VirtuaLego

Final Project Report

Introduction to Digital Image Processing

Noam Brickman 203317979

Yael Schor 208937615

Table Of Contents

Abstract:

VirtuaLego project is a system for real-time 3D lego model detection and a related computer game.

The system transfers the physical 3D lego model, built by a user on a dedicated lego board, into a virtual structure using image processing methods.

The related game contains levels with virtual character that should cross from the beginning to the end of a track, while avoiding obstacles. The path for the character is built by the 3D modeled lego structure (figure 1).

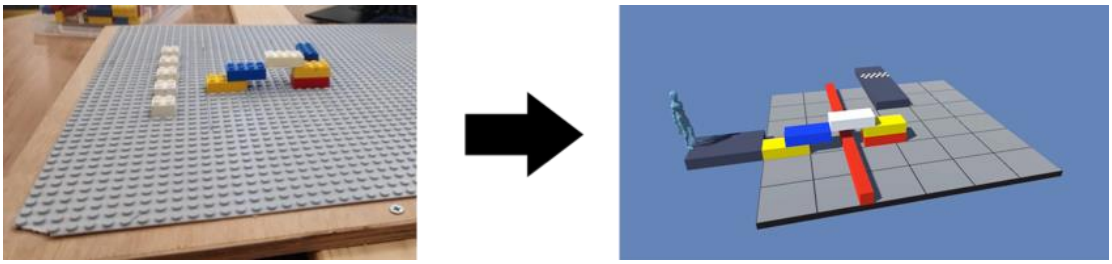


Figure 1: Demo of the system flow. From a physical model to a virtual model.

Project description:

Our system contains two main components - the 3D model detection component, and the virtual game component. We will describe each of those components and the interface between them.

The 3D model detection component is made of a physical structure, connected to a Matlab based algorithm.

The physical structure is a wooden stand with a lego board in the center and 4 web cameras mounted at the top, right, left and back sides of the lego board (figure 2). During this paper we named the right, left and back cameras as "side cameras".

Figure 2: The wooden device, with the 4 web-cameras marked in red circles.

The algorithm controls the cameras in order to detect and analyze the model.

The interface of the user with the system is done using 5 buttons made of lego cubes on the lego board. Putting or removing fingers from the buttons make them pressed or released, using image processing method.

The game component built in a Unity software, which is a platform for games development. In the virtual game, there are several stages which each of them defines a track with a starting point, ending point, and obstacles between them. The user should build the physical model lego, which appears in real time inside the game. When a valid path between the starting and ending points exists, the character crosses the track and the level completes.

The communication between the two components is done using a text file (.txt file). The Matlab's algorithm analyzes the lego model using the cameras and outputs a message to the text file, contains information that need to be shown in the game component (figure 3).

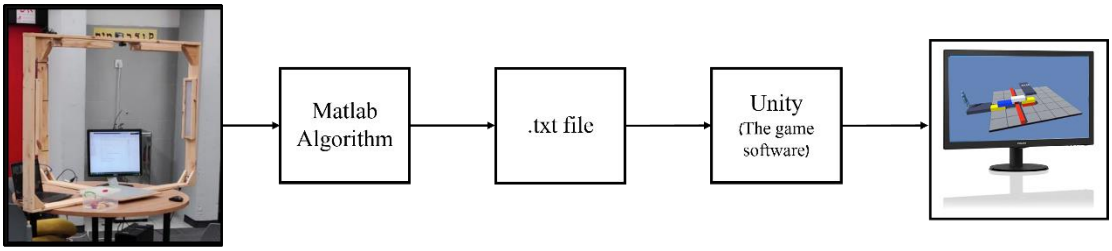


Figure 3: The interface between the system's components.

All the image processing is done only in the Matlab algorithm component, and the virtual game is made only for user interface and user experience.

The image processing algorithms we used in this project contain methods we learned along the course, including geometric transformations, binary masks, color spaces, lines detection, triangulation between multiple cameras and more. Implementation of those methods in our system will be described below.

For a demo video of the project - the overview, building process, and a live demonstration - visit the link or scan the QR code in figure 4.

Figure 4: QR code for a demo video.

System presentation:

In this section, we will explain about the 3D model detection component of the system.

The system in managed by a GUI (figure 5), made in Matlab, with a menu to the relevant functions in the system (calibration, start system and more). This menu is not for the user, but for the system operator only.

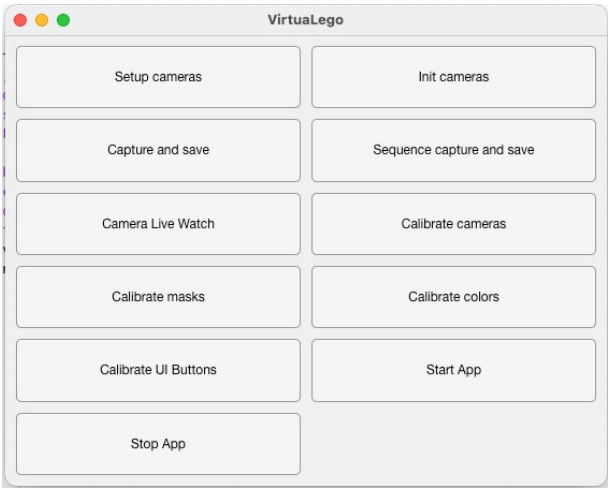


Figure 5: Screenshot of the system's GUI.

This component contains an algorithm, written in Matlab, which analyzes the inputs from the 4 cameras. We present the algorithm in 2 blocks diagrams (figure 6, 9), where the first diagram (figure 6) is the main system loop, and the second diagram (figure 9) is a zoom-in on the "state analysis" blocks in the first diagram.

We will explain about each block of those diagrams - principles of work and included image processing algorithms.

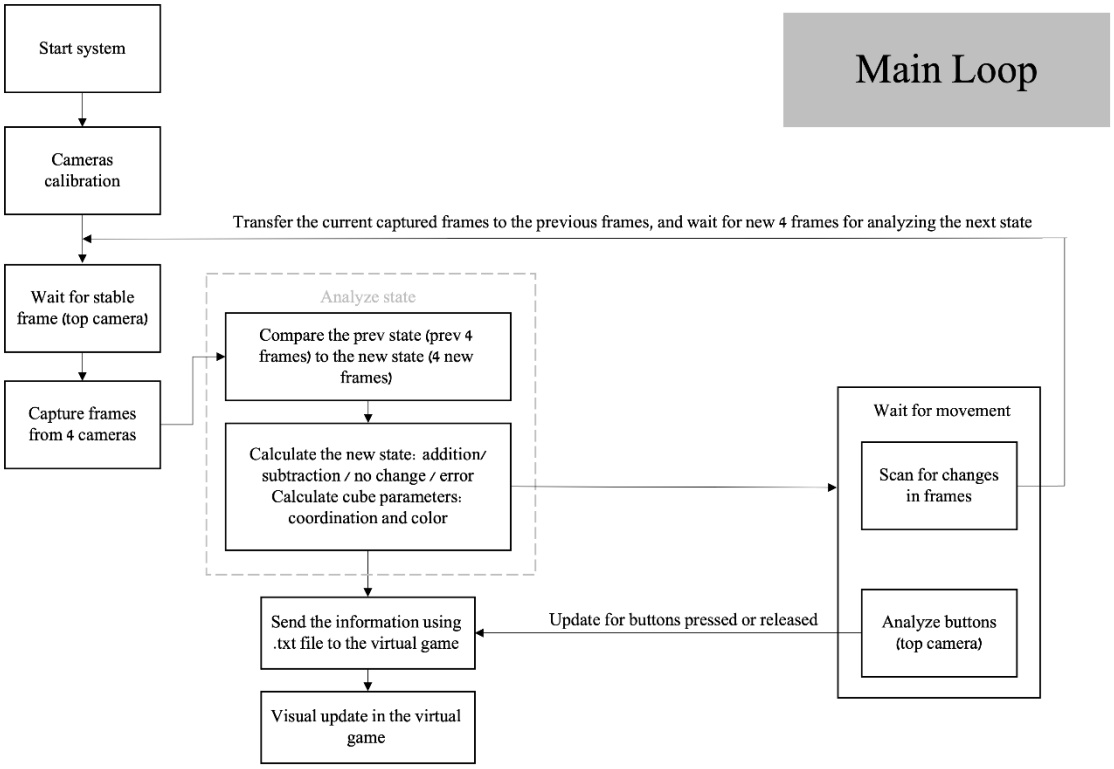


Figure 6: System blocks diagram – Main Loop.

Main Loop:

The main loop goals are:

- Initial calibration (done once at system lunch).
- Generating sets of 4 images (one from each camera) of two different time steps (before and after a physical lego model changing - for example: adding or removing a cube from the model).
- Analyze buttons state.
- Sending commands to the interface text file.

Description for each block:

Cameras' calibration:

The cameras calibration contains 4 different calibrations. This operation needs to occur once, after changing the system location, moving of the cameras or environment changes like a change in background lights.

All the calibrations are done using Matlab's built-in 'ginput' function, and its output is used in different manners for each calibration type, which will be described below. At the end of each calibration, the final calibrated parameters saved in mat files and loaded upon request using the algorithm.

The types of calibrations are:

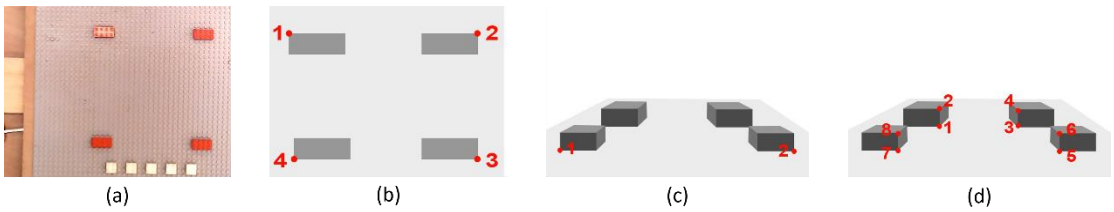
1. Image straightens:

This calibration meant to handle cameras physical movements and also having relevant measurements for the lego cubes coordinate estimation. In order to calibrate the cameras, we firstly need to locate 4 lego cubes in specific coordinates on the lego board, which define the lego board edges (figure 7a). For the top camera, we sample the 4 corners using 'ginput' command (figure 7b). Later, using those corners, we are straight the frame and crop the relevant area from the frame (using geometrical transformation, as describe in the "Algorithms" section).

The type of transformation is 'projective', where the source points are the sampled ones, and the target points are 4 pre-defined coordinates. The type of the transformation selected to be 'projective' since we want to fix also perspective distortions if exist, and those are done only by using this type of geometrical transformation, which require at least 4 points.

For each side camera (right, left and back), we do the calibration process in two steps. First, we sample 2 points on the left and right sides of the frame. Using those samples, we tilt and stretch the frame to fit two pre-defined coordinates (figure 7c). This transformation is also done using the 'estimateGeometricTransform2D' command for geometric transformations. In this case, consider the need for this kind of straighten, we used the 'similarity' type of transformation (further explanation can be found in the "Algorithms" section). This type requires at least 2 source and target points for being calculated.

At the second stage of this calibration, we preform the last calculated transformation from the first stage (for sampling on the straight image) and then sample the base and the top of each of the four calibration cubes (figure 7d). Those samples enable us to process each camera's perspective for estimating the cube's coordinates.



Figures 7: (a) The calibration cubes (b) Calibration for top (c) Calibration for side - first step (d) Calibration for side - second step.

2. Camera masks:

Inside the field of view of the side cameras, behind the board, there is also a significant area of the background. Movements of people or objects in this area can fool the system since the system is based on comparison between frames, and pixels that changed in the background could accidently be treated as cubes. As an initial solution for this issue, we generate a 5-points polygon binary mask for each camera, where all the comparison binary maps are masked with this polygon. In that way, changes in this area do not affect the detection system (figure 8).

In order to do that, this calibration mode enables to capture (using ‘ginput’) the 5 points for each polygon, for the 3 sides cameras, and store them for the polygon generation in a further stage.

Figure 8: A 5-points polygon camera mask.

3. Colors calibrations:

Our system predicts the color of the cubes during the algorithm process. Although we have built-in lighting on our wooden stand, we affected a lot by the background lights.

Therefore, in order to adapt to the current environment, we calibrate the colors of our cubes.

For doing that, we first locate cubes in all the optional colors, on the board. Then, from each of the 4 cameras and for each cube's color we sample using ‘ginput’ 4 different pixels on the cube area. After that, we store the average RGB value (a 1x3 vector) as calculated from the 4 sampled pixel of each color. The reasons for selecting RGB color space will describe in the “Algorithm” section.

4. Buttons calibration:

In similar to ‘Image straightens’ calibration and using exactly the same method presented for the top camera, we store the 4 corners of each of our 5 buttons. We do that for straightening and cropping each button area for checking rather its pressed of released.

Wait for a stable frame:

At this part of the system loop, we need to wait for detecting stability in the lego building area in order to capture frames from all cameras. We do that since we cannot allow taking frames with hands or other elements which are not lego cubes in the field of view.

For that, we use the top camera to detect a stable frame. The stability check we developed is based on comparison of changes between a frame to a memory of frames.

A stable frame, in our context, defined as a frame which ranked as “stable” compared to all the frames in the memory. The comparison to the memory is needed, because a very slow movement of a hand inside the scene can be detected as stable if we would base on a single comparison. On the other hand, too long memory increases computation time and cause large delay in the system. The final selected parameter is memory length of three frames.

For considering two frames as “stable”, we make a comparison between them. We make a map of the difference between the images by subtracting the images channel-wise, then summing the absolute value of the 3 color channels (R, G, B) into a single channel, and normalizing the result into [0,1] range. From that, we binarize the result with threshold of 0.2 using the ‘imbinarize’ built-in function and removed blobs that have area smaller than 200 pixels using ‘bwareaopen’ built-in function, since we consider them as noise.

Using the binary map, we sum the total pixels that have the value 1. If the sum is less than a threshold of 200, the frame considered as stable. Otherwise, it considered as unstable.

Capture frames from 4 cameras:

After having a stable state, we capture frames from the 4 cameras. Each camera produces an image in [480x640x3] pixels resolution.

Analyze state:

Using the current and the previous sets of four images, we analyze the combination of 8 images in order to detect what is the building step that has been made (addition / subtraction / no change detected). Moreover, we detected the cubes' parameters, such as coordinates and color. This process is shown in figure 5 and described in detail after the main loop description.

Wait for movement in scene:

After the state analysis, we would like to wait until some change in the physical model occurs. For that, we enter a loop that does not break until we have some movement in the building area. During this waiting for movement loop, the system is also analyzing the buttons state. Those 2 functionalities describe below.

<u>Scan for changes in frames:</u>

We calibrated the movement detection thresholds such that it can detect a hand getting into the building area (and not been infected by noises)

In the state, we compare each frame to the previous frame and looking for a significant difference between the frames. After having three comparisons in a row with movement detected (3 comparisons to eliminate significant noise), we define movement in the scene and break the waiting loop.

The two frames comparison is made by similar a concept to the stability checking that described above. This time, we want to make sure that the total area of the binary map is in a range of [2000,100000] to define movement between those frames. The upper threshold is made for eliminating light differences that cause significant change in all pixels in the binary map, and do not pointing at a real movement on those frames.

Analyze buttons:

At the entrance to the wait for movement block, we take reference images of the 5 buttons in their empty state (not pressed). Since then, we compare the current image of the buttons (after they straighten using the geometric transformation and cropped) to the respective button reference image. Using a binary map, in similar concept to the binary maps that described above, we identify if the button is pressed or released.

We decided to take the reference image on every time we get inside the waiting loop, and not a single time at the system startup, since we found out that light changes caused all binary maps to be active, and all the buttons detected accidentally as pressed. From that, we understood the importance of comparing images with new reference images each time.

Send the information to the text file:

The result of the state analysis, so as buttons pressing and releasing events, are added to the text file in a pre-defined message format that is readable by the game software.

Visual update in the virtual game:

The text file is read by the game software. The software shows the cubes and button statuses following the messages from the text file.

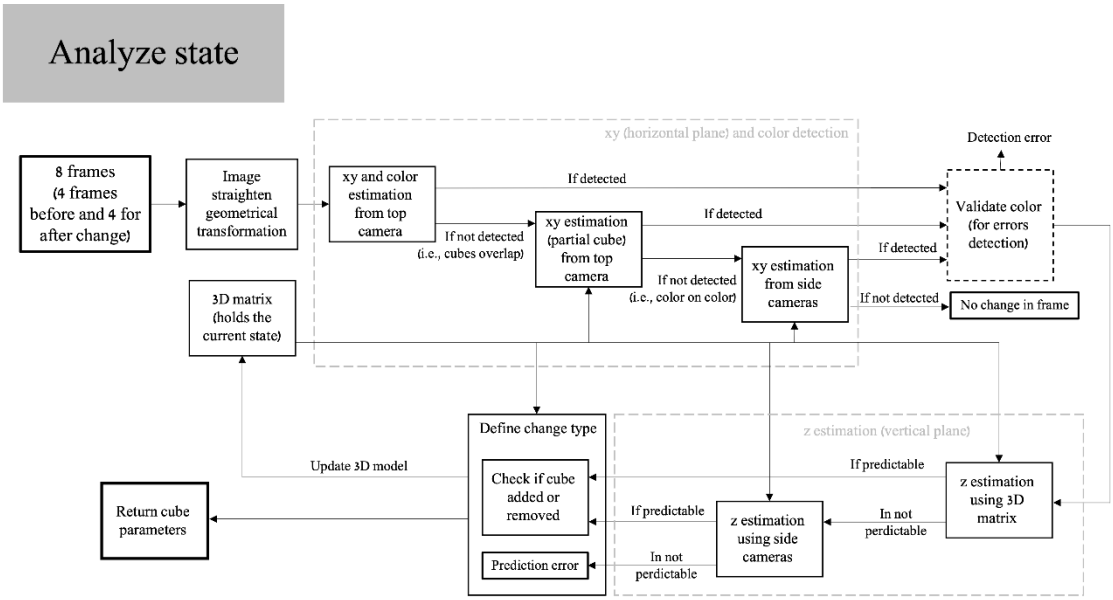


Figure 9: System blocks diagram – Analyze State.

Analyze state:

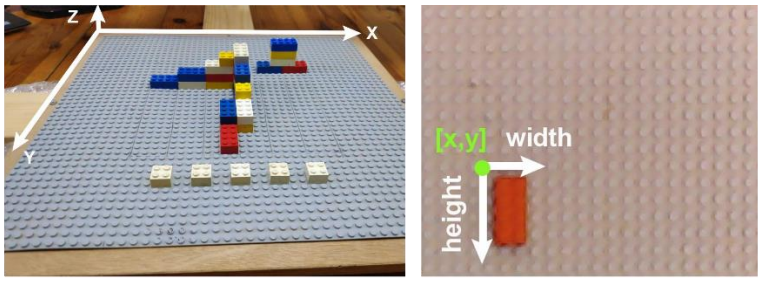
In this part, as briefly described above, our goal is to detect if some change is done in the physical lego model using an input of 8 images (4 for the previous state, and 4 for the current one). If a change is detected, we would like to predict the change type (addition or subtraction of a cube), so as the cube parameters - coordinates (x, y, z) and color of the changed cube.

For having that goal accomplished, we have an algorithm that splits into multiple paths according to results of the previous states in the analysis process. During the analysis process we maintain a 3D matrix of the current lego structure that built so far. This model is an input for many sub-units in this algorithm and can prevent detection errors many times.

For the state analysis explanation, we will set some definitions:

1. We work with 3D zero-based coordinates system (figure 10a).
2. The board is divided into 24x24x24 grid correlated to the lego physical grid. The z unit is a single lego cube height.
3. Each cube is [2x4x1] or [4x2x1] dimension (figure10b). The system predicts the [x, y, z, width, height, color] of a cube, where the x, y coordinates is the upper left corner's coordinates. The width is the length along the x-axis (in lego grid units) and the height is the length along the y-axis (in lego grid units).

The general flow of the algorithm is first predicting the x, y coordinates and the color of cube, and then, using this information, predict the z (height) of the cube.

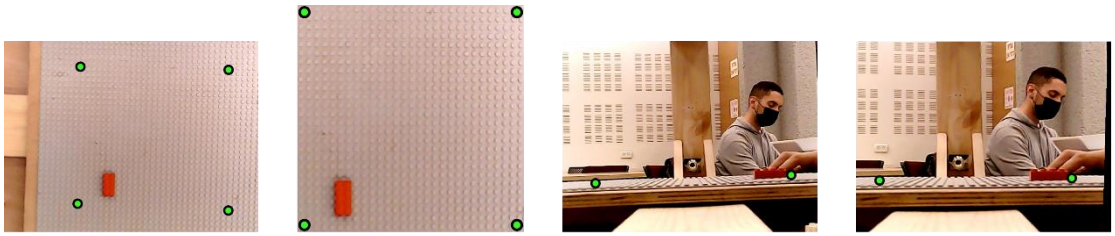


(a) (b)

Figure 10: (a) System coordinates system (b) A cube in [2x4x1] dimension.

Image straightens geometrical transformation:

Before any other operation, we preform the image straighten of the 8 images by geometric transformations that saved during the calibration (figure 11). As in the calibration step, the top image transformed using a 'projective' transformation and the side images are transformed using a 'similarity' transformation.



(a) (b) (c) (d)

Figure 11: (a) Top camera before transformation (b) Top camera after transformation (c) Side camera before transformation (d) Side camera after transformation.

XY (horizontal plane) and color detection:

The lego board divided into a lego grid - the lego grid is 24x24 units (corresponding to the physical lego circle connectors). Each 1x1 lego grid square is 20x20 pixels tile of the top image. The x, y that we aim to predict are the values of the lego grid, means [x, y] in range of [0,23]x[0,23].

For predicting x, y and color, we have few different methods. The algorithm starts with the simple ones, and if they fail it moves to the more complicated methods.

In general, we found out that it is better to analyze using the top camera rather than the side ones. The reason is that the side cameras also capture the background, while the top camera captures only the board.

XY and color estimation from top camera:

This unit is the basic method and the most reliable for detecting a cube, if it falls under some criterions - the cube has different color related to the cube below in all the cube's area.

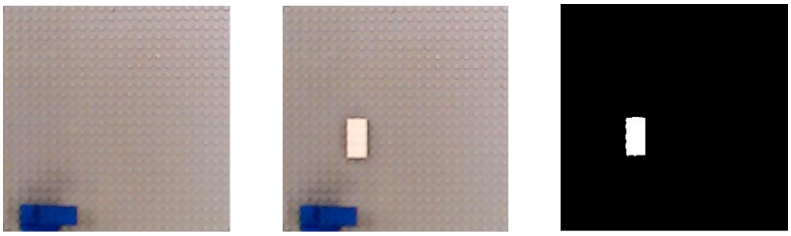
The algorithm is creating binary maps of the difference between the images (figure 12). Further description about the creation of the binary map is in the “Algorithms” section.

If the algorithm finds that the area of the binary map falls in a range that fits to a cube typical area, so that blob will be defined as a cube and the algorithm finds its coordinates.

For doing that, the algorithm uses a line detection algorithm (that also will be described in the “Algorithms” section) and rounding the results for fitting to the lego grid (cube cannot fall in a non-integer value on the grid).

For improving the accuracy of the method, we preformed two main actions:

1. We found out that some cases of shadow (created by other cubes) expended the blob in the binary map and caused shifting in the x, y prediction. For solving that, instead of making one binary map, we created 4 binary masks that each of them filters a color (red, blue, yellow, white). Using them, we created 4 binary maps filtered with those masks. That filtering helped removing the shadow from the binary maps, since the shadow is not in one of those colors.
2. We inferred the x, y from the lines that detected more clearly. For example, if the left edge of the cube detected as "straighter" compared to the right one, the x value of the cube based on the left edge. Deeper explanation about this method is given in the "Algorithms" section, including the meaning of "straighter" edge.



(a) (b) (c)

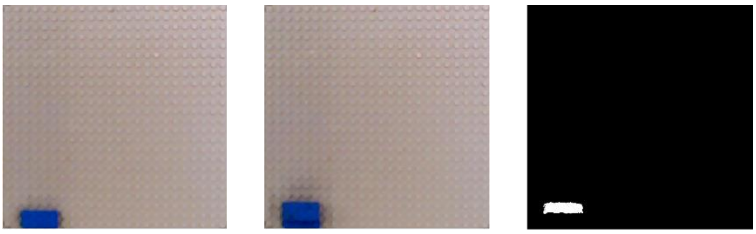
Figure 12: (a) Previous state image (b) Current state image (c) Binary map of the difference between the images.

XY estimation (partial cube) from top camera:

In case there was no detection in the previous component the algorithm gets to this component. This component checks the addition of a cube in a color on top of a cube in the same color, with partial or complete overlap (figure 13b). In case of partial overlap, the binary map contains a blob made by the non-overlapped area (figure 13c).

For analyzing this situation, we use the 3D matrix for checking whether we can figure out the coordinates of the new cube based on other known cubes.

For example, if we identify in the binary map a [1x4] cube (in lego grid units) in blue color, we suspect that this is a blue cube that added on top another blue cube in a shift (figure 13).



(a) (b) (c)

Figure 13: (a) Previous state image (b) Current state image – the new blue cube is overlapping the bottom blue cube (c) Binary map of the difference between the images. The active area is only the non-overlapped area of the new cube.

XY estimation from side cameras:

In case there was no detection in the previous component, this component is trying to detect a cube using the side cameras.

We created an algorithm for solving that issue. This algorithm is complicated and not straight-forward, therefore it will be described completely at the “Algorithms” section of this paper.

No change in frame:

In case of all the previous components did not detect a cube, the system returns that no change has been made in the lego board.

This can occur in two main cases:

1. The user actually did not make a change in the lego board. For example, put his hand inside the board area and took it out. In this case, the non-changed prediction is correct.
2. Some change did happen, and the system did not detect it. In this case, the system is wrong. For managing this issue, we built a solution for the user: It is possible to the user, using the game buttons, to remove the non-detected cube or to manually fix its coordinates to the correct ones, in case of detection but not in the right coordinates.

Validate color (for errors detection):

For detection errors, we added another mechanism that re-samples the color of the predicted x, y cube and compares it to the originally predicted color (from the x, y predicting process). Mismatch in the colors can point an error during the process. This component acts as a validation for our prediction correctness.

Z estimation (vertical plane):

After predicting x, y and color, the next step is predicting the z value (height) of the cube.

The vertical axes divided into a lego grid as well, which every unit is a single lego cube height. The z of a lego cube is in range [0,23], where cube in the first layer on the lego board has z=0 value.

We have two options for predicting z value - the first one is using the 3D matrix, and in case that it is not possible to predict using it - predicting using the side cameras. More than that, this part must predict whether its addition or subtraction of a cube from the model.

<u>Z estimation using 3D matrix:</u>

As said above, there are few cases that the z value of a cube can be calculated only using the 3D matrix. For understanding those cases, it is important to remember that the matrix holds the state before the current analyzed change.

For finding the z value using the 3D matrix, we first look for the cube with the highest z value in the coordinates defined by the [x, y, width, height] area.

The cases this component can predict are the following:

1. In case that in the detected [x, y, width, height] area of the cube there were no other cubes before, and the detected color is a color which related to a cube (red, yellow, blue, white) - this is a cube addition in z=0.
2. In the same concept, in case of that in the detected [x, y, width, height] area of the cube there was a cube in z=0, and the detected color is the board color - this is a cube subtraction in z=0.
3. In case of that in the detected [x, y, width, height] area of the cube there were more than a single color before the current change that means that a cube was added in z=highest+1 (where “highest” is the z value of the current cube with the highest z in this area).
4. In case the detected cube is in a different color from the two cubes below it in the same area, it cannot be a subtraction, so this is an addition. It cannot be a subtraction because in that case we would sample a color that we already sampled in this area. For example, if we had a tower made by a red on top of a blue cube, and now we sampled a yellow color, it cannot be a subtraction of the blue cube since we did not sample a red color.

Z estimation using side cameras:

In case that we cannot predict using the 3D matrix, we use the side cameras to predict the z height. This is also an algorithm that explained under the “Algorithms” section.

Define change type:

This unit returns the type of change that has been done - addition of a cube, subtraction of a cube, or an error.

In case that the two z-estimator components did not succeed predicting z value, we return a flag indicating at a prediction error. Otherwise, if we have all the predicted coordinates (x, y, z, width and height), we check the change type with the 3D matrix. We can determine that the change is addition, subtraction or an error. For example, adding a cube in place where other cube marked as already exists, or subtraction in place of empty coordinates are all errors.

Return cube parameters:

In case of successful prediction process, the “analyze state” component returns the change type and the cube parameters (x, y, z, width, height and color).

In addition, the 3D matrix is updated with the current change that has been done.

Assumptions and Limitations:

Assumptions:

1. All the cubes must be inside the pre-defined building volume. The volume is defined also by the xy grid, and a maximum building height in each area of the building board.
2. All the cubes have 4 by 2 size (measured in lego grid units) and have color which is one of the pre-defined colors: red, blue, yellow or white.
3. The orientation of the model building process is upward. That means, in the current configuration of the system, the user is not allowed to add or remove a cube below an exist cube on the same area (or even overlapping with area of exist cube).
4. The user is allowed to make changes in the lego board only when it is allowed by the system ("wait for movement" status) and is not allowed to enter a hand or other element into the building area when it is not the status mentioned above. This is because during other statuses the system captures frames of the building area.

Limitations:

1. The system allows addition or subtraction of no more than one cube at a time. Making a change involved more than one cube will not be detected correctly by the system.
2. The added or removed cube must be visible and not hidden for the back camera and at least one of the right or left cameras. This is the minimum requirement for detecting changes in the lego board. Usually, the algorithm uses more cameras for the prediction, but this is the minimum that can be allowed for the algorithm's operation.
3. The system should be located in a lighting conditions environment such that all cameras are not over-exposed. In extreme over-exposed images, it is impossible to distinguish between the cubes' colors. Thus, those cases cannot be treated using color calibration.

Limitations that we overcame:

1. The background around the project environment should be relatively static. The algorithm uses comparison between frames to detect a cube. Some changes between the frames that caused by the background might interpolated as a cube.

   We made many operations for minimizing those error probability (and for evidence, the system did not make many false predictions during the demo day).

   The main solutions for overcoming this issue are:

   A. Working as much as we could with the top camera and the 3D matrix, so we worked with the side cameras only where there was no other option. That because the side cameras are the only cameras that capture the background in their field of view.
   B. In case we had to work with the side cameras, we added many masks to the binary map for filtering the background noises from the binary map: a polygon-based mask (as explained in the calibration section under "camera mask"), a

color-based filter (we look for changes only in the cube color, as described in “Algorithms” section under “color space”), and a model-based filter (as described in “Algorithms” section under detect xy from side cameras).

2. Shadows effect - when analyzing based on binary maps of differences between frames, we found out that shadows can affect the binary map area of the detected change. Since our system must be accrue enough for predicting the coordinates of the cube, such a distort of the binary map caused errors in the predictions.

   For example, at the beginning we predicted the cube’s coordinates with a naive estimation based on center of mass of the binary map’s larger blob. The shadow, in that case, made many errors in the prediction.

   The main solutions for overcoming this issue are:

   A. We moved to a better prediction of the coordinates, based on line detections instead of center of mass based. Description about this algorithm in described in the “Algorithms” section under “lines detection”.
   B. As described in other sections in this paper, we intersected the binary map of changes with a mask based on the searched color. Such an intersection filtered the shadow almost completely.
   C. We preferred predicting the coordinates based on the edges that detected more significantly, under the assumption that they will be also more accrue.

3. Changing in the device environment - when we moved the device between locations, we found out that the cameras moved a little bit, and the light conditions changed. In the requested accuracy of the system, those movements and lights changing caused significant errors in the coordinates and the colors predictions. To overcome this issue, we built many calibration mechanisms as described in the “calibration” section.

4. Cubes subtraction - we started with the assumption that the user is only adding cubes. During the developing process we added a functionality that allows the user to subtract a cube instead of adding one. Still, the limitation is removing only one cube at a time.

   More than that - at the beginning we planned to have buttons to let the system know if the user is added or subtracted a cube from the board. But at the end we even succeed doing it automatically. For doing it, we had to check things not only on the current frame, but also on the previous one. For example, if the binary map that produced the largest area is the one that filtered with the board color, and not the cube colors, it means that this is a subtraction. In that case, we must build new binary map with the frames from the previous state because there we can see the cube before it was removed.

Algorithms:

Geometric transformations:

Geometric transformations enable us to transform an image based on a set of source coordinates and a set, in the same length, of target coordinates.

We used geometric transformations at this system every time we took a new frame from our cameras. In all of the cases, the transformation matrix built during the calibration process. The source coordinates were sampled using 'ginput' (described under "calibration") and the target coordinates were pre-defined coordinates that we wanted to transform our frame into them.

The difference between the geometric transformations preformed on the top camera and the side cameras are the transformation's type [1]:

1. The top camera transformed using 'projective' type, which require at least 4 source and target points. We used that in order to flatten the lego board area, which might include perspective manipulation (not a significant one, since the top camera is mounted looking down).
2. The side cameras only stretched and tilted in order to transform the bottom left and right corners of the board into two pre-defined coordinates. Such manipulations fit the 'similarity' transformation type, which enables scaling, rotating, translating and reflecting (not needed in our case) of the image. This transformation requires at least 2 source and target points.

Note: those two types of transformations can be calculated over more than 4 and 2 points respectively. In that case, the "estimateGeometricTransform" function will calculate the transformation matrix for best matching the points at the optimal way, and return the inlier and outlier points (points that fit and doesn't fit to the calculated transformation).

Binary map:

Binary map of differences between two frames (from the same camera) is a fundamental building block in many units in our algorithm (detecting a cube from top, from sides and more). Our goal is to get a clear binary map which represents the lego cube area that changed between the frames.

For doing that, we have a series of manipulations to the binary map in order to get a map that is noise free and contains a single object.

The algorithm is:

1. We make a map of the difference between the images by subtracting the images channel-wise, then summing the absolute value of the 3 color channels (R, G, B) into a single channel, and normalizing the result to [0,1] range.
2. Binarize the result with threshold of 0.2 using the 'imbinarize' built-in function.
3. If a mask needed (for example, while working with sides camera, we use masks that described under the relevant algorithm), we intersect the current binary map with the input mask.
4. Remove blobs that have area smaller than 200 pixels using 'bwareaopen' built-in function, since we consider them as noise.

5. If the map is not empty, means there is at least one blob of pixels in the binary map, we want to keep only the main blob (the blob with the largest area) on the map and remove the rest. We do that from the assumption that the element that we aim to find, a cube, should be a connected area, and the rest of the blobs are suspected as noise or background movement. We can assume such a thing thanks to the masks that applied in step number 3. For this operation we use the 'regionprops' built-in function for "pasting" the largest blob pixels on an empty map in its original coordinates.
6. Return the binary map.

Detect cube lines from top:

A fundamental need in the system is the ability to detect lines in a binary map. Using this, we calculate the coordinates of the cube appears in the binary map.

From the following reasons, we decided to use an algorithm that we built and not using Hough transform for line detection:

1. Hough transform can detect lines in many directions, while we were interested only in horizontal and vertical lines. We thought that using Hough for this task will be overkill.
2. Because of the shadow, some edges of the cubes appeared wavy in the binary map and did not detect by Hough transform (figure 14a).
3. Hough calculations have computational complexity and will slow down the system speed performance.

Instead of Hough, we built more efficient algorithm that fits our needs.

The assumption for our algorithm is the existence of a single blob of pixels in the [NxM] binary map. The blob has a general shape of a square, which is lined up respect to the image (means, its edges appear as horizontal and vertical edges).

The algorithm:

1. We calculate the center of mass coordinates [cx, cy] of the pixels in the binary map (figure 14a).
2. Using the [NxM] binary map, we calculate two 1d vectors: one sums the row values (N length vector) and another one sums the column values (M length vector) – figure 14b.
3. Calculate approximation for derivative for those vectors by calculating the difference between pairs of values (figure 14c).
4. Detecting edges coordinates by looking for the min\max values in the derivative row\column vectors, in some direction related to the center of mass coordinates (figure 14c):
    a. For detecting the left edge, we look for the max value of the derivative row vector in left to the center of mass (in x values [1, cx]).
    b. For detecting the right edge, we look for the min value of the derivative row vector in right to the center of mass (in x values [cx, M]).
    c. For detecting the top edge, we look for the max value of the derivative column vector above the center of mass (in y values [1, cy]).
    d. For detecting the bottom edge, we look for the min value of the derivative column vector below the center of mass (in y values [cy, N]).

5. The 4 coordination of the edges are the results of the previous step. Moreover, we can define what edge detected more significantly by comparing the absolute values of the min and max values from the previous step. Defining the significant edges has been used for defining the cube coordinates, as described in the detection from top for example.

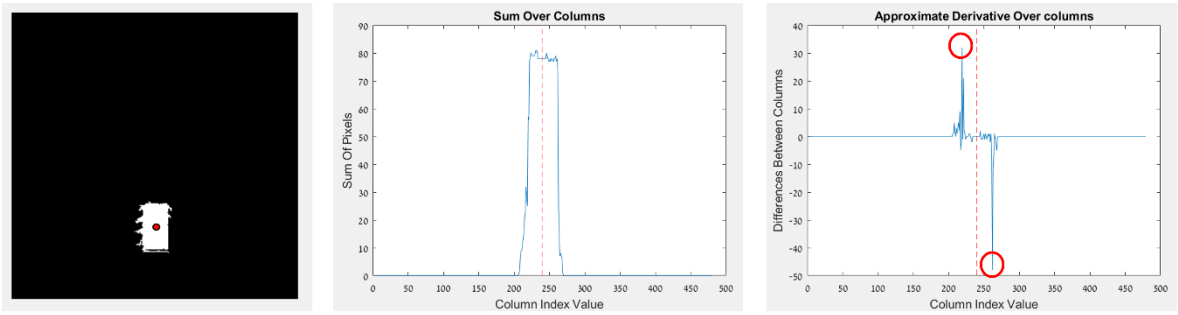


Figure 14: (a) The binary map with the calculated center of mass (b) The 1D sum over columns vector. The red vertical line is the center of mass (c) The Approximate derivative of the 1D vector. The selected min/max values in respect to the center of mass marked in red circles.

Color spaces:

In many stages of the algorithm, we want to filter the binary map by a color-based filter. Since we know exactly the color values of our cubes, due to the color calibration process that calculated the average color value of every color (red, blue, yellow, white and board color) from each of the 4 cameras. Using those color samples, we had to do two procedures in the system:

1. Produce a color-based mask for an image, a target color and the selected camera.
2. Decide what is the cube color based on sampling the image and decide what is closest color.

We searched for the most useful color space to preform those operations (RGB, HSV or Lab). We found that the HSV is not the most suitable choice due to fact that we work with white cube, which can get any H (Hue) value and thus cannot be filtered using this channel.

Finally, we found the RGB color space as the most useful. The main reason for that is the fact that the 3D points which represent the [R, G, B] values of our colors (red, blue, yellow, white and board color) in the 3D space (of the RGB color space) are the most spaced in space. This is an important criterion since the color estimation procedure requires looking for the closest color vector in the Euclidean 3D color space (figure 15).

The creation of the masks, as described above, are just selecting the pixels of the image that fall not far than a threshold from the sampled values in each channel of the R, G, B channels.

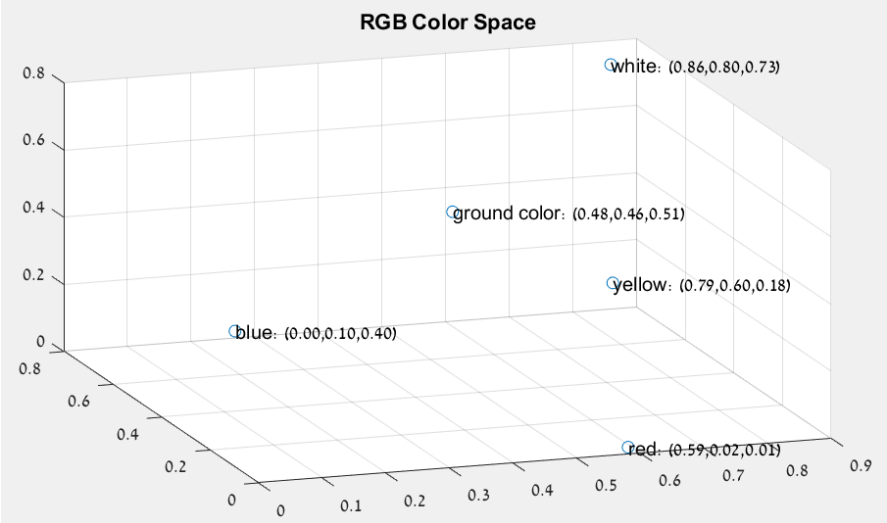


Figure 15: RGB color space with the cube colors as data points on the 3D space.

XY prediction using side cameras:

This algorithm uses the 3 side cameras (left, right and back) for predicting the x, y of a cube.

The challenge with this kind of prediction is the need to match (triangulate) in high precision the predictions from the side cameras, in order to get a good accuracy.

The algorithm divided into three main parts:

1. Generating binary maps for the side cameras
2. Generate "coordination sweep" for each camera
3. Match the cameras for having the best prediction

We will explain each of those parts.

For each potential color (red, blue, yellow, white) we do the following steps:

1. Generating binary maps for the side camera:

The binary masks of the differences between two frames are being produced using the algorithm "Binary maps" that described in this section. The mask for creating this binary map is constructed by the intersection of 3 different maps (figure 16e). Having those effective masks are important key for the ability to get reliable predictions from side cameras.

The three masks are:

a) A color based binary mask (as described in "color space" section) – figure 16b.
b) A polygon based binary mask (as described in "calibration" section) – figure 16d.
c) A model based binary map: that map is limiting the z-height based on the current build cubes on the model. For example, in empty areas of the model, the cameras should not look for new cubes above the first layer z=0 (since no cube can be added suddenly in z=1). In contrast, in area where the model has few cubes, also the area for changes searching should by higher (figure 16c). This mask is being created per camera (and its own perspective parameters) and current model that saved in the 3D matrix.

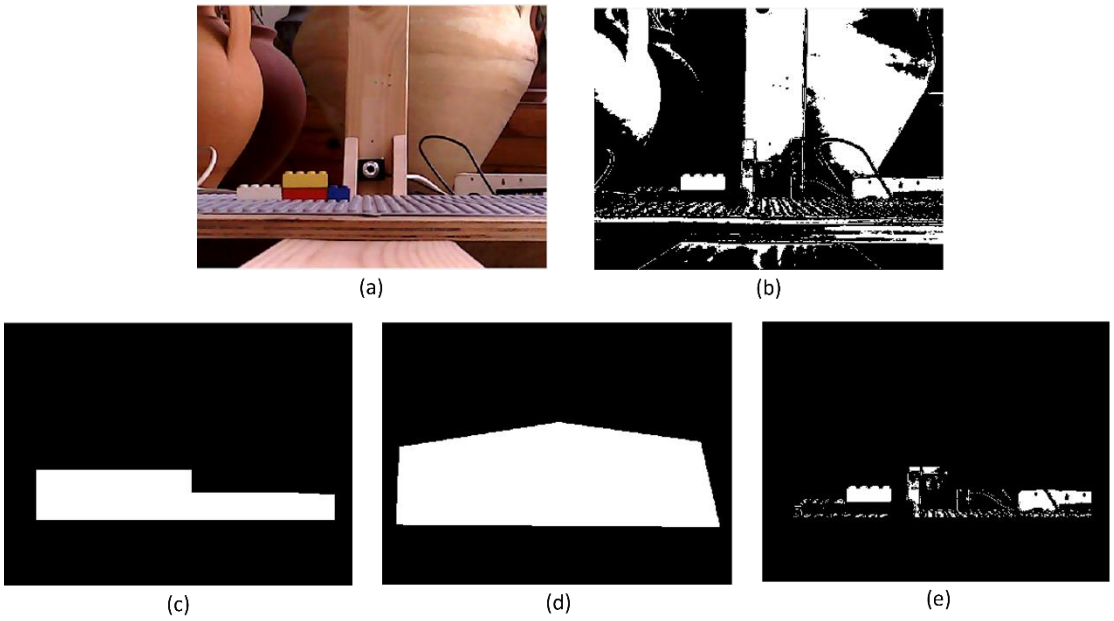


Figure 16: (a) The original image from right camera (b) Yellow color binary mask (c) Model-Based binary mask (d) Polygon-Based binary mask (e) Final binary mask.

2. Generate “coordination sweep” for each camera:

Using the binary map, we need to estimate x, y, width and height parameters of a cube. The main challenge is that when analyzing the binary map from a single camera perspective, the camera has a degree of freedom (or too many unknown parameters).

For example, we have a binary map from the left camera with a blob of pixels. Only using that, we cannot estimate the parameters because this blob can fit into many combinations of [x, y, width, height] values.

We solve that by iterating over one of the values, assuming it is a “search parameter”, and calculating the rest of the parameters related to the value of the search parameter.

We defined the search parameter for each camera as the coordinate that related to the distance of the cube from the camera (the depth of the cube from the camera). Thus, for the left and right camera the search parameter is the x value, and for the back camera it is the y value.

The result of this part is a table of the search parameters running in range [0,23], and the corresponding other x or y parameter (y for left and right, x for back camera) related to the search parameter value, and also the predicted width or height of the cube related to the same search parameter value.

3. Match the cameras for having the best prediction:

Using those "coordination sweep", we aim to find the most reasonable match that leads to the predicted [x, y, width, height] values.

First, as a limitation, we must get a valid coordination sweep from the back camera and at least one of the left or right cameras. Otherwise, the algorithm fails to predict. That requirement comes from the need to triangulate cameras.

For the matching process, we will choose the back camera and only one from the left or right cameras. If both of them succeeded generating a valid coordination sweep, we will choose the one with the largest blob's area in the binary map.

The algorithm for the matching process between the 2 coordination sweeps is based on a price function. The algorithm aims to find the [x, y] couple with the minimal price.

The matching process require iterating over x in range [0,23]. For each x, get the y value in the left/right camera's coordinates sweep. This y is not a rounded number. We will check for $y_1 = \lfloor y \rfloor$ and $y_2 = \lceil y \rceil$, and the prices are $C_{y_i} = |y - y_i|$ for the two y_i values $(i = 1,2)$. In the same manner, we will check what are x values that related to the two y_i values $(i = 1,2)$. For every y_i $(i = 1,2)$ we will get an un-rounded x_i $(i = 1,2)$ value from the back camera's coordinates sweep, and rounding them into two rounded values $x_{i1} = \lfloor x_i \rfloor$ and $x_{i2} = \lceil x_i \rceil$. The price is the rounding error $C_{x_{ij}} = |x_i - x_{ij}|$.

Finally, for every x_{ij} $(i, j = 1, 2)$ that equal to x, we check the sum of the prices:

$$Price = C_{y_i} + C_{x_{ij}}.$$

The x, y couple that have the minimum price are chosen as the selected x, y parameters.

The width and height can also be calculated from the coordination sweeps and the selected x, y values.

At the end of those steps, under the described conditions, the algorithm predicts the x, y, width and height parameters of the cube, or indicating that no change has been detected in the scene.

Z prediction using side cameras:

This algorithm in used when the z value (height) of the cube cannot be predicted using the 3D matrix. When reaching the z prediction stage, the [x, y, width, height] parameters of the cube are already predicted. In that case, the algorithm contains the following steps:

1. For every camera with line of sight to the new cube, calculate the z prediction from the camera
2. Weighted-sum the predictions into a final z prediction.

We will explain about those two steps.

1. For every camera with line of sight to the new cube, calculate the z prediction from the camera:

For each side camera, we first check if a line of sight from the camera to the new cube exists. This line of sight is checked using the 3D matrix - if the volume between the camera and the new cube is marked as not-occupied, we will call it free line of sight.

In that case, we predict the z value by those stages - first, we generate a binary map of the differences between frames from this camera. The algorithm for the binary map creation is described in this section. The mask for creating this binary map is constructed by the intersection of three different maps. Having those effective masks are important key for the ability to get reliable predictions from side cameras. The three masks are:

a) A color based binary mask (as described in "color space" section) – figure 17c.
b) A polygon based binary mask (as described in "calibration" section).

c) A model based binary map: we generate a mask that looks only on narrow range around the [x, y, width, height] values of the cube. In contrast to the xy prediction from the side cameras, where we did not know the coordinates of the cube and had to look at the whole board, here we can focus only on the cube area and avoid affections from background in a better way (figure 17d).

As a second stage, we can look at the blob in the binary map and predict its z (figure 17e). We do that by calculating the y coordinates (in image pixel space) of all the potential z values. Then, we choose the z related to the closest actual y value from the binary map.

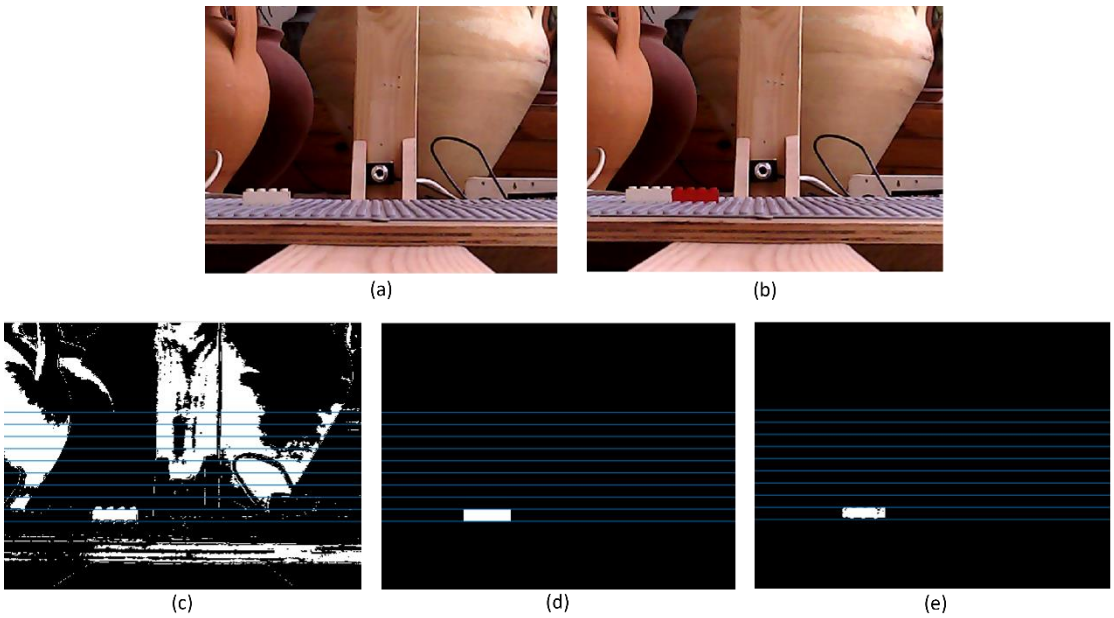


Figure 17: (a) The previous state image (b) The current state frame, red cube added (c) Red color binary mask (d) Model-Based binary mask (e) Final binary map. Each horizontal line represents a potential z height. The closest line to the bottom of the cube is selected.

2. Weighted-sum the predictions into a final z prediction:

For each camera that could generate a z prediction, we store the prediction with a score for the prediction quality. The score is based on the distance of the cube from the camera.

We assumed that z prediction on cubes that are closer to the camera will be more precise compared to a prediction from a faraway cube. Thus, we score the prediction quality of the range of 0.5 (for the most far away cube) to 1 (for the closest cube).

The final predicted z is the result from the weighted sum.

Computational Analysis

In the computational analysis section, we deeply analyzed one component of the system - predicting x, y of a new cube.

The motivation for this analysis is to evaluate the redundancy in our system. In our system, we have four cameras - one on top and three on sides (left, right and back). In the normal operation, the top camera is used for [x, y] prediction and the side cameras are used for z prediction. With that, the side camera can also predict the [x, y] of a cube as well. We wanted to check, using this analysis, if the [x, y] can be predicted in the same accuracy as it has been done by the top camera. If that is the case, we could remove the top camera for our model and simplify our system (smaller device, less hardware and better speedup due to access to less cameras). If this is not the case, we wanted to figure out what are the main settings that cause the accuracy drop.

As a preliminary step, we created a dataset. The dataset contains frames of lego building steps, taken using the 4 cameras mounted on the device (around 400 frames - 100 from each camera). The building steps represent the full distribution of allowed building steps in our system. They spread around the full allowed building areas, in all the allowed colors, different heights and orientations.

For evaluating the component, we manually recorded the ground truth [x, y] of those building steps, using for comparing the predictions made by the component.

We decided to measure the performance by measuring the 2D Euclidian distance between the ground truth [x, y] to the predicted [x, y]. An accrue prediction yields distance zero, and the prediction marked as worse as long as this distance rises.

For better presenting and analyzing the results, we divided the measurements into three classes:

1. Predicted correctly (error distance = 0 in lego units) - where the system predicted the [x, y] correctly.
2. Small errors (0 < error distance <= 2 in lego units) - errors of distance under 2 measured in lego grid units, as explained in the definitions of this paper. Those errors are usually related to a correct detection of the cube with some inaccuracy in the measurement process: the creation of the binary map, the line detection, the lego grid quantitation, image perspective (figure 19), etc.
3. Large errors (error distance > 2 in lego units) - those errors are significant and can be related to wrong detection of the cube. Those errors usually occur because some other elements in space have stronger characteristic of a cube than the real one. Thus, the system measures their coordinates instead of the real cubes. We made many error detection mechanisms in the system to prevent those cases or at least detect that the prediction is incorrect, but the system is not error-proof and those errors happens under some conditions.

At the first step of the analysis, we analyzed the [x, y] detection from the top camera. The algorithm in this case is quite straight-forward - create a binary map from the top camera, detect lines and round them to the lego grid units.

We compared the [x, y] predictions from the top camera to the ground-truth [x, y] values of the cubes. The results presented in figure 18.

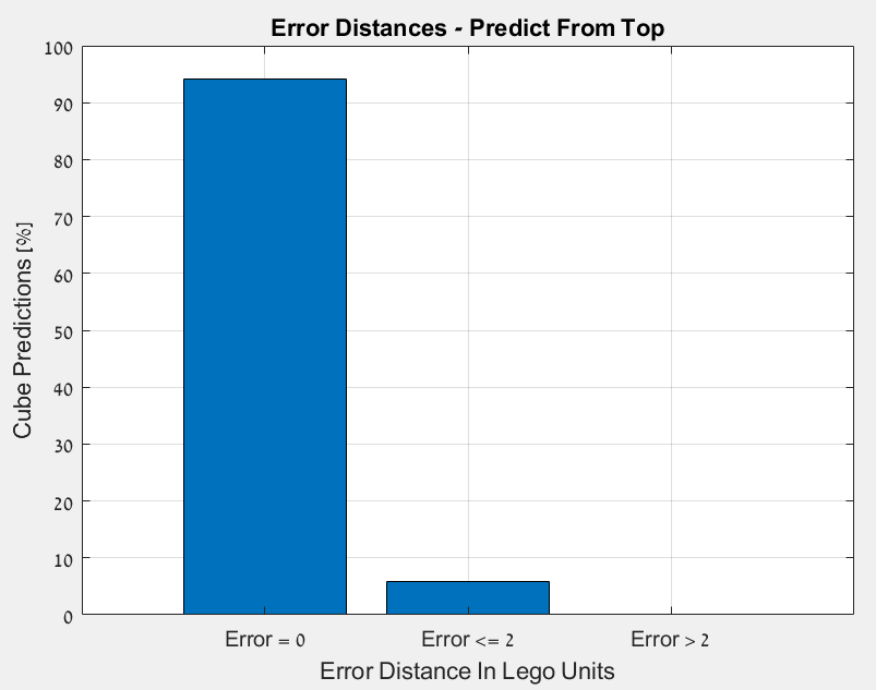


Figure 18: The distance error distribution from top camera prediction.

As we can see in figure 18, the [x, y] prediction from top is very accrue component. All the errors of those components have magnitude of not larger than 2, and the probability for error is low.

We can see an example for a small error in the top prediction (figure 19). In that case, we added a white lego cube in z=5 (relatively high). The top camera prediction was y=1, but the cube is actually in y=2. The reason for the error is perspective of the top camera (due to the facts that the cube is close to the corner, and in a high z value). Those cases did not get a special correction in our system because the rare probability for them, since the users are not asked to build such high structures in the corners, but this can example when do the top camera mis predict.

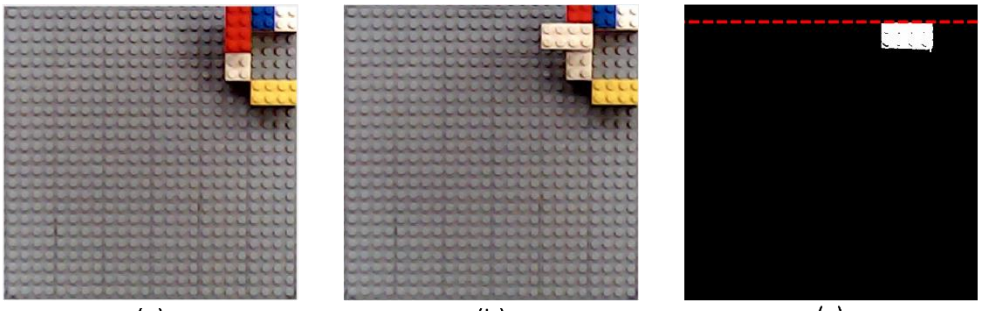


Figure 19: (a) The previous state image (b) The current state image (c) Top binary map. The red line represents the predicted y value.

As a second step of the analysis, we analyzed the [x, y] prediction from the side cameras. This algorithm is described under “Algorithms” section. This is a complicated component, contains creating well-masked binary maps (because those cameras are sensitive to background elements) and a matching algorithm.

During the normal operation of the system, in most cases the [x, y] of the cube is predicted using the top camera, because of the presented above. In this analysis, we checked the performance of the prediction from side component also in cases that it is not actually needed by the entire system.

We can see in figure 20 that the prediction from the side is less accurate, compared to the prediction from top.

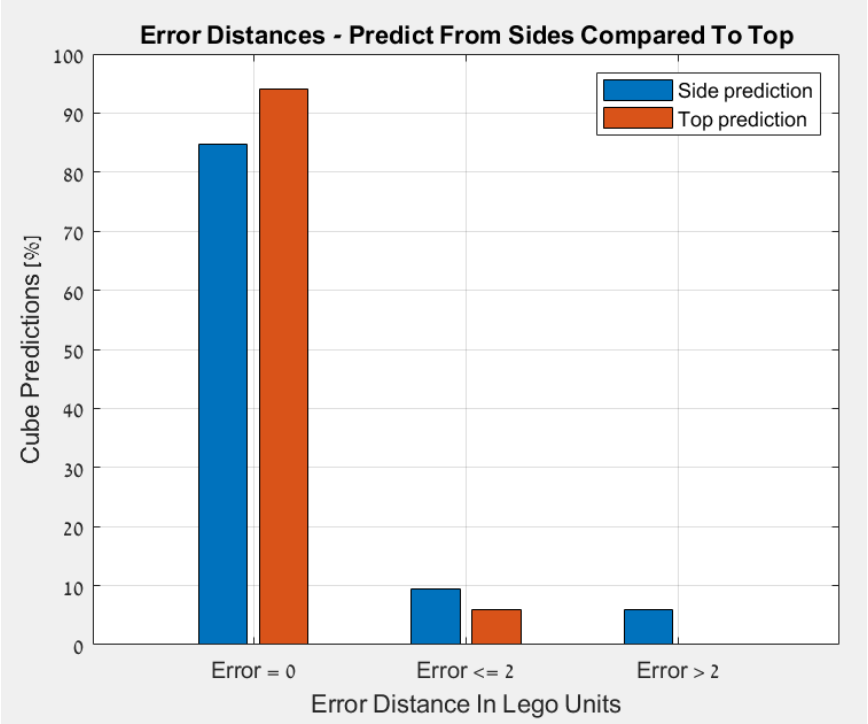


Figure 20: The distance error distribution from side and top cameras prediction.

As we can see, the accuracies we obtain from this component are lower. We see more small prediction errors, and large errors that we did not see in the top case.

We aimed to figure out, using the analysis, if we can point at some setting or parameters that caused the prediction errors. The first parameter that we checked is a parameter related to the color mask in the binary map.

When masking the binary maps from the side cameras, one mask is for defining the cube color. We made this mask by selecting all the pixels that fall no longer than T threshold from the sampled R, G, B values of the cube. We checked the accuracies of the system using three different setting of this parameter:

1. T = 0.1 - Small margin. The mask keeps only pixels that are very close to the calibrated color. That setting make the system not sensitive to light condition

changing, since color which is not very similar to the calibrated color will be rejected from the mask.

2. T = 0.3 - The value we used in normal operation.
3. T = 0.5 - Large margin. With this value, the mask almost not filtering colors at all.

The results presented in figure 21.

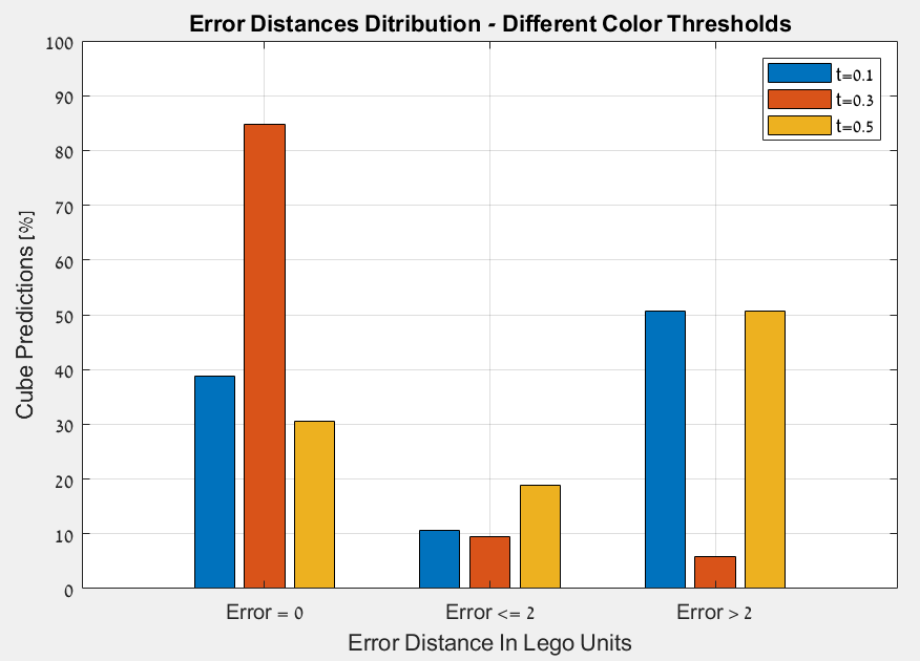


Figure 21: The distance error distribution for three different colors masks parameters.

We can obtain from the plot that the T=0.3 parameter is the optimal between those values. The accuracy drops in T=0.1 is reasonable since the binary mask is very selective (does not select the complete cube area and causes cubes missing in the final binary map). The interesting insight is about the T=0.5 case. Letting too large margin can help with various light conditions but can lead to large errors. One of the reasons is the potential to detect other background elements as the cube, also if their color is not exactly as a cube color (due to the lack of color filtering).

The next element to be under test is the model-based mask. This mask is described under the algorithm's explanation in “Algorithms” section. In a briefly explanation - it blocks areas that are not in potential build coordinates (for example, looking on z=3 where z=0 layer is empty in that area).

We compare the accuracies with and without this mask (with the optimal color threshold parameter, T=0.3) and the results shown in figure 22.

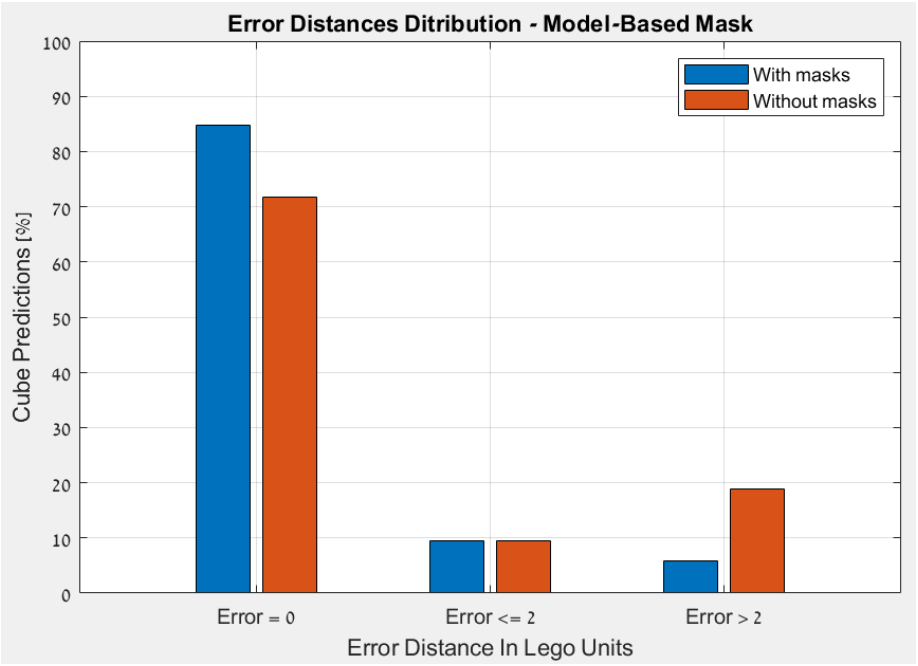


Figure 22: Comparison between model-based mask settings.

From the plot we can see that without using this mask, the small error rate did not change much while the large error rate increased. The reason for that is the larger potential of the system to detect a noise element in the background and treat it as a cube. That case causes a large error. If so, this mask is helpful and necessary.

So far, we can see that the parameters that we originally used (with model-based masks and color mask threshold of 0.3) achieve the best results among the tested parameters.

We would like to note that the parameters we chose tested empirically on a wide range of sets (we didn't present all the tested parameters for clear presentation). Because the parameters chosen empirically, we cannot ensure that there are no other parameters that can lead better results. With that, the accuracies using the current parameters yields good enough accuracies for good a user experience in real-time usage as could be seen in the demo day.

If so, at this stage of the analysis we would like to investigate the prediction errors by checking few aspects of them in order to figure out the reasons for those errors.

The first aspect is the direction of the prediction error. For each prediction, we checked the x and y 1D Euclidian distance separately (the distance between the predicted x or y to the ground-truth x or y, respectively). The results presented in figure 23.

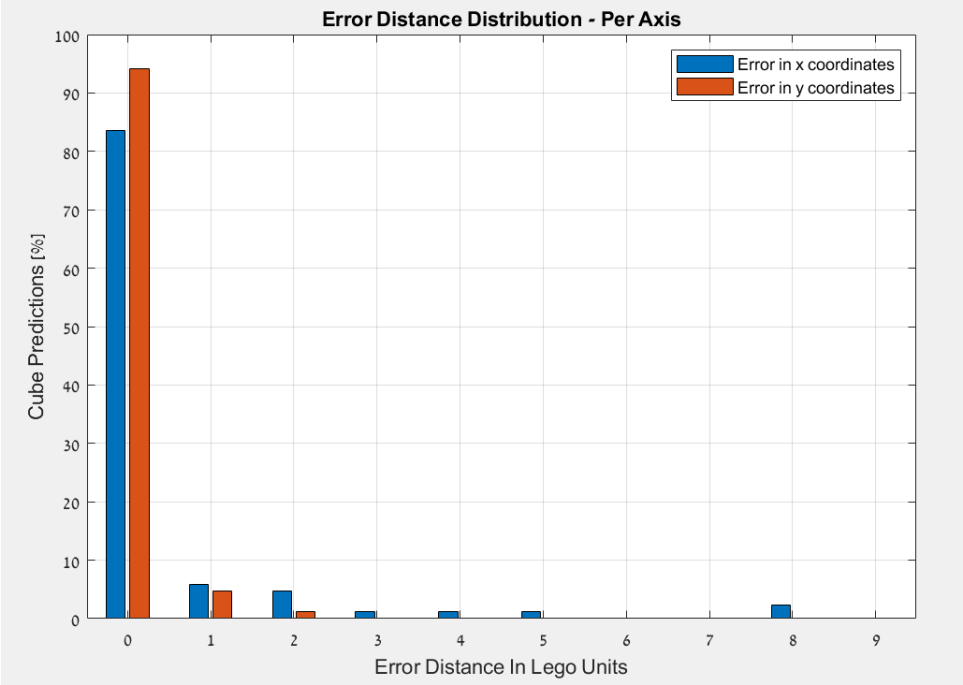
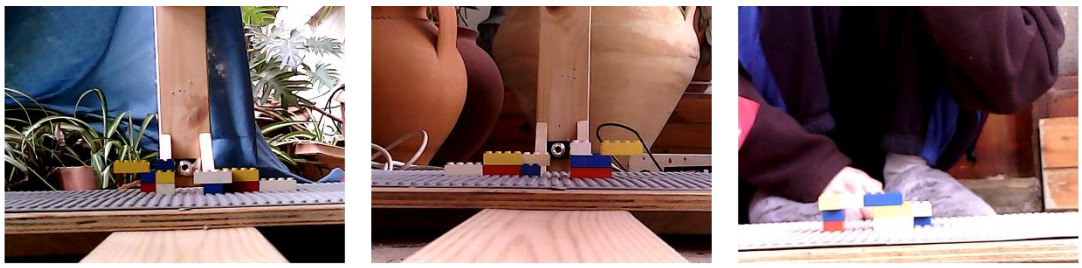


Figure 23: Error distance distribution in each x, y axes.

We can see from figure 23 that the x-axis tends to have more errors. By deep looking into our dataset, we found out two potential explanations for this result. Both explanations lie on the fact that in the [x, y] prediction from side cameras algorithm, the left and right cameras are more dominant with setting the y prediction, while the back camera is the main responsible to the x value prediction.

The reasons are:

1. We have two cameras that predict the y-value (left and right), while only one is predicting the x-value. In case that both left and right could detect, the more significant result is chosen. In contract, the back camera (and the related x prediction) has no redundancy.
2. In the current dataset, the back camera has worse light conditions in some of the images. That caused less quality images (little bit over-exposed), which yields less accrue binary map and larger x value inaccuracy. The difference between the side cameras image quality is demonstrated in figure 24.



(a) (b) (c)

Figure 24: (a) Left camera balanced-expose frame (b) Right camera balanced-expose frame (c) Back camera over-expose frame.

We will example a significant error from the back camera due to detection of a wrong object (hand instead of a cube). Main reason for that is the over-exposed frame, caused the yellow and white to have similar colors.

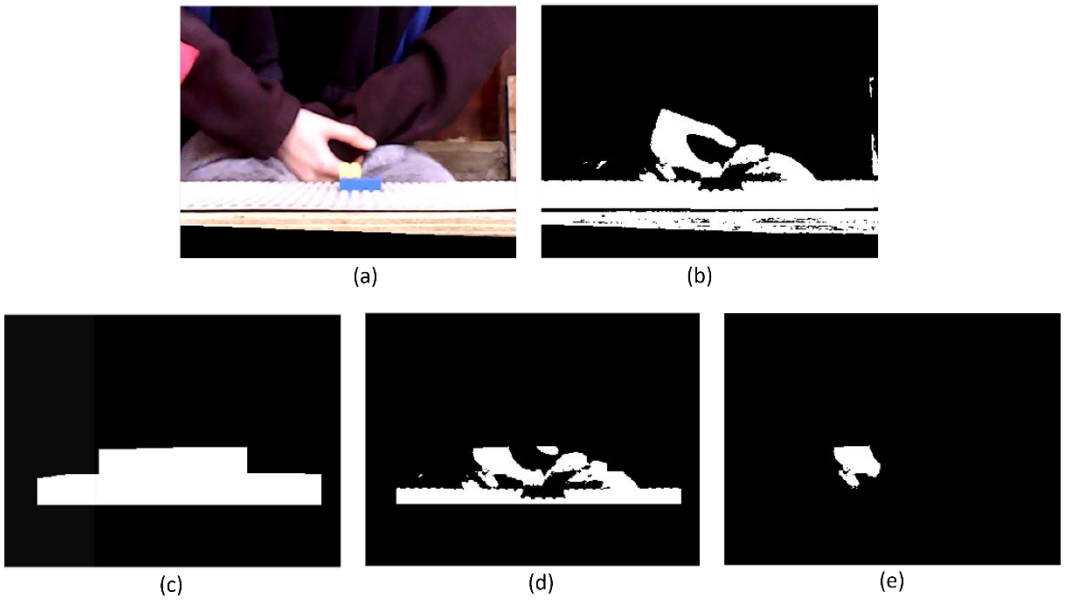


Figure 25: (a) The analyzed frame. The yellow cube was added (b) Yellow color binary mask (c) Model-Based binary mask (d) Final binary mask (e) Final binary map (changes is frame intersected with the binary masks). We can see that the hand detected instead of the cubes.

Another way to evaluate the x, y prediction errors, combined with the difference between the top to the side cameras prediction, is using the scatter plots for x-axis (figure 26) and for y-axis (figure 27).

In those charts, we present the coordinate prediction using top camera on the x axis, and the respective prediction from the side cameras on the y axis. Points that do not fall on the main diagonal indicate prediction errors between those units.

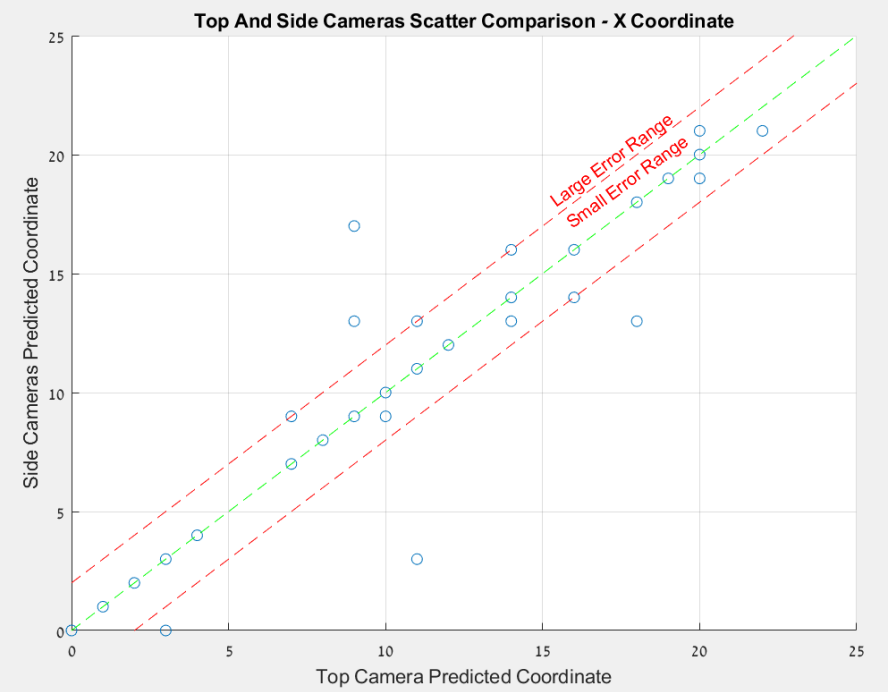


Figure 26: Scatter plot for x-axis prediction by top and side cameras.

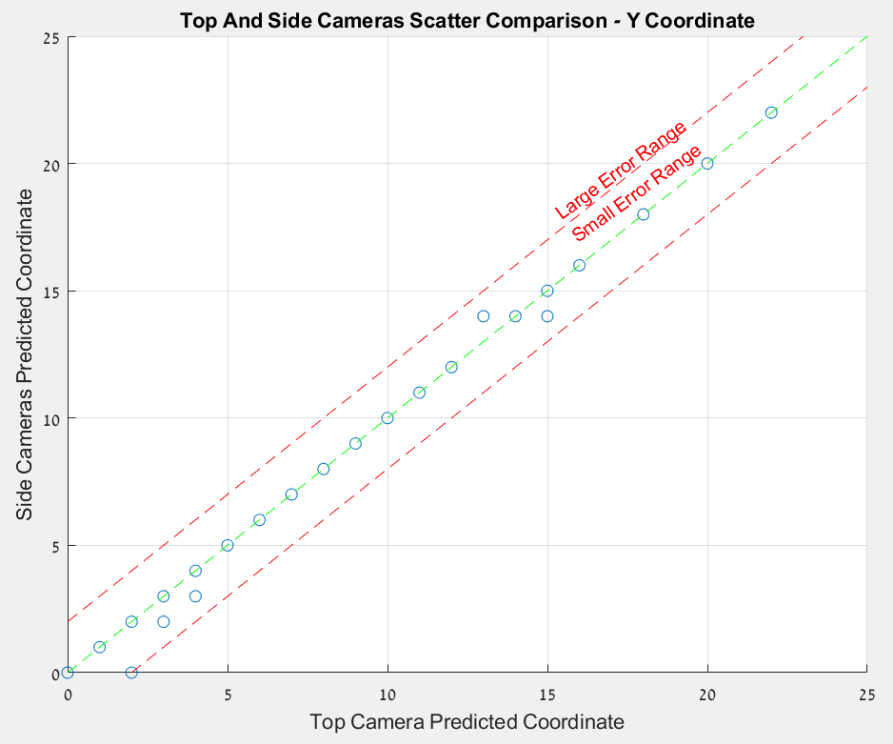


Figure 27: Scatter plot for y-axis prediction by top and side cameras.

On the y coordinate plot, we can see that all points fell down in the small error margin, compared to the x plot where some points fell further.

We have to note that this plot compares the side prediction to the top prediction and not to the ground truth, so the error can be smaller or larger than the ground truth error, depend on the combined two components prediction's precision.

For decomposing the system even more, we would like to explore the ability of the component to detect cases where it cannot detect the cube. In some cases, due to hidings, one of the left or right cameras cannot detect the cube. In that case, we expect the camera to mark with a flag that it did not detect the cube.

For evaluating this phenomenon, we manually marked for each frame of the left and right cameras in the dataset if they have a line of sight (clear view from the camera) to the changed cube. We compare it with the predicted line of sight flag returned by the cameras. We name the real existence of the line of sight as positive and negative, and the returned flag for the cameras indicate the true\false of the prediction.

The confusion matrix we calculated presented in table 1 [2].

		True Line Of Sight	
		Positive	Negative
Predicted Line Of Sight	Positive	TP 90.00%	FP 2.35%
	Negative	FN 3.53%	TN 4.12%

Table 1: Confusion matrix for side cameras line of sight analysis.

As we can see, the system generally acts as expected, with 94.1% accuracy.

We can also see that 2.3% of the cases are false positives. False positives (marking a cube detection while it is not visible) usually point at detecting a wrong object or detecting only part of the cube (figure 25) that can lead to final wrong [x, y] prediction.

We also can see 3.5% of false negatives, which are cases that line of sight is actually exist but not detected. This value is relatively low, means that the system tends to detect the line of sight if it is actually existing. We analyze that the errors in the system are more sensitive to the false positives rather than false negatives.

The side cameras' line of sight precision is 97.5%. The left 2.5% can be the reason for the large error occurrence. This is a low value, but those cases are critical to the system because they can fool the final prediction of the system.

The 96.2% recall is not a major issue, since if a camera is not detecting the line of sight also if one is existing, we can usually backup using the other side camera without harm the system performance.

Results and Discussions:

We divide the results to the overall system results and the analysis results.

We analyze the overall system as a general block which gets as an input of a physical 3D lego structure and outputs a virtual 3D model to the game software. If so, we measure how the output, which is the virtual model, has the desired accuracy as we expected.

In average, the overall system predicted the cubes correctly in 93% of the total cases. We calculated this value using the dataset that we used for the computational analysis. Those cases included simple scenarios, but also complicated ones - for example, a cube that is been located above cube in the same color, which force the system to detect and predict the coordinates from the side cameras. From the user experience aspect, during the demo day we recorded only rare cases where the system mis predicted a cube coordination or color. Also in those cases, thanks to the cube fixing functionality, the system maintained a continuous game flow that yields positive feedbacks from the users.

Figures from the system algorithms and computational process are being presented along the paper, thus we did not add more figures in this chapter as well.

The computational analysis checked the [x, y] coordinates prediction, and mainly focused the prediction of those coordinates from the side cameras.

Along the component analysis, we obtained that the top component has better accuracies and have no large errors, from the kind that we found in the side prediction component from time to time.

Although the side cameras' prediction had not the best accuracies, it reached very well results (94% accuracy).

We also noticed a difference in the prediction accuracies between the x coordinate and the y coordinate. We found out the main reasons for that are the lack of redundancy in this camera direction, and bad image quality in the back camera. It is hard to evaluate what is the contribution of those two issues on the final results.

In this section we also explored the influence of the color calibration, color threshold and binary mask that we used in the project. The setting that we used during the system operation achieved the best accuracy values among the tested parameters.

We believe that addition of more algorithms for error detection and prevention in the [x, y] from side prediction component can even improve the current accuracy, as measured in the analysis.

At the beginning of the computational analysis chapter, we aimed to check whether we can remove the top camera and predicting the [x, y] from the side cameras as well. Regarding that issue, with some improvements based on this paper analysis, we think that this change is possible.

The tradeoff between keeping or removing the top camera is having more compact device, with less hardware and better speedup due to access to less cameras on one hand. On the other hand, removing the top camera means losing accuracy and skipping redundancy in the system.

Conclusions:

In this project, we managed to use 4 simple web cameras in order to detect in good accuracy the coordinates and the color of lego cubes, while succeeding overcome major limitations such as predicting using triangulation from multiple cameras, dynamic background, environment conditions and other difficulties that engineers find out while converting an idea into a final project.

More than that, we even added functionalities that helped improving the system robustness such as the ability to fix the coordinates of mis predicted cubes.

The final project is a complete and joyful game, which can be operated completely by the user.

As a future work, we can add more features or functionalities for improving the system. Due to time and team size limitations, we did not do it during this project scope, and we focused on the main game flow while developing the system.

We could not reach such an impressive and complicated project without the image processing tools that we learned during the course and the support from the course stuff, and we would like to thank for that.

Bibliography:

[1] Hartley, Richard, and Andrew Zisserman. Multiple view geometry in computer vision. Cambridge university press, 2003.

[2] Alvarez, Sergio A. "An exact analytical relation among recall, precision, and classification accuracy in information retrieval." Boston College, Boston, Technical Report BCCS-02-01 (2002): 1-22.