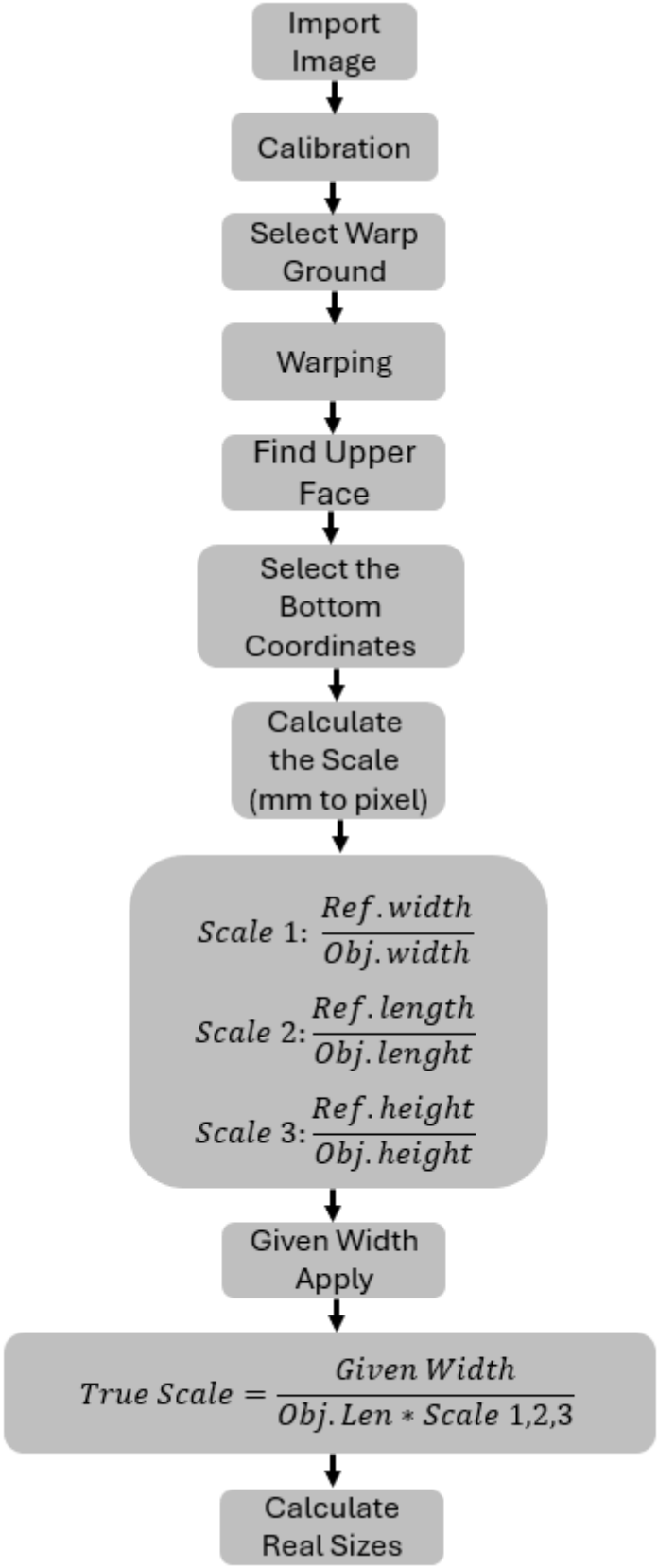
Dataset

A picture of an object taken from three different angles.

Dataset link: Download the test image

II. Algorithm

1. Overview



1. Load the image of the object you want to analyze first.
2. Correct any distortion in the image through calibration.
3. The corrected image is warped from 3D to a 2D plane by the user marking the area to warp with points.
4. The warped image is converted to grayscale and then thresholded to identify the contours of the upper face of the target and reference objects.
5. The user then clicks on the bottom vertices of the target and reference objects on the warped image to input their coordinates.

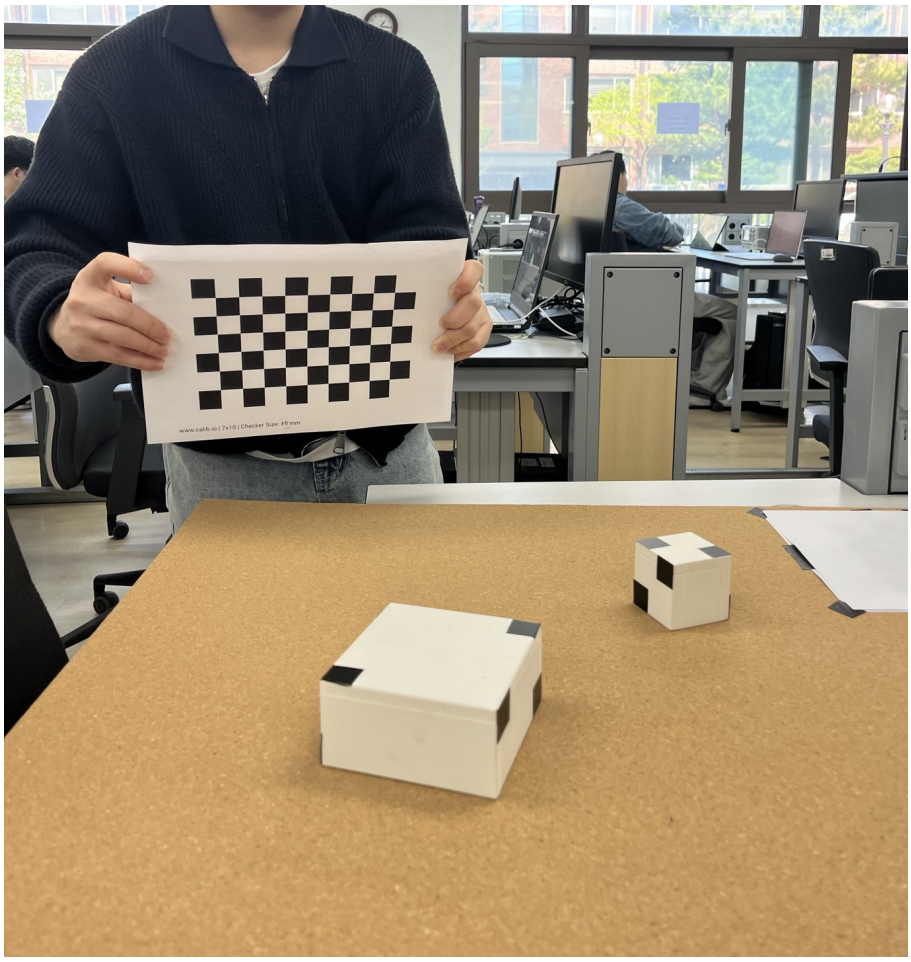
6. Use the coordinates found from the contours and those inputted in step 5 to adjust the scale to the actual length (mm) of the reference object in pixels.
7. Re-adjust the scale using the width provided under LAB conditions to obtain more accurate measurements.
8. Convert the findings from pixels to millimeters using the true scale calculated to determine the actual size (length, height) of the object.

2. Assumption

1. The top surface of the object being measured must be parallel to the Warp plane.
2. All four corners of the Warp plane must be visible in the photo.
3. Three sides of the object being measured must be visible.
4. The side surfaces of the reference object and the object being measured must be parallel.
5. The reference object must be located to the right of the object being measured.

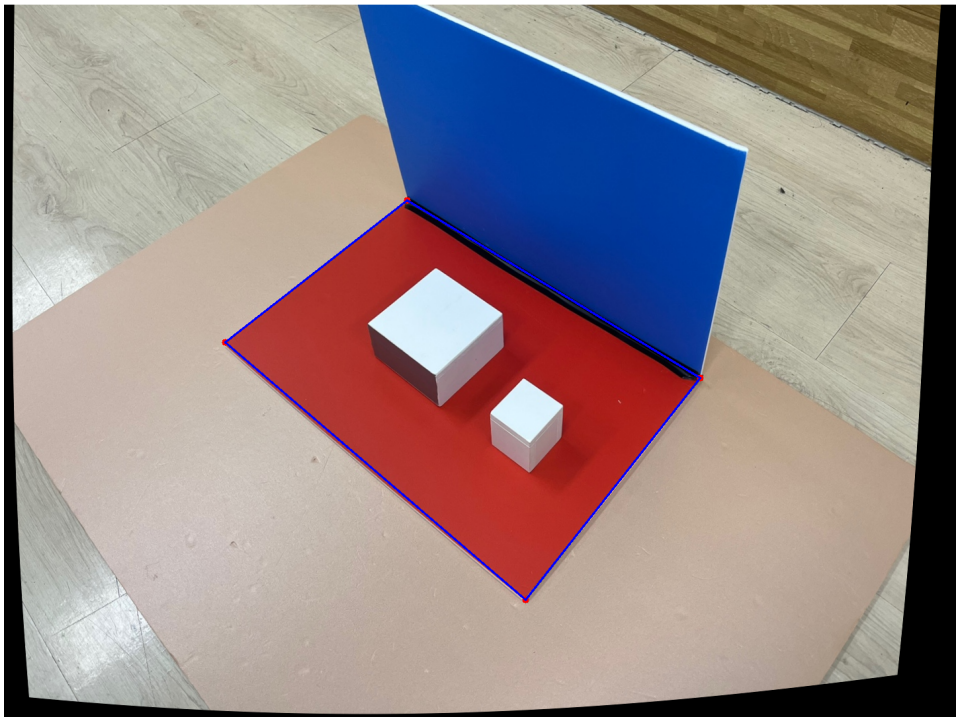
3. Procedure

1) Calibration

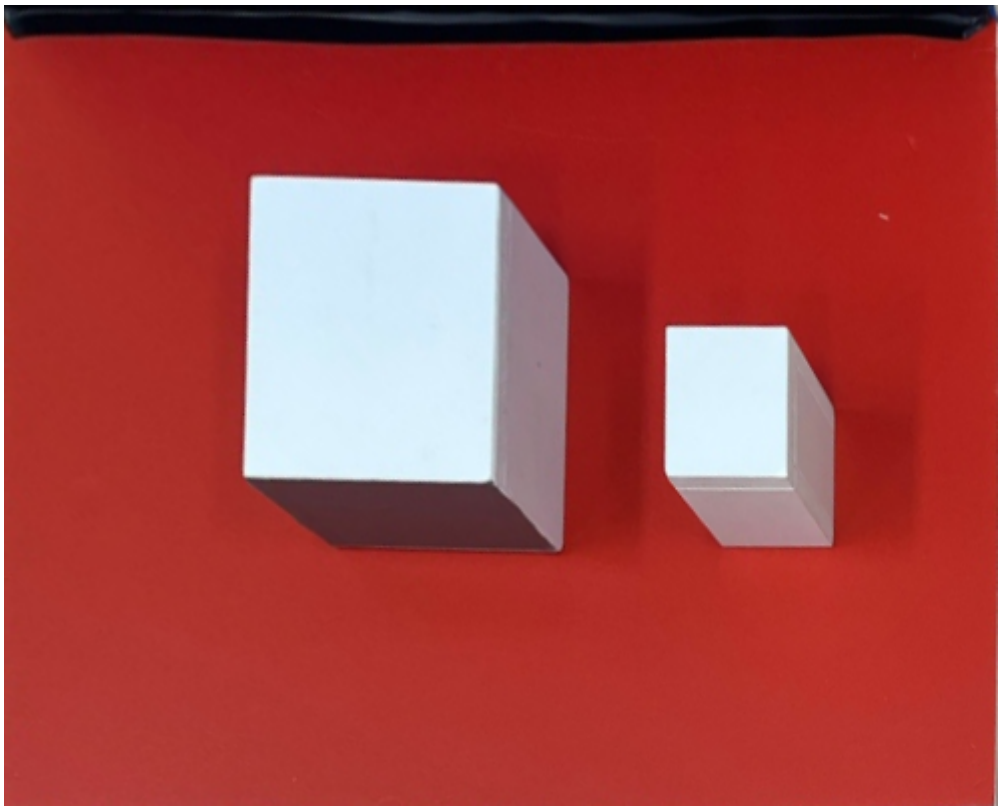


The calibration was performed as described above using an iPhone 13 mini.

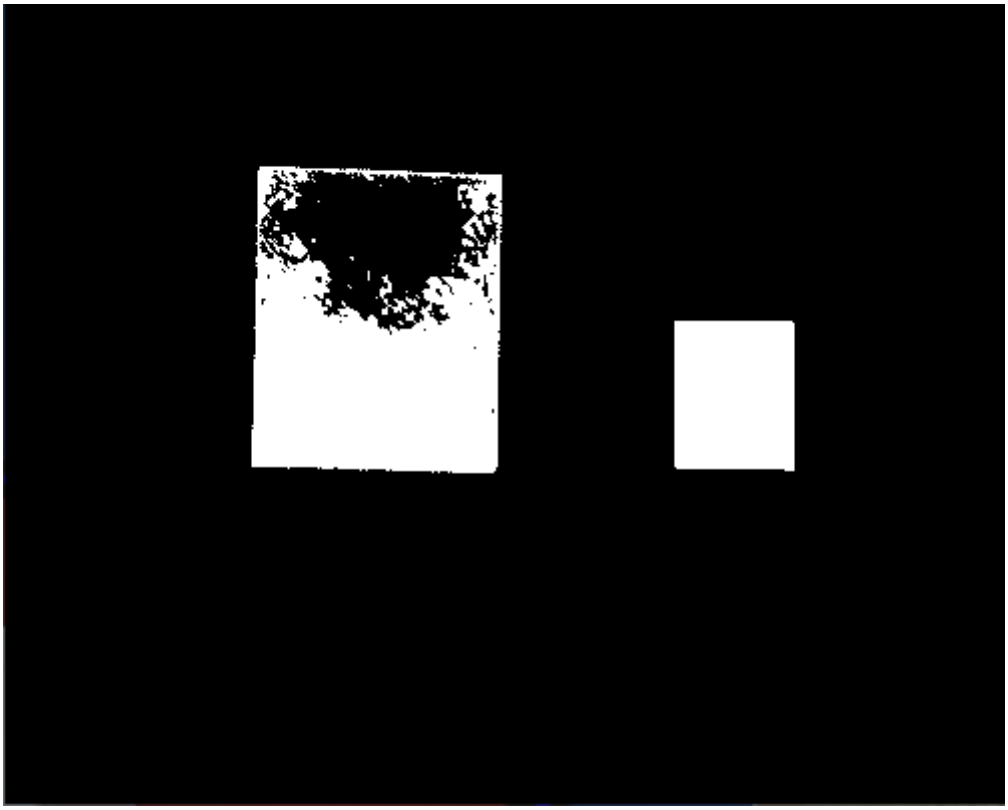
2) Warping

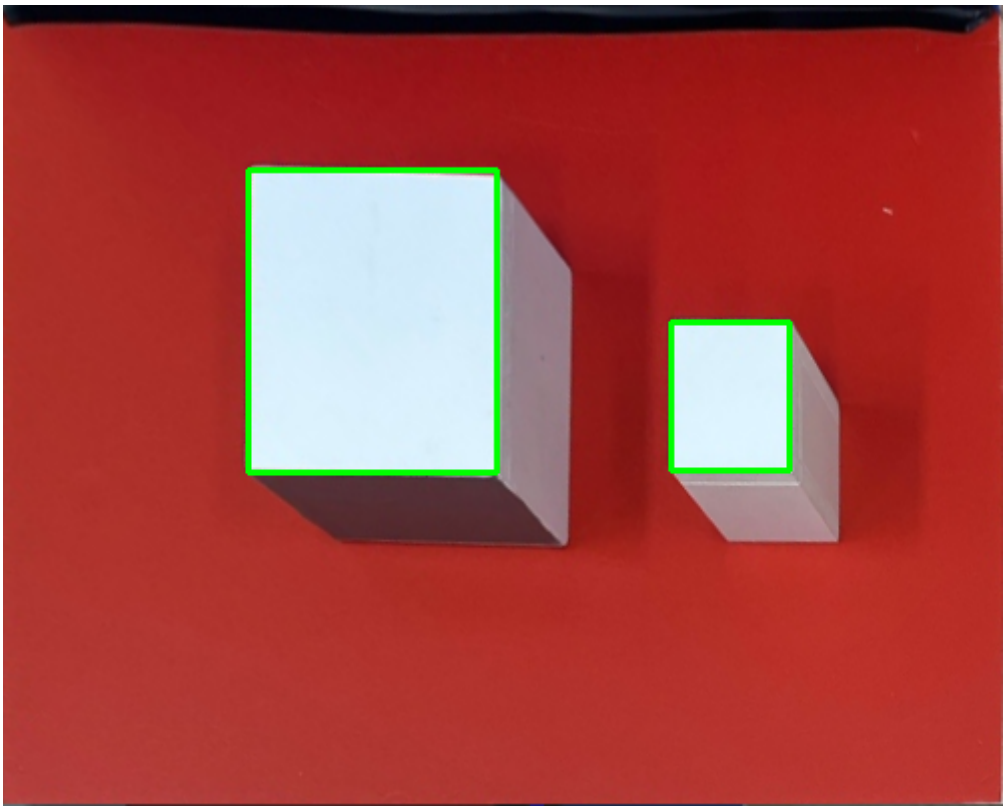


The user selects the four bottom vertices to be warped in the calibrated photo. The vertices are chosen starting from the top left and moving clockwise.



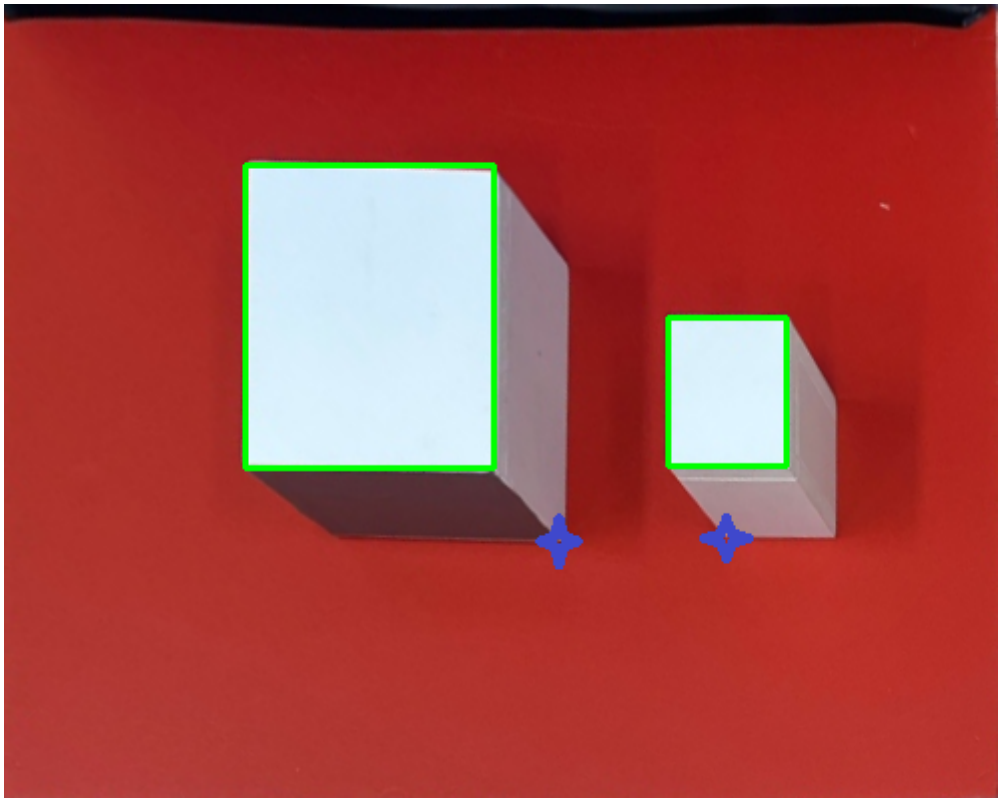
3) Thresholding and Contour





After converting the image to grayscale and applying thresholding, contours are identified as described.

4) Scale Conversion



In the photo, the user selects the bottom vertices of both the reference object and the target object. Since the dimensions (width, length, height) of the reference object are known, the first scale can be determined using the reference object as described.

$$Scale = \frac{Ref.height}{Object.heigth}$$

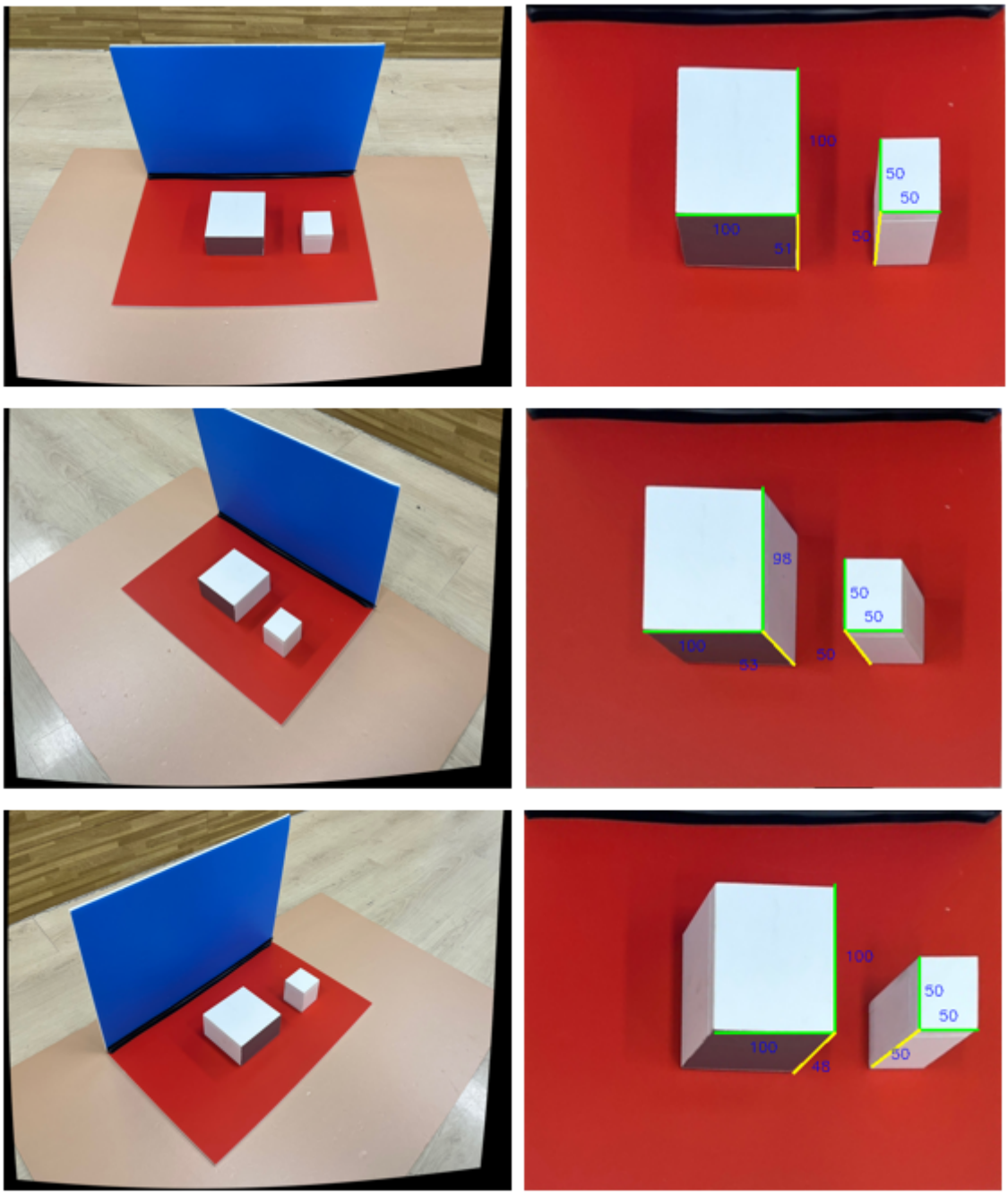
$$Object.height = \sqrt{(Refbox.x - Point.x)^2 + (Refbox.y - Point.y)^2}$$

Next, by using the given width value, you can find the True Scale using the formula provided. This allows you to apply a more accurate scale to your measurements.

$$True\ Scale = \frac{Given\ Width}{Object.Len} \times \frac{Object.height}{Ref.height}$$

III. Result and Discussion

1. Final Result



It can be confirmed that the error margin is within 3mm from all different angles as described.

Demo Video Embedded: https://www.youtube.com/watch?v=dX_aqF7gaDM

2. Discussion

Items	True (Width/Length/Height)	Estimated (Width/Length/Height)	Accuracy
Angle 1	100 / 100 / 50	100 / 100 / 51	99.6
Angle 2	100 / 100 / 50	100 / 98 / 53	98
Angle 3	100 / 100 / 50	100 / 100 / 48	99.2

This algorithm, although requiring the user to precisely define the warp range and the coordinates of the ground plane, has the advantage of not needing multiple photos to measure the width, length, and height of an object. Instead, it can measure all three dimensions from a single photo.

To implement this, the `warpPerspective()` function is used. Additionally, to help users more easily find the warp range, a red wooden block is placed.

As demonstrated, the results confirm that highly accurate measurements can be obtained, as shown in the table above.

Conclusion

The algorithm requires users to manually specify the warp range and enter the coordinates of the ground plane, which can be cumbersome.

To address this, the algorithm initially used HSV color detection to identify the warp area and the `approxPolyDP()` function to automatically detect the vertices of the edges for warping. However, using the `inRange()` function to segment the HSV colors proved to be very sensitive to the shooting angle, the shadows of the object, and the lighting conditions, often resulting in failed detections.

Attempts were also made to automatically detect the top and front surfaces of the target object by applying gray tape and using HSV to differentiate the colors. However, similar issues as mentioned earlier frequently resulted in unsuccessful detections.

To reduce user intervention and improve the robustness of the algorithm, it is crucial to develop an approach that is less sensitive to environmental variations. Implementing a deep learning model to recognize and locate objects could be a valuable strategy for enhancing the algorithm's effectiveness, making it more reliable under varying conditions.

Appendix

```
int main()
{
    // Calibration and read image file
    cameraParam param("lab2.xml");
    src = imread("ex3.jpg");
    dst_undistorted = param.undistort(src);
    if (src.empty()) {
        cout << "Could not open or find the image!" << endl;
        return -1;
    }
    namedWindow("Source Image", WINDOW_NORMAL);
    // Convert the image to Gray
    cvtColor(dst_undistorted, src_gray, COLOR_BGR2GRAY);
    setMouseCallback("Source Image", CallBackFunc, nullptr);
    imshow("Source Image", dst_undistorted);
    while (true) {
        char key = static_cast<char>(waitKey(0)); // Wait indefinitely for a
user key press
        if (key == 'c' || key == 'C') {
            clickedPoints.clear();        // Clear first set of points
```

```
            warpedImage.release();   // Release the image created by first
warping
            secondClickedPoints.clear();  // Clear second set of points
            pointSelectionStage = 1;      // Reset the point selection stage to
the first set
            imshow("Source Image", dst_undistorted);  // Redisplay the original
image
        }
        if (key == 27)
            break;
    }
    waitKey(0);
    return 0;
}
```

In the `main()` function, an XML file and an image file are loaded to remove any distortion from the image. This typically involves reading camera calibration data from the XML file, which contains parameters that are used to correct distortions in the image, such as lens distortions or perspective issues.

```
void CallBackFunc(int event, int x, int y, int flags, void* userdata) {
    vector<Point2f>* currentPoints = (pointSelectionStage == 1) ? &clickedPoints
: &secondClickedPoints;
    if (event == EVENT_LBUTTONDOWN) {
        selectedPointIndex = -1;
        for (int i = 0; i < currentPoints->size(); ++i) {
            if (norm(Mat((*currentPoints)[i]), Mat(Point2f(x, y))) < 5) {
                selectedPointIndex = i;
                break;
            }
        }
        if (selectedPointIndex == -1 && currentPoints->size() < 4) {
            currentPoints->push_back(Point2f((float)x, (float)y));
            std::cout << "Point " << currentPoints->size() << ": (" << x << ", "
<< y << ")" << endl;
            if (currentPoints->size() == 4 && pointSelectionStage == 1) {
                updateWarpedImage();
                pointSelectionStage = 2; // Move to second stage after first set
of points
            }
        }
    }
    else if (event == EVENT_MOUSEMOVE && selectedPointIndex != -1) {
        (*currentPoints)[selectedPointIndex] = Point2f(x, y);
    }
    else if (event == EVENT_LBUTTONUP) {
        selectedPointIndex = -1;
    }
    // Redraw all points
    dst_temp = dst_undistorted.clone();
    for (size_t i = 0; i < clickedPoints.size(); ++i) {
        circle(dst_temp, clickedPoints[i], 5, Scalar(0, 0, 255), FILLED);
        if (i > 0) line(dst_temp, clickedPoints[i - 1], clickedPoints[i],
Scalar(255, 0, 0), 2);
        if (clickedPoints.size() == 4) line(dst_temp, clickedPoints[3],
clickedPoints[0], Scalar(255, 0, 0), 2);
    }
```

```
    imshow("Source Image", dst_temp);
}
```

The `CallBackFunc()` function is used for users to specify the warp range. In this context, the function likely sets up a callback mechanism within a graphical user interface where the user can interactively select points on the image. These points define the region that will be transformed or warped. The function captures the coordinates of these points, which are then used to compute the transformation matrix for warping the image accordingly.

```
void updateWarpedImage() {
    if (clickedPoints.size() == 4) {
        int W = 500, H = 400;  // Arbitrary dimensions
        vector<Point2f> dstCorners = { Point2f(0, 0), Point2f(W - 1, 0),
Point2f(W - 1, H - 1), Point2f(0, H - 1) };
        Mat transformMatrix = getPerspectiveTransform(clickedPoints,
dstCorners);
        warpPerspective(dst_undistorted, warpedImage, transformMatrix, Size(W,
H));
        imshow("warpedImage", warpedImage);
        findBoundingRect(warpedImage);
    }
}
```

The `updateWarpedImage()` function is responsible for performing the warping of the image based on the points specified by the user. This function typically uses these points to create a transformation matrix, often through methods such as perspective transformation. It then applies this matrix to the original image to generate a warped image where the designated area is transformed to a new perspective, typically mapping a quadrilateral to a rectangle to simplify further processing or measurements.

```
void findBoundingRect(const Mat& image) {
    dst_result = image.clone();

    // change to grayscale
    cvtColor(image, gray, COLOR_BGR2GRAY);
    threshold(gray, thresh, 230, 255, THRESH_BINARY);
    // Finding contours
    findContours(thresh, contours, hierarchy, RETR_EXTERNAL,
CHAIN_APPROX_SIMPLE);
    if(clickCount == 0) setMouseCallback("warpedImage", onMouse, nullptr);
}
```

The `findBoundingRect()` function processes a warped image by converting it to grayscale and then adjusting the thresholding value to identify contours. This function typically involves several steps:

1. **Convert to Grayscale**: The warped image is first converted to grayscale, which simplifies the data by reducing it from three color channels (RGB) to a single intensity channel. This step is essential for the subsequent thresholding process.
2. **Apply Thresholding**: Thresholding is applied to the grayscale image to create a binary image where pixels are turned either black or white, based on a threshold value. This helps in distinguishing the foreground (the object) from the background.
3. **Find Contours**: Once the image is thresholded, the function uses contour detection algorithms to identify the edges of objects within the binary image. These contours can be

used to locate and measure objects within the image.

The result of this function is often a bounding rectangle around the detected contours, which provides the coordinates and dimensions of the object in the warped image, aiding in further analysis or measurement tasks.

```
void onMouse(int event, int x, int y, int flags, void* userdata) {
    if (event == EVENT_LBUTTONDOWN) {
        // Select Coordinate
        switch (clickCount) {
            case 0:
                box1x = x;
                box1y = y;
                break;
            case 1:
                box2x = x;
                box2y = y;
                break;
        }
        clickCount++;
        calculate();
    }
}
```

The `onMouse()` function is code that saves the coordinates where the user clicks on the ground plane.

```
void calculate(){
    if (clickCount == 2) {
        cout << "Box1 Coordinates: (" << box1x << ", " << box1y << ")" << endl;
        cout << "Box2 Coordinates: (" << box2x << ", " << box2y << ")" << endl;
        for (size_t i = 0; i < contours.size(); i++) {

            Rect boundingBox = boundingRect(contours[i]);
            // Restrict of boundingBox
            if (boundingBox.width < 40)
                continue;
            rectangle(warpedImage, boundingBox, Scalar(0, 255, 0), 2); // Draw a
bounding box in green

            if (boundingBox.width > 45 && cnt == 0) {
                // Calculate corner points of the bounding box
                Point2f topLeft(boundingBox.x, boundingBox.y);
                Point2f topRight(boundingBox.x + boundingBox.width,
boundingBox.y);
                Point2f bottomRight(boundingBox.x + boundingBox.width,
boundingBox.y + boundingBox.height);
                Point2f bottomLeft(boundingBox.x, boundingBox.y +
boundingBox.height);

                double knownWidth = 50.0;  // Desired width in actual units
                double knownLength = 50.0;  // Desired length in actual units
                double knwonHeight = 50.0;  // Desired height in actual units

                scale1 = knownWidth / boundingBox.width;  // Calculate scale1
based on the first contour
```

```
                scale2 = knownLength / boundingBox.height;  // Calculate scale1
based on the first contour
                scale3 = knwonHeight / (sqrt(pow(box1x - bottomLeft.x, 2) +
pow(box1y - bottomLeft.y, 2)));
                cnt++;

                line(dst_result, bottomLeft, bottomRight, Scalar(0, 255, 0), 2);
                line(dst_result, topLeft, bottomLeft, Scalar(0, 255, 0), 2);
                line(dst_result, bottomLeft, Point(box1x, box1y), Scalar(0, 255,
255), 2);

                // Calculate the size text
                string sizeText_W = to_string(static_cast<int>
(ceil(boundingBox.width * scale1)));
                string sizeText_L = to_string(static_cast<int>
(ceil(boundingBox.height * scale2)));
                string sizeText_H = to_string(static_cast<int>
(ceil(sqrt(pow(box1x - bottomLeft.x, 2) + pow(box1y - bottomLeft.y, 2)) *
scale3)));

                // Add the size text next to the lines
                putText(dst_result, sizeText_W, bottomLeft - Point2f(-20, 10),
FONT_HERSHEY_SIMPLEX, 0.5, Scalar(255, 0, 0), 1.5);
                putText(dst_result, sizeText_L, topLeft - Point2f(-5, -40),
FONT_HERSHEY_SIMPLEX, 0.5, Scalar(255, 0, 0), 1.5);
                putText(dst_result, sizeText_H, bottomLeft - Point2f(30, -30),
FONT_HERSHEY_SIMPLEX, 0.5, Scalar(255, 0, 0), 1.5);
            }
            else {
                // Calculate corner points of the bounding box
                Point2f topLeft2(boundingBox.x, boundingBox.y);
                Point2f topRight2(boundingBox.x + boundingBox.width,
boundingBox.y);
                Point2f bottomRight2(boundingBox.x + boundingBox.width,
boundingBox.y + boundingBox.height);
                Point2f bottomLeft2(boundingBox.x, boundingBox.y +
boundingBox.height);

                double distance = sqrt(pow(box2x - bottomRight2.x, 2) +
pow(box2y - bottomRight2.y, 2));
                double realWidth = 100.0;
                sc = realWidth / boundingBox.width; // scale1 of object about
width
                real_sc = sc / scale1;   // scale1 between reference object and
object that we wanted to calculate

                line(dst_result, topRight2, bottomRight2, Scalar(0, 255, 0), 2);
                line(dst_result, bottomRight2, bottomLeft2, Scalar(0, 255, 0),
2);
                line(dst_result, bottomRight2, Point(box2x, box2y), Scalar(0,
255, 255), 2);

                // Calculate the size text
                double actualWidth = boundingBox.width * scale1 * real_sc;
                double actualLength = boundingBox.height * scale2 * real_sc;
                double actualHeight = distance * scale3 * real_sc;
```

```
                string sizeText2_W = to_string(static_cast<int>
(ceil(actualWidth)));
                string sizeText2_L = to_string(static_cast<int>
(ceil(actualLength)));
                string sizeText2_H = to_string(static_cast<int>
(ceil(actualHeight)));

                // Add the size text next to the lines
                putText(dst_result, sizeText2_W, bottomRight2 - Point2f(90,
-20), FONT_HERSHEY_SIMPLEX, 0.5, Scalar(255, 0, 0), 1.5);
                putText(dst_result, sizeText2_L, topRight2 - Point2f(-10, -80),
FONT_HERSHEY_SIMPLEX, 0.5, Scalar(255, 0, 0), 1.5);
                putText(dst_result, sizeText2_H, bottomRight2 - Point2f(25,
-40), FONT_HERSHEY_SIMPLEX, 0.5, Scalar(255, 0, 0), 1.5);
            }

            resize(dst_result, dst_result, Size(500, 400));
            imshow("Result", dst_result);
        }
        clickCount = 0;
    }
}
```

The `calculate()` function finds the scale of the image, and using this scale, it displays the measurements of the object on the image with lines and text.