ingerso see the details of the ylel3D Scene Models from Images

**Mengqian Liu, Tianjiu Yin**

Contentingerso see the details of the ylelingerso see the details of the ylel

3D Scene Models from Images 1

1 Goal 2

2 Background 2

3 Methodology 3

3.1 Edge Detection 4

3.2 Structure Recognition 5

3.3 Camera Calibration 7

3.4 Ratio Calculation 9

3.5 3D Construction 9

3.6 Texture Mapping 9

4 Result, Analysis and Optimization 11

4.1 Result and Analysis 11

4.2 Optimization 12

5 Mobile Application 12

6 Future Work 15

7 Conclusion 16

8 Difficulties and Solutions 16

9 Reference 16

# 1 Goal

The goal of our project is to build an application that can reconstruct an accurate 3D model of an arbitrary cuboid given the 2D image of it. The user can get the 3D model simply by taking a picture of the cuboid, and then the user is allowed to manipulate the model in 3D (changing the viewpoint, zoom in and out).

The problem our project tries to solve is very interesting since it will be very convenient if users can get the virtual 3D model of a real scene anywhere they want with their mobile devices. It is also very fundamental because after the model is generated, there are a lot of applications that can be implemented on the model, such as digital museum, virtual reality, digital city, etc.

# 2 Background

In recent years, the 3D reconstruction based on single view image or multi-view images attract a lot of attention. There has been much research on multi-view reconstruction by getting the depth map of scene, which leads to very accurate reconstruction. However, multi-view method has its own disadvantage when applied to mobile applications because it needs correspondence between images, which increases the calculation workload of devices. However, the calculation speed and storage space are very limited on mobile devices. Also, users need to make more effort to take pictures repeatedly. So we chose single view method to approach our goal.

3D reconstruction based on single view metrology requires measurements on affine 3D geometry. A classic method to do this is discussed in [1]. This method is able to compute the distance between planes parallel to the reference plane by using a vanishing line of the reference plane and a vanishing point for a direction not parallel to the plane. The limitation is that it can only get the length ratio in one direction. If we want to implement this method to get the length ratio in three directions, we need to know more reference distances in advance, which is not ideal in the application. [2] provides a method to do single view metrology along orthogonal directions while getting the camera intrinsic matrix without extra camera calibration. However, there are many approximation used in the method, which increase the inaccuracy of the result. Other methods require the application to do camera calibration before measurement using chessboard of circle array pattern, which also limit the function of the application. So we proposed a single view metrology method that uses the special geometric features of cuboid, which will be discussed in the next section.

# 3 Methodology

The input of this project: an image of a cuboid object taken from anywhere.

The output of this project: a 3D model of the cuboid including its texture.

Assumption: single cuboid, relatively strong contrast between cuboid and background.

The flow chart below shows our methods to solve this problem:



In this chapter, we will discuss about how to decompose this challenging problem into 6 pieces and solve them one by one.

In this report, we will use this image as input and show the process of this image in detail step by step.



## 3.1 Edge Detection

The first step is to detect edges of cuboid in an image. Opencv provides canny function to detect all possible edges in a given image. But in this project, it’s hard to utilize this method directly because we are only interested in the edge of cuboid only. We should ignore all possible lines detected either in background or in texture.

To solve this problem, we decided to use flood fill and opencv Houghlines function to detect cuboid objects. The advantage of this method to canny function is that we can use flood fill to differentiate background and cuboid easily so that we can get rid of irrelevant lines. After we get the edges of the cuboid, we can get lines of cuboid edges using Houghlines function.

But there are still some problems to consider:

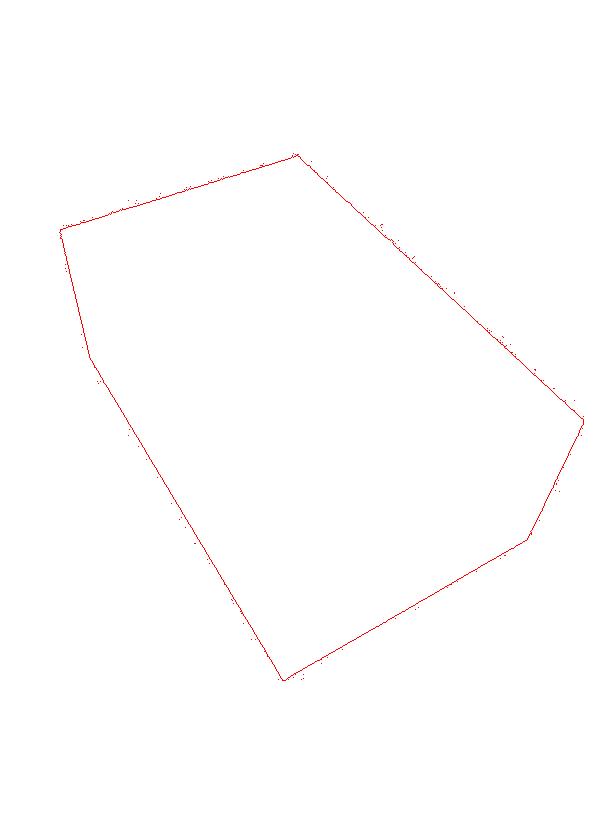
1 how to flood fill background since background is not pure color?

2 the Hough transform will detect much more lines than needed, how to choose approximate edge line function?

To solve 1st problem, we need to determine the distance of color. After research, we found that LAB color space represent the distance of color much better than RGB. So, in the following steps, we will always transform input image from RGB to LAB before processing.

But it still need more careful consideration, in different situations, the background of image can be relatively close or different. It means that the distance of background color can vary between different input images. To solve this problem, we develop two modes in edge detection, fixed and variable background color distance. We will talk more about this in Chapter 5 Mobile Application in detail.

The second problem is even trickier and we will talk about this in Chapter 3.2 separately. So far, the image after processing will look like this:



## 3.2 Structure Recognition

After edge detection, the input image will be converted into image with boundaries of cuboid. But since there is too much noise in an image, we can hardly get extract cuboid lines we expect from Houghline function. The boundaries of processed image aren’t even straight if we take look at it in detail. The following figure shows this problem:



If we use Houghline function, we will get a lot of short lines instead of a complete line of cuboid edge we expect. We come out a simple way to solve this problem. Our goal is to divide all these short lines into different groups to which the edge they belong to and compute average line function based on all lines in group. So the problem would be how to group all detected lines.

We solve this problem in this way: first we randomly chose a point inside cuboid. Then, we compute the perpendicular vector from this point to every detected line. Finally, we can group all the lines according to vector angles easily since we transform 2D line functions problem to 1D angle problem. The pseudo code would be:

1 randomly choose a point p

2 for detected line

3 compute perpendicular vector

4 compute the angle of vector

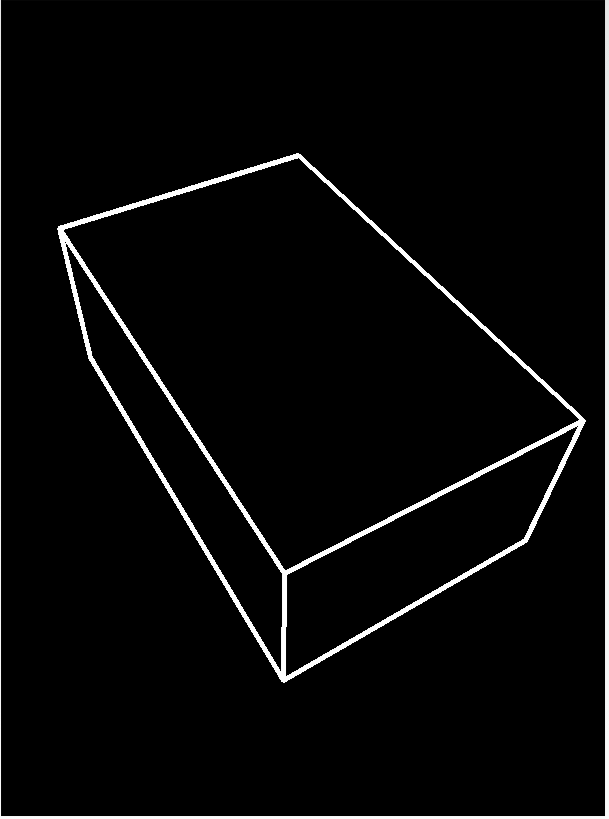
5 sort

6 compute angle difference

7 select K largest difference, K is the number of edges

8 group all line into K groups according to K largest difference

After this step, the processed image will show the structure of input cuboid, the processed image is shown as follows:



Note that the three edges within the contour are calculated instead of detected. When three lines that are parallel in real world are projected to the image plane, they will intersect at a vanishing point. Thus, after we get two parallel lines, we can calculate the third line easily.

## 3.3 Camera Calibration

After two preprocessing steps of 3.1 and 3.2, we can get the exact edge and vertex coordinates from input image. Before we calculate the ratio of cuboid edges, we need first calibrate the camera. Since our target is cuboid, we can get camera calibration information from input image as well.

The classic way to calibrate camera is to compute projection matrix:

But the problem is that this method either needs much more rectangles than we have in a single cuboid image or need approximation. After trial of this method in this project, we found that the result is too coarse to use.

After researching on this problem, we found a new way to calibrate this problem:

persperctive2.eps

In fact, we can get camera calibration parameters from geometric primitives. From the image above, we can see the principal point (*u*, *v*) is the projection point of view point on image plane and focus length is distance between viewpoint and image plane.

In this problem, we found that the parallel line in object plane is parallel to the line connecting viewpoint and its corresponding vanishing point on the image plane. And the three edges on object plane are perpendicular to each other. So the line connecting viewpoint and three vanishing points are perpendicular to each other as well.

According to this discovery, we can compute viewpoint positions according to following equations:

Where *v1*, *v2*, *v3* are the distance between 3 vanishing points, and *a, b, c* are the distances between viewpoint and vanishing points. Given position of viewpoint, we can easily derive (*u, v*) position and focus length as well. So, in this way, camera can be calibrated without approximation.

## 3.4 Ratio Calculation

According to above figure and analysis in Chapter 3.3, we can get ratio between different edges as well.

Since we already get position of viewpoint in Chapter 3.3, we can compute the direction vector of line between viewpoint and vanishing point, as well as their corresponding direction vector of parallel line in object plane. We can assume the position of one vertex. According to direction vector, we can compute all the positions of other vertices easily. After getting all the coordinates of vertices, the ratio between different edges can be derived directly.

According to experiments, the result of this method is quite accurate. We will discuss more about the results in Chapter 5.

## 3.5 3D Construction

In this step, we use OpenGL ES 2.0 to construct 3D model. Since we know that the object that we want to reconstruct is cuboid, we predefine a cuboid mesh with variable height, length and width by multiplying three scalars to the coordinates of the cuboid vertices in x, y and z direction separately. After we get the ratio from the former steps, we change the scalar accordingly. In this way we can construct different kinds of cuboid easily.

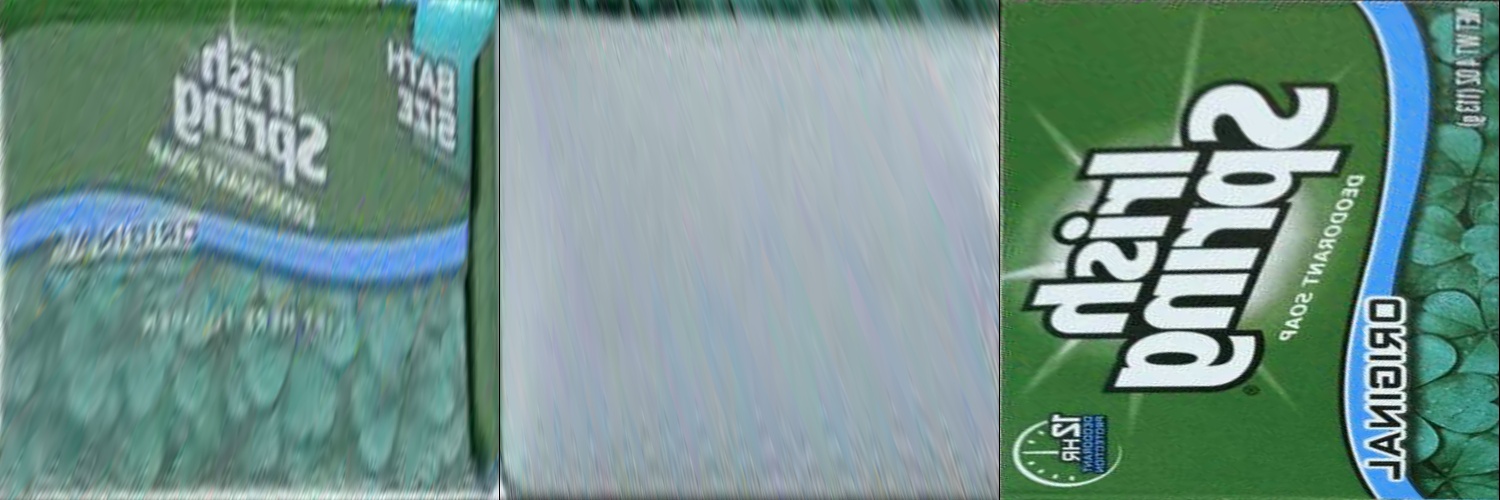
## 3.6 Texture Mapping

After we know the positions of the cuboid’s vertices in the image, we can get the texture of each surface as is shown in the figure below:

However, we cannot replace the texture coordinates in OpenGL with the coordinates of the vertices in the image directly, because OpenGL render the model in unit of triangle. If we do texture mapping from an irregular shape texture to a rectangular model surface, the texture will be distorted instead of stretched uniformly to fill the rectangular area. Because the rectangle is separated into two triangles, and the stretch directions in the two triangles are different. So only when the texture source image is also rectangular (does not need to have the same width/length ratio with the model surface), can it be assigned to the model correctly.

So what we need to do is stretch the area that is used as the texture source image into rectangle, and then align them in to one image. Here we use OpenCV Imgproc.warpPerspective function to fit the irregular shape into a triangular area with fixed size. The result is shown in the figure below:



Next, we can fix the texture coordinates. We only need to change the texture image source every time we build a new model.

# 4 Result, Analysis and Optimization

## 4.1 Result and Analysis

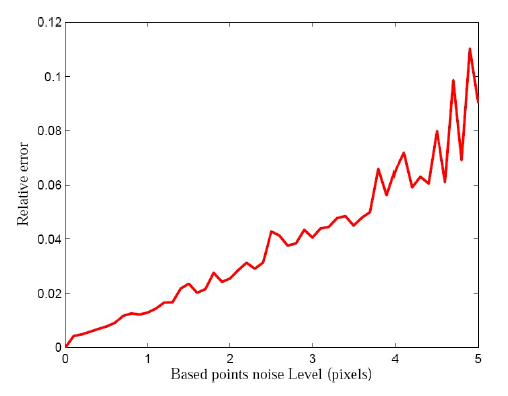
The measurements of input cuboid:

|  |  |
| --- | --- |
| Length | 9.8 cm |
| Width | 6.2 cm |
| Height | 3.8 cm |

The comparison between the actual measurements and computed results according to Chapter 3.3 and Chapter 3.4 is show below:

|  |  |  |  |
| --- | --- | --- | --- |
| **#** | **Real** | **Result** | **Relative Error** |
| **Length/Width** | **1.58** | **1.578** | **0.1 %** |
| **Width/Height** | **1.63** | **1.577** | **3.2 %** |
| **Length/Height** | **2.57** | **2.490** | **3.1 %** |

The method has excellent performance in computing the ratio between length and width. The relative error is much larger involving height. The reason might be that the height is much shorter than length and height. Therefore, the noise in height might cause higher error rate.



Compared to research[2] , our method can exceed it to some degree.

## 4.2 Optimization

Since we use flood fill to determine the edges of cuboid, it’s very time-consuming to flood fill a very large image taken by smart phone. The initial implementation of this method took about 20s to process edge detection and it will easily result in out of memory. But after optimization, the final implementation was able to detect cuboid object within 200ms and output results in live camera frame.

The main three aspects of optimization is as follows:

1 Flood Fill algorithm

We improve this algorithm by filling the whole image in line basis instead pixels basis. We also improve performance a lot by implement the algorithm in native code, written by C++.

2 Memory optimization

In order to avoid out of memory problem, we refract our code to minimize bitmap and mat construction and copy. We also manually recycle these variables as soon as they are useless before Java automatically recycle it. The most important approach is passing native address of mat to native code so that we don’t have to copy memory when using native code.

3 Variable accuracy

In order to achieve better user experience, we improve our algorithm so that it can compute result in different accuracy level. In that case, users can choose less accurate results in live frame but faster output.

# 5 Mobile Application

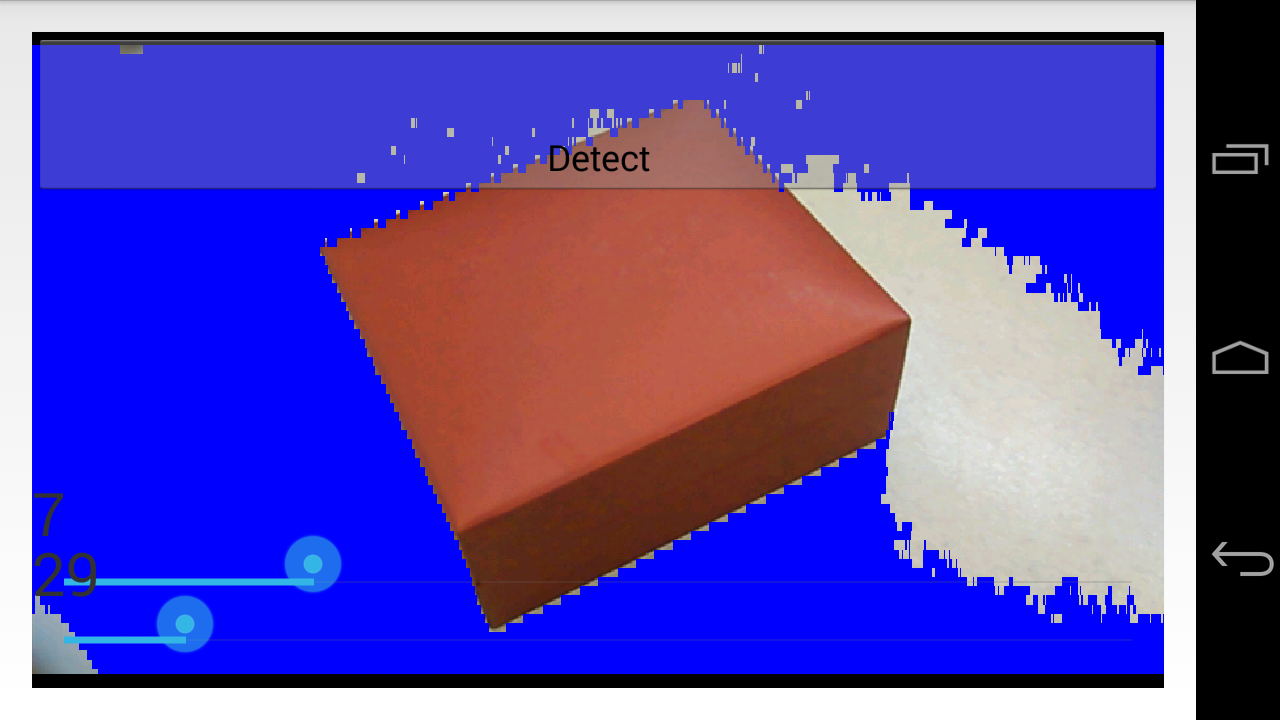
Our application is composed of four main parts: input image, edge detection, geometrical calculation and 3D model display.

1 Input Image

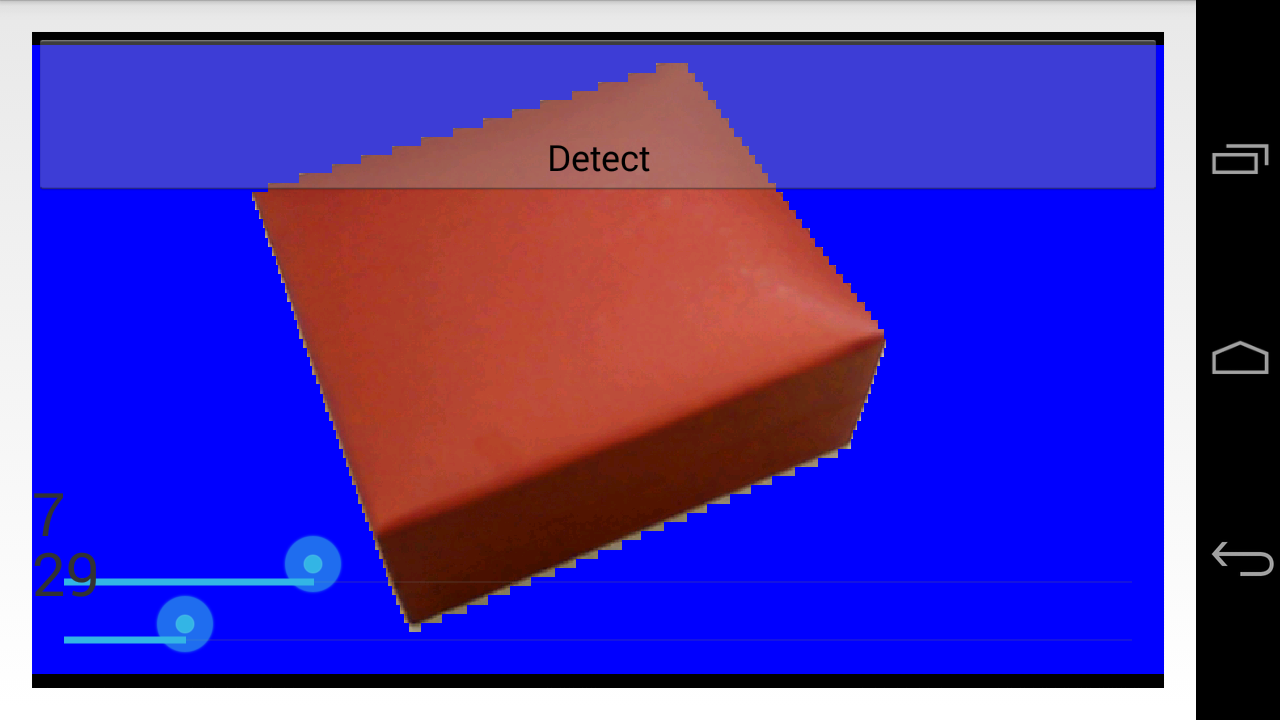
In this step, users can see the following interface:



There are three options provided in this activity. It is recommended that users first click the “Detect” button to calibrate the parameters needed in the following recognition step. After pressing “Detect”, users can see the interface below:



There are two sidebars that allow users to adjust the accuracy and the distance of background color. As mentioned in Chapter 3.1, the best distance to distinguish the background and the object varies in different situation (object and background color, lighting condition, etc.) So users need to adjust the sidebars until the object is extracted correctly as follows:

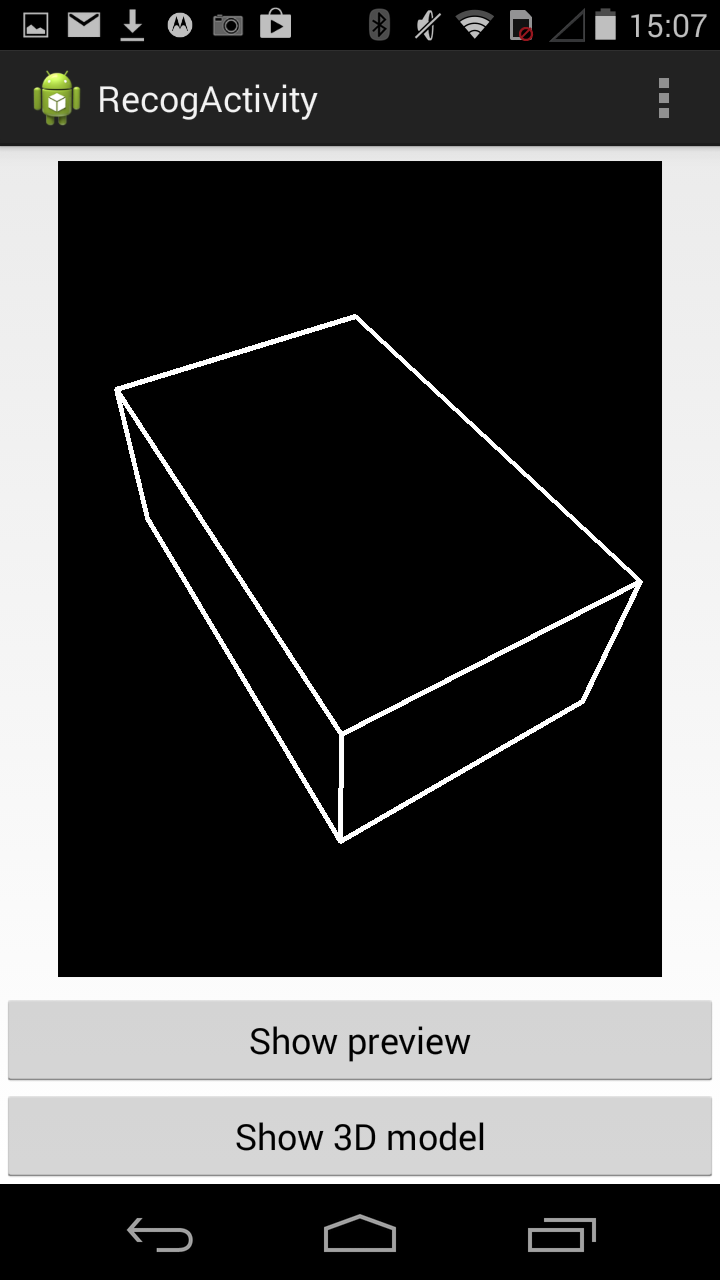


Then users can press “Detect” and go to the next activity. The present camera preview is saved for edge recognition.

Also, in the first activity, users can click “Take picture” to take a picture directly without calibrating the parameters or choose a existed photo from the gallery.

2 Edge Detection

In this part, the application will do edge detection and structure recognition to get the exact edge and vertex coordinates of the cuboid from the input image. Then the application will show the detection result as follows:



Users can click “Show preview” to go back to the initial activity of click “Show 3D model” to see the 3D model.

3 Geometrical Calculation

After getting the 2D information of the cuboid, the application will do camera calibration and ratio calculation to get the ratio between the width, length and height of the cuboid.

4 3D Model Display

Using the ratio we get in Geometrical Calculation part, the application will call OpenGL ES to draw the 3D model on the screen. The transformation of the texture is also done in this activity using the method in Chapter 3.6. The following interface is what users can see finally:



Users can rotate the virtual 3D model by moving their finger on the screen and zoom in / out to see the details of the model using two fingers.

# 6 Future Work

The future work of our project is based on our assumption:

1. We would like to apply to our project from single cuboid to multiple cuboids in one image.

2. We would like to enhance our algorithm so that it can recognize cuboid object even its color is close to background color

3. We would like to enlarge our project application areas so that it can build 3D models for more kinds of objects like ball, cylinder, etc.

# 7 Conclusion

1. In specific image processing problem, we can solve difficult problem in a simple and accurate way by taking advantage of its geometric primitives.

2. In developing real applications, we should balance between accuracy and computation speed. It’s also very important to provide user smooth experience.

3. In developing mobile imaging applications, we can get better results by integrating user interactions.

# 8 Difficulties and Solutions

In the beginning, we tried to implement the method in [2]. However, the result we got was very inaccurate so we decided not to use this method, despite the fact that we have already devoted a lot of time to it. Then after searching for many papers and websites, we still could not find a satisfying way to implement single view metrology to reconstruct cuboid, especially under the limitation of mobile devices. Then we realized that we could not rely on the solutions of others, what we should do is specify our unique situation and proposed a new method by our own. Luckily we came up with the algorithm introduced in Chapter 3 and it feels so good to finally change our idea into a functional application rather than repeating other’s work.

# 9 Reference

[1] Criminisi A, Reid I, Zisserman A. Single view metrology[J]. International Journal of Computer Vision, 2000, 40(2): 123-148.

[2] Peng, Kun, et al. "Single view metrology along orthogonal directions." *Pattern Recognition (ICPR), 2010 20th International Conference on*. IEEE, 2010.