GMD Report 5

GMD –
Forschungszentrum
Informationstechnik
GmbH

Frank Śmieja

Fast 3D Cube Vision for Real- World Systems

March 1998

GMD –
Forschungszentrum Informationstechnik GmbH
Schloß Birlinghoven
D-53754 Sankt Augustin
Germany
Telefon +49 -2241 -14 -0
Telefax +49 -2241 -14 -2618
http://www.gmd.de

Anschrift des Verfassers/Address of the author:

Dr. Frank Śmieja

Institut für Systementwurfstechnik
GMD – Forschungszentrum Informationstechnik GmbH
D-53754 Sankt Augustin

This work has been carried out while the author
was a member of the
RWCP Theoretical Foundation GMD Laboratory

ISSN 1435-2702

Abstract

The step from robot simulation to a real-world machine is a big one, and arguably the most daunting aspect for anyone performing this step is the sudden need to introduce real perception into their system. The most typical and enticing way to enrich sensory information is to open the robot's eyes to the visual spectrum. The visual world is however one of arbitrary complexity and to hope for a general method for performing any particular aspect of visual recognition would be unrealistic. Therefore one must introduce constraints in the form of assumptions about the scene interesting to the robot and for the task to be performed. This requirement becomes most clear when speed is of the essence, as it typically is for robotic applications. This paper describes a method that makes a number of assumptions about the scene in order to generate a fast cuboid model (around five cubes per second) of cuboids interesting for our robot, JANUS. The 3D information is obtained through the use of two cameras mounted on a common movable head. This paper concerns itself also with the embedding and extension of such an algorithm in a reflective team architecture.

Keywords: Edge detection, fast vision, 3D-synthesis, reflection, teams.

Contents

1 Introduction

Some way of perceiving the world is necessary for real robots operating in a 3D world, especially if they are to possess a reasonable degree of autonomy. Unfortunately, one of the most information-rich ways, vision, is a very compute-intensive and arbitrarily involved procedure, which tends to make it typically and potentially very slow. For a robot that is to work and react in the real world containing moving objects, a vision method must operate at least at a rate of a few frames a second. Such methods are known as "fast vision" and generally owe their speed to various assumptions that are made about the environment and its contents, and to a rough but adequate processing of frame information.

The method described in this paper has no qualms about identifying its own limitations and the assumptions necessary for its smooth operation. Indeed we are convinced of the usefulness and reasonableness of developing algorithms that are capable of this kind of self-assessment. In this way it is possible to reuse them for other tasks and as part of teams [1].

The robot in our laboratory possesses two manipulators and a head with two cameras attached. It needs to manipulate blocks (cuboids) on and above a workbench. In the simulation phase of system development the information about the outside world (outside of the robot control system) took the form of simple data modelling of cuboids. This description was enough for us to be able to develop the main area of interest for us: the motoric skills of the robot. It eventually became necessary to extend our simulated investigations and methods to a real robotic system, in order to bring interesting dynamic effects in to the system. Since we did not wish to spend too much time and effort in developing complex visual recognition methods in order to extract the information we require from the external world, we decided to limit our excursion in the visual world to reproducing the information that the robot would normally receive from the simulation, and allowed any scene constraints that were necessary.

The paper is organized in the following way. In the next section we give an overview of the algorithm, from camera pixel images to the final 3D hypotheses. Sections $3-5$ describe the various steps in detail, and in section 7 we show some performance tests using the method. Section 8 explains how the algorithm parameters are to be estimated. Section 9 introduces reflective teams and shows how the algorithm may be represented as a team, and also be improved upon by reflection. In section 10 we provide our conclusions.

2 Algorithm overview

2.1 Assumptions

The following assumptions are made about the observed world of the robot:

- **Color.** The objects are all monochrome and have color belonging to a set of predefined cube colors. The background and all uninteresting objects are some shade of grey (from white to black).

- **Form.** The objects are all assumed to be cuboids of one of a few predefined sizes.
- **Calibrated cameras.** We assume the two cameras used have already been calibrated. This is an assumption for this paper only, since it is not our goal here to describe the calibration process, which can be found in another publication of our group [3]. The calibration of the cameras is necessary for generating 3D points from 2D image plane points from each camera (see section 5.1).

2.2 Advantages gained from the assumptions

- **Shadows.** Given the colorful nature of the objects, detection methods can be developed that take advantage of this and thus shadows, which tend to be mainly grey, no longer pose great problems (although see later for cases where they do disturb).
- **Reflections.** Similarly, since reflections onto grey objects also tend to have a high grey content (although not completely), they also pose little problem.
- **Lighting.** As a result of the last two factors, the intensity and configuration of the lighting is no longer a critical factor. It must of course be bright enough, but we found that we could get good results just by using normal room lighting. This is an important gain, especially when the robot is expected to work in a normal working environment.
- **Background noise.** Since detection is based heavily on the colorful nature and cuboid form of the objects in question, other types of visual cues can be easily ignored. Thus the background color and content is not of a critical nature. For example, objects that are too small or too large will be ignored, even if they have the right color; conversely, objects of the right size but non-cuboid color will also be ignored (this helps us to filter out the arms themselves, which have a yellow/orange hue).
- **2D-shape.** Since objects may only be cuboids, they will all have a typical 2D projection, which also helps to identify corresponding points in the two cameras, and to make hypotheses about missing[1] corners.

2.3 The detection process

Figure 1 shows the overall structure of the detection process. The input to the system consists of two images of the current scene from the calibrated head cameras. Each image is processed to produce two sets of 2D points representing the corners of the cubes identified in the images. This process consists of 4 stages:

1. Extract "blobs" from the image. A blob is a monochrome area with a minimum size, whose main color belongs to the group of allowed cuboid colors.
2. Find edge points for each blob that define the 2D projection of the cuboid.

[1] Information can be lacking when cube orientations are unfavorable (e.g. head on, or from above), or when one cube hides part of another

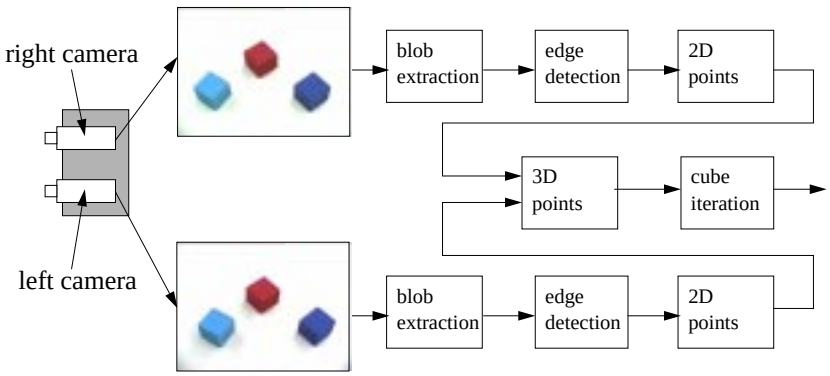


Figure 1: Highest-level structure of the vision system: Images enter from the head cameras, are processed individually to produce 2D vertex information, combined with camera model information to generate 3D points for the stereo synthesis, which generates a cuboid hypothesis as output.

3. Fit a reduced set of lines to the edge points.
4. Find at most 6 intersection points from the set of lines. These should be at the corners of the 2D projection of the cuboid.

Having obtained a set of corner points of the 2D-projection of each cuboid the next stage is to match blobs from the two camera pictures and then find corresponding points of the 2D projections. Once the corresponding points have been decided one has a set of 3D points describing some corners of the cuboid. The final stage is an iteration of the possible cuboid sizes within the set of points to get the best fit. The result is the 3D hypothesis of the cuboid.

3 Blob extraction

The first process performed on an image takes advantage of the colored nature of the objects. The image is scanned quickly for relatively large connected colored shapes, using a coarse-grained grid. The colored shapes are called **blobs**, and are manipulated by their enclosing **box** $(x_{min}, y_{min}, x_{max}, y_{max})$ defined in the pixel space of the image. Further processing is performed solely within the boxes (see Figure 2).

The extraction is performed by splitting the image into a square grid of side ΔG pixels (in our case $\Delta G = 30$), reading the pixel RGB value ρ_{ij} of the pixel (i, j) near the middle of a grid square and checking the angle $\alpha(\rho_{ij}, \rho_c)$, between the pixel RGB and the RGB of a defined color c, where:

$$\alpha(\rho_1, \rho_2) := \arccos \frac{\rho_1 \cdot \rho_2}{|\rho_1||\rho_2|}. \tag{1}$$

Each color $c \in \mathcal{C}$, where $\mathcal{C}$ is the set of predefined object colors, is compared until an angle less than a threshold τ_{blob} (in our case $\tau_{blob} = 10°$) is found, and the associated grid square is said to be **colorful**. If no angle under this threshold is found the grid square is assumed to contain background image only.

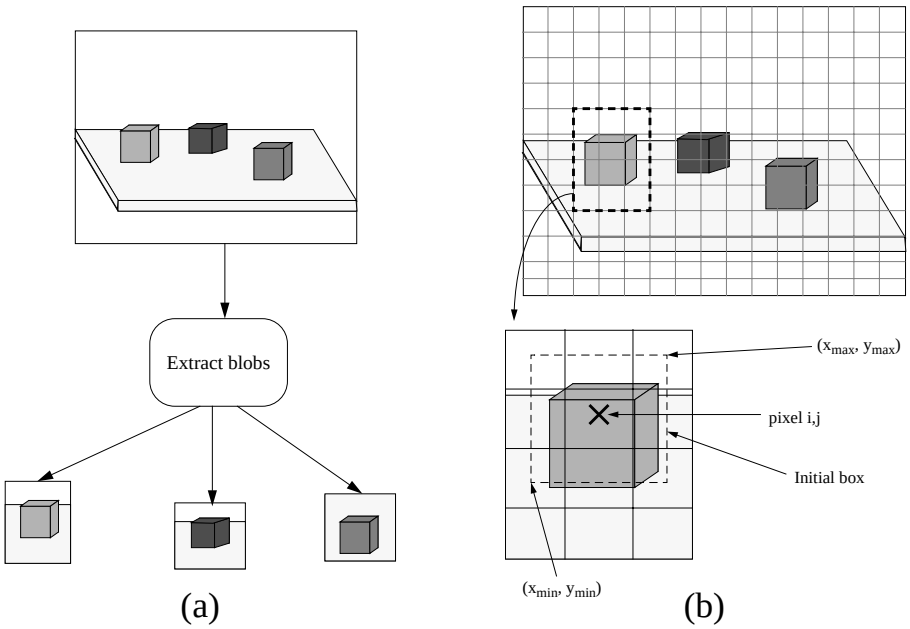


Figure 2: (a) The blob extraction process (b) grid and initial blob construction

When a grid square is found to be colorful a new box $(x'_{min}, y'_{min}, x'_{max}, y'_{max})$ is defined as the boundaries of the given grid square. This box is then immediately expanded to include the nearest halves of all adjacent grid squares (Figure 2b).

Each initial box thus found is called an *intermediate box*. It is then checked against the set of m current boxes $\mathcal{B}$. If it overlaps with any other box associated with the same object color it is subsumed into it, by generating a new bounding box enclosing both old box and intermediate box. If it does not overlap with any box in $\mathcal{B}$ it forms a new box $m + 1$ and is added to $\mathcal{B}$. Initially $\mathcal{B}$ forms the empty set.

Since boxes are only merged when the associated predefined object color is the same, it is possible to have overlapping boxes of different object colors.

Blob extraction requires order of 5 ms real time on a Pentium 2 PC.

4 Edge detection

We split this section into two related phases of the edge detection process. The first phase (sections 4.1–4.3) starts with no information about possible location of edges, and the second phase (section 4.4) uses information gained from the first phase to improve the detection of the edges.

4.1 Basic edge detection

In order to speed up the edge detection process (we have no dedicated hardware for our vision), no 2D image convolution procedures are employed, and a main requirement is to

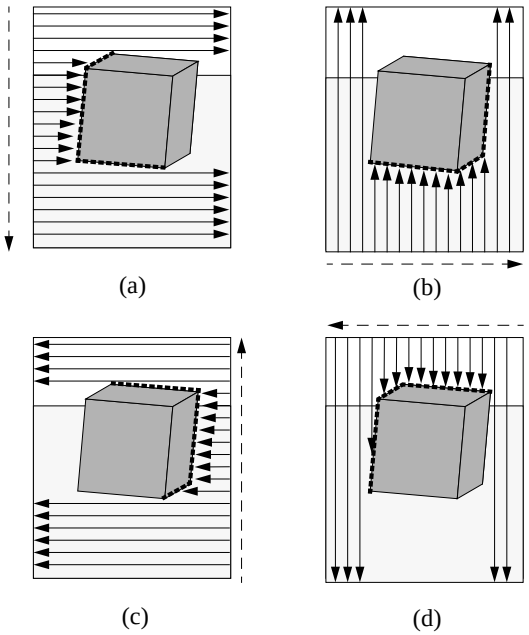


Figure 3: Edge detection is carried out by running along the image in four scan directions and along a number of scan lines in the search directions. The unbroken arrowed lines indicate the scan line direction, and the broken arrowed lines the search direction. (a) Search direction south, scan direction east (b) search direction east, scan direction north (c) search direction north, scan direction west (d) search direction west, scan direction south

keep the number of pixel value accesses to a minimum. The approximate size of an average box is $80 \star 80 = 6400$ pixels, and if every RGB pixel value had to be accessed and processed the time required per box would be too high for real-time processing.

4.1.1 Assumptions made by the algorithm

It is assumed that the 2D projection of the object on the image plane is a single topologically closed surface. It may be concave: this will later be used to determine whether a cube is occluded by another. Furthermore we will be assuming that object edges are defined as transitions between two more or less monochrome surfaces having a minimum size.

4.1.2 Four directions

Each box is processed along a number of **scan lines** separated by $N_{sample} - 1$ pixels in the **search direction** in each of four **scan directions** NWSE from outer edge towards the center (see Figure 3). An **edge point** is discovered by running along a particular scan line and comparing pixel RGB values on the line. Thus, no comparison is made between pixels in successive scan lines.

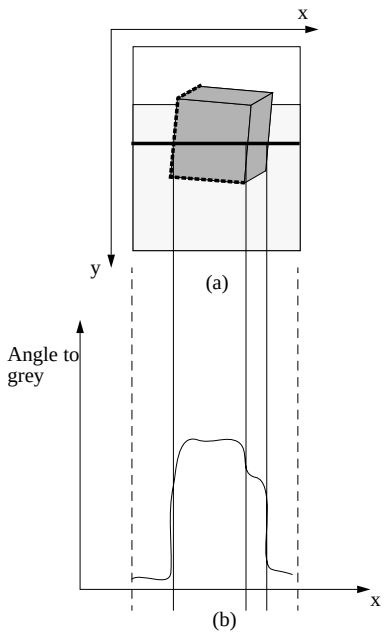


Figure 4: Illustration of how the angle to grey for each pixel may vary along a scan line

4.1.3 Edge location

Each pixel along a scan line has an RGB value. These values must be used to identify the position of an edge. The definition of an edge determines the method used to detect it. Thus, if an edge is defined as the position along a scan line with the maximum rate of change of RGB intensity (however this may be defined), then it is found by defining an appropriate discrete pixel interval for measuring differences, and checking all the values of intensity rate along the scan line. One could also define an edge at the point at which the angle between pixel value and the RGB value for a shade of grey changes from a low value to a high value. Figure 4 illustrates this. The disadvantage is of course that the value can change over a number of pixels before becoming relatively stable again on the object surface. Furthermore, definition as the point at which the rate of change of angle to grey is highest may also not be terribly reliable when the edge becomes more smeared, which is the case for some less vibrant colors and bad lighting conditions, or overlapping blobs and shadows.

4.1.4 The two-surfaces method

We decided on the two-surfaces method for edge detection, which is defined in the following way.

1. **Scan line.** A scan line l defines a sequence of pixel RGB values $l = \langle \rho_1, \rho_2, \ldots, \rho_N \rangle$, where N is the length of the line, and i indicates the pixel index along the line, which is completely contained within the box.

2. **Locate an object surface.** A scan line is progressed, starting at pixel index 1, until a point on the colored blob is found.

   A pixel j in the line l is defined to be on a *blob surface* if the following condition holds:

   $$\alpha(\rho_{j-D/2}, \rho_{j+D/2}) < \tau_{surface} \tag{2}$$

   where D is a parameter defining the minimum pixel extent in one direction of a surface, and could have a variable value depending on the overall size of the blob. Thus a blob surface is only checked along the scan line (i.e. in a one dimensional slice of the 2D surface).

   A pixel j in the line l is part of a *colored surface* if it lies on a blob surface, and the following also holds:

   $$\alpha(\rho_1, \rho_j) > \tau_{bg} \tag{3}$$

   where ρ_1 is the RGB value of the first pixel on the scan line, defined to be the RGB value of the blob background (bg) for this scan line (we assume all blobs lie completely within the borders of the camera image). The *background* of a blob is defined as the areas in the box external to the colored surface of the blob. The transitions from blob surface to background may not always indicate a cube edge (see section 6.1).

   A pixel j in the line l is part of an *object surface* if it is on a colored surface and the following also holds for some $c \in \mathcal{C}$:

   $$\alpha(\rho_c, \rho_j) \leq \tau_{ocolor} \tag{4}$$

   As soon as a pixel of a scan line is determined to be part of an object surface the following tuple is recorded:

   $$(j_{osurface}, \rho_{j_{osurface}}),$$

   where the subscript "$osurface$" denotes object surface, and $j_{osurface}$ is the first scanned pixel found on the surface.

3. **Locate background surface.**

   The scan line is then progressed from $j_{osurface} - D/2$ in the (opposite) direction of decreasing pixel index, until a background surface is found.

   A pixel j in the line l is part of a *background surface* if it on a surface, and the following also holds:

   $$\alpha(\rho_{j_{osurface}}, \rho_j) > \tau_{ocolor} \tag{5}$$

   The result is a tuple

   $$(j_{bsurface}, \rho_{j_{bsurface}}),$$

   where the subscript "$bsurface$" denotes the background surface, and $j_{bsurface}$ is the first pixel found on this surface.

4. **Get edge point.**

   The pixel representing the part of the object edge lying on the scan line, we call the **edge point**. It is to be found somewhere between $j_{bsurface}$ and $j_{osurface}$. Figure 5 shows an example of the progression of the angle with $\rho_{j_{osurface}}$ of the pixels between a background surface and object surface (red).

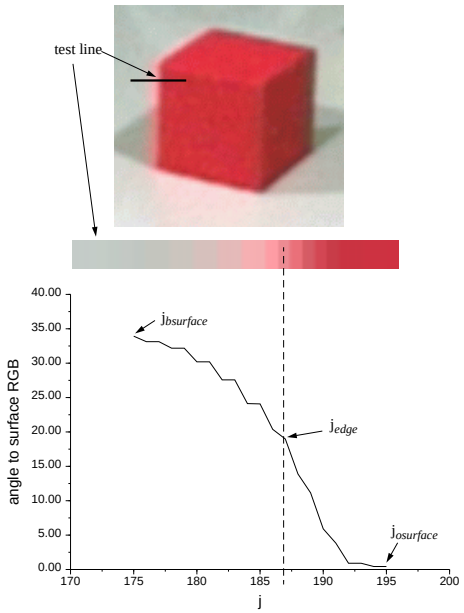


Figure 5: The heavy line in the top picture indicates part of the west–east scan line and under the picture are shown the actual pixels in the line. Shown in the graph are the angles with $\rho_{j_{osurface}}$ of the pixels along the line. The points $j_{osurface}$ and $j_{bsurface}$ are marked on the graph, as is the identified edge point j_{edge}.

It can be seen that even for this relatively good and clear edge transition it is not obvious where to define the edge point exactly. We present two hypotheses for the edge point pixel j_{edge}:

Using angle thresholding. The edge point is at the pixel j_{edge}, where the angle between $\rho_{j_{edge}}$ and $\rho_{j_{osurface}}$ is nearest to half of that between $\rho_{j_{bsurface}}$ and $\rho_{j_{osurface}}$:

$$j_{edge} : |\alpha(\rho_{j_{edge}}, \rho_{ocolor}) - \theta_{edge}| = \min_j(|\alpha(\rho_j, \rho_{ocolor}) - \theta_{edge}|) \tag{6}$$

where

$$\theta_{edge} := \alpha(\rho_{bcolor}, \rho_{ocolor})/2 \tag{7}$$

We show some examples of angle profiles of edges around the test object in Figure 6.

Using intensity thresholding. The edge point is that point with intensity difference to $\rho_{j_{osurface}}$ closest to a constant factor f_{edge} of that between $\rho_{j_{osurface}}$ and $\rho_{j_{bsurface}}$:

$$j_{edge} : |I(\rho_{j_{edge}}, \rho_{j_{ocolor}}) - \Delta I_{j_{edge}}| = \min_j(|I(\rho_j, \rho_{j_{ocolor}}) - \Delta I_{j_{edge}}|) \tag{8}$$

where

$$\Delta I_{j_{edge}} := I(\rho_{j_{bcolor}}, \rho_{j_{ocolor}}) * f_{j_{edge}} \tag{9}$$

The intensity function is given by:

$$I(\rho_i, \rho_j) := |\rho_i - \rho_j| \tag{10}$$

where the euclidean distance metric is used between the 3-component RGB vectors. However, the value of $f_{j_{edge}}$ is not necessarily (as in the angle method) the simple

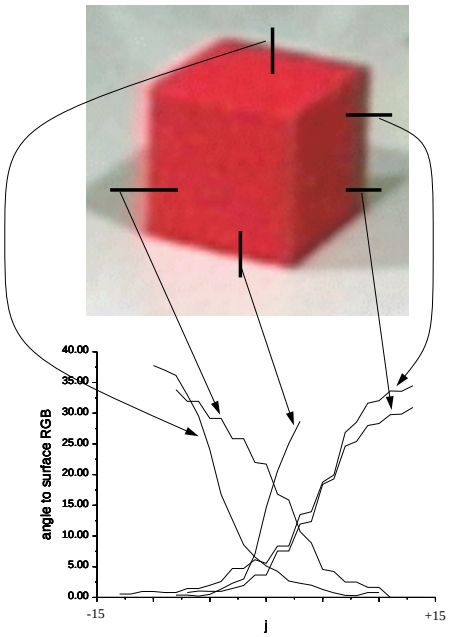


Figure 6: Examples of edge profiles around a test cube. The scan lines are marked as thick black lines, and are matched by the arrows to their angle profiles. The center of each scan line is placed at the pixel judged to be at the edge point.

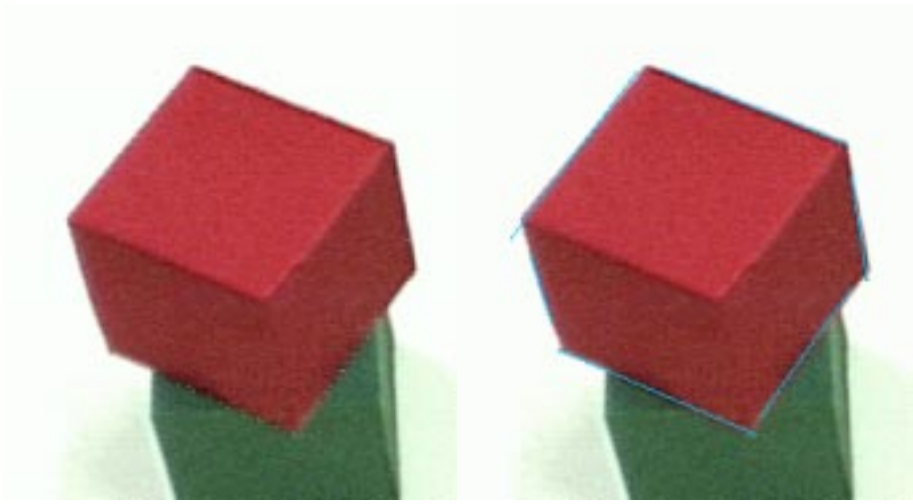
value 0.5. Because of shadows and reflections the correct value can vary greatly. Although variations of optimal value also occur in the angle method, we found the latter more reliable, given our color assumptions.

Each search direction results in a sequence of edge points for the object edges. Some of these object edges are present in more than one search direction. The four search sequences are combined in the order south , east, north, west to form a further sequence $\langle E_1, \ldots, E_{N_e} \rangle$ where $E_i = (u_i, v_i)$. The resulting sequence is already partially ordered along the perimeter of the blob. It is then re-ordered to produce a set of edge points that progressively mark the outer perimeter of the blob. This is termed the **edge point chain**.

4.1.5 Construction of edge point chain

The edge point chain is a sequence of edge points ordered counter-clockwise along the perimeter of the blob. An edge point chain $\mathbf{C} = \langle (u_1, v_1), \ldots, (u_{N_C}, v_{N_C}) \rangle$ is constructed from the sequence of N_e collected edge points in the following way:

1. Set $i = 1$
2. Tag point i
3. Find non-tagged point E_j with smallest euclidean distance to point E_i
4. Store E_j in C, set $i = j$



(a) (b)

Figure 7: Typical result of basic edge point detection. (a) Edge points are green, (b) the edge lines (blue) resulting from the line formation algorithm.

5. Goto step 2

Built into this process is a mechanism for avoiding loops (this can occur when the edge points are all close to each other), escaping from deadends (backtracking) and ignoring points that are too far away.

4.1.6 Result of edge point detection

Figure 7a shows a typical sequence of edge points obtained from the detection process thus described. The scan lines spacing $N_{sample} = 7$ pixels. Section 8 describes the automatic determination of this and other parameters.

You may notice that some scan lines produced no edge point. This is the result of poor surface identification: rather than risk a highly erroneous edge point we skip it completely when one or both surface pixels cannot be found reliably.

4.2 Line formation

Given the edge point chain a set of lines must be generated. This is done by repeated application of a linear regression algorithm, which searches for sequences of points that lie near to straight-line segments in $\boldsymbol{R}^2$.

A full set of such **edge lines** $\mathcal{L}$ is constructed by invoking the recursive function **edge_line_set** as follows:

$\mathcal{L}$ = **edge_line_set** $(\mathbf{C}, \{\})$

edge_line_set (q, l) inserts straight-line segments into the set l. The segments are obtained progressively from the sequence of points q in the continuous 2D space of the image (where the base unit is defined by pixel separation) by application of function **longest_line_chain**. The parameter N_{min} determines the minimum number of points defining a line segment, $corr_{min}$ the minimum acceptable correlation of a segment with the set of points.

longest_fit_chain $(q, len, n_{worse}, corr_{max})$ returns the initial sequence of points in q which maximizes the line correlation $corr_{max}$ and for which $n_{worse} < N_{worse}$. **longest_fit_chain** is a recursive function, and is repeatedly called increasing len by 1 each time. n_{worse} counts the number of successive increases in len that resulted in a total correlation lower than the last maximum.

edge_line_set $(q, l) :=$
if $|q| < N_{min}$
then l
else if corr (**subseq** $(q, N_{min})) < corr_{min}$
then edge_line_set (**remainder** $(q,$ **subseq** $(q, N_{min})), l)$,
else edge_line_set (**remainder** $(q,$ **longest_fit_chain** $(q, N_{min}, 0, -1))$,
$l \cup \{$**line** (**longest_fit_chain** $(q, N_{min}, 0, -1))\})$

longest_fit_chain $(q, len, n_{worse}, corr_{max}) :=$
if $len > |q|$
then q
else if corr (**subseq** $(q, len)) < corr_{max}$
then if $n_{worse} = N_{worse}$
then subseq $(q, len - n_{worse})$
else longest_fit_chain $(q, len + 1, n_{worse} + 1, corr_{max})$
else longest_fit_chain $(q, len + 1, 0,$ **corr** (**subseq** $(q, len)))$

remainder (q, f) returns the sequence q minus the starting sub-sequence f. **subseq** (q, j) returns the sub-sequence consisting of the first j elements of q. **line** (q) returns a line $(\mathbf{m}\ (q), \mathbf{c}\ (q), \mathbf{corr}\ (q))$ fit to the sequence of points q, where:

$$\mathbf{m}\ (q) = \frac{\mathbf{A}\ (q)}{\mathbf{D}\ (q)} \tag{11}$$

$$\mathbf{c}\ (q) = \frac{\sum y_k}{|q|} - \mathbf{m}\ (q)\frac{\sum x_k}{|q|} \tag{12}$$

$$\mathbf{corr}\ (q) = \sqrt{\frac{\mathbf{A}\ (q)^2}{\mathbf{B}\ (q)\mathbf{D}\ (q)}} \tag{13}$$

$$\mathbf{A}\ (q) = \sum x_k y_k - \frac{\sum x_k \sum y_k}{|q|} \tag{14}$$

$$\mathbf{B}\ (q) = \sum y_k^2 - \frac{(\sum y_k)^2}{|q|} \tag{15}$$

$$\mathbf{D}\ (q) = \sum x_k^2 - \frac{(\sum x_k)^2}{|q|} \tag{16}$$

In the above x_k and y_k denote respectively the x and y components of the kth element of q.

An edge line tuple $(m, c, corr)$ defines the locus

$$y = mx + c \tag{17}$$

in the continuous image space.

There are three parameters that control the edge line search procedure: the *minimum number of edge points per line*, N_{min}, the *minimum overall line correlation*, $corr_{min}$ and the *maximum number of consecutive points acceptable whose addition does not improve the correlation*, N_{worse}. The first parameter sets a lower limit on definition points in a line, the second determines when to start constructing a new line, and the third rejects lines of bad correlation. The second is a very important parameter, because it influences the exact point of detection of corners in the blob outline. Figure 8 shows the effect of variation of N_{worse} on the test picture using the edge points shown in Figure 7. It is a difficult parameter to set, since it will depend on the length of the edge in question, and this can be arbitrarily small (e.g. a cuboid viewed from almost directly above). In order to limit the number of arbitrary parameters we set

$$N_{worse} = N_{min} \tag{18}$$

so that this length now determines the minimum edge length in terms of edge points one can reasonably hope to resolve. The resolution limit is, however, lower-bounded (as indicated in Figure 8) and other measures must be utilized to handle difficult cases (see section 9 for a discussion of this and other algorithm limitations).

What we can deduce from the above parameters is a lower bound on the acceptable pixel length of a line L_{min}:

$$L_{min} = (N_{min} - 1) * N_{sample} \tag{19}$$

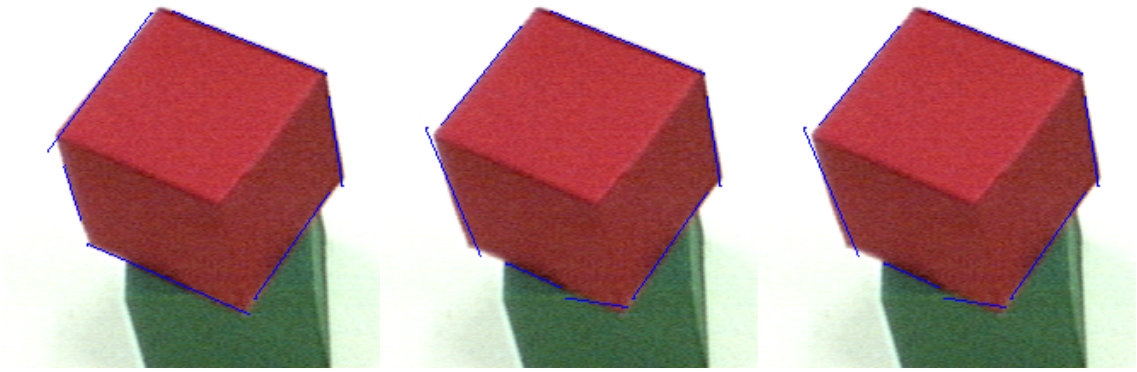
For the typical parameter values we used this lay in the region of 15–30 pixels.

4.3 Finding vertices

Vertex formation proceeds in the following way:

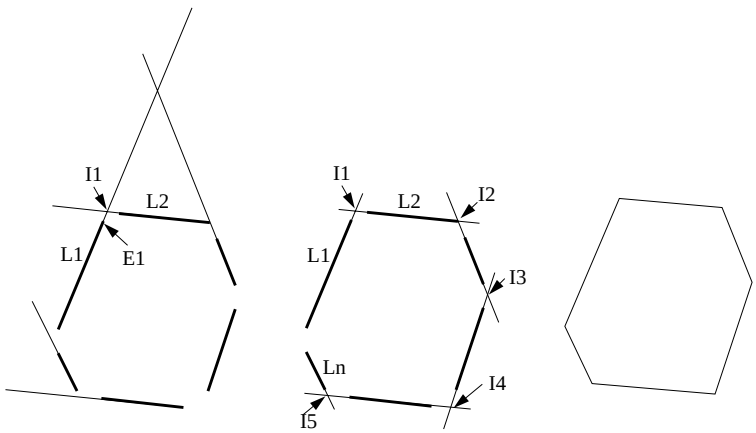
1. **Find intersections.** The lines are used to find the corners of the cross-section of the cube. This is done by considering initially an arbitrary line and intersecting with all other lines (using their extensions if necessary). The intersection closest to the nearest endpoint of the original line is chosen (Figure 9a). The next line to be considered is the line that successfully intersected, and the endpoint in question is then its other one. This process continues until all lines are used up. In this way all lines will be intersected at both ends apart from the first one and the last one (Figure 9b). These are then intersected with each other to produce a closed figure (Figure 9c).

2. **Check angles: cuboid assumption.** Now the resulting closed figure is considered and the angles of the corners are investigated. If an angle is too acute it either



(a) (b) (c)

Figure 8: Influence of the line search parameter N_{worse} on the line construction algorithm: (a) 5, (b) 3, (c) 2. If the parameter is too low the edge is undershot. If the parameter is too high this affects unsharp and short edges adversely (not in illustrated case).



(a) (b) (c)

Figure 9: (a) A line is chosen (L1) and extended in the direction of a chosen endpoint (E1). All other lines (heavy) are extended (thin) in the direction that intersects with this extension. The intersection (if any) nearest to E1 and the line it came from (L2) are then stored. (b) This process is continued until either no intersection point is found (= failure) or all lines are used up. The two dangling lines L1 and Ln are then intersected in the dangling directions to close the figure (c).

indicates that the intersection was false, or that the object is partly hidden by another object. Discussion of the case of hidden objects is delayed to section 6.1. In either case the corner in question is removed, and the two corners that were adjacent to it are joined by a straight line. The angles of corners are checked in this way until no more changes are made.

3. **Reduce to 6 corners.** The resulting form should be a 2D projection of a cube. Thus if there are more than six corners present then the excess ones must be removed. This is ranked on the basis of their estimation error, calculated from the other detection phases (see section 6.2 for a summary of errors gathered from the recognition process). If there are less than 6 corners remaining it may still be possible to use some in a future 3D synthesis. Such cases occur when only four edges may be identified (view practically face-on). If only 5 corners remain it may be possible to make a reasonably good guess at the missing one (see section 6.1).

The closed figure is defined by the ordered set of vertices $\mathcal{V} = \langle V_1, \ldots, V_{N_{verts}} \rangle$ where N_{verts} is the number of corners and V_i indicates the 2D pixel position of corner i in the continuous image plane.

4.4 Guided edge detection

The preceding method of generating closed figures used what we term the "basic edge detection" method for locating the edge points. This method can however be improved upon, by allowing a further processing phase that uses the closed figure generated above to *guide* and focus the search for the edges in a particular region of the blob. The advantage is that since the areas are smaller it is possible to use a lower value of N_{sample} to obtain more candidate edge points per line, and also, since the approximate positions of the edges are known, more difficult edges may be better localized and the lines better fitted.

The process itself is quite simple. Instead of considering an entire blob, N_{verts} regions $\mathcal{R} = \{R_1, \ldots, R_{N_{verts}}\}$ are chosen, defined by:

$$R_i = Rect(V_i, V_{i+1}) \qquad i \in (1, N_{verts} - 1) \tag{20}$$

$$R_i = Rect(V_i, V_1) \qquad i = N_{verts} \tag{21}$$

where $Rect(V_i, V_j)$ defines a rectangle with 2 opposing corners at V_i and V_j respectively, which is additionally blown up if necessary to have a minimum width and height of D_{rect}.

Each region R_i is taken, as well as the first hypothesis of the line inside it, and the two-surfaces scan line method is repeated (section 4.1.4) using a lower value of N_{sample}, and taking the value of the first line hypothesis at the scan lines to concentrate the two-surfaces search around the first hypothesis for the blob edge.

Figure 10 show a typical result of using this method to improve the closed figure obtained using the basic edge detection, and Figure 11 compares the two closed figures generated directly.

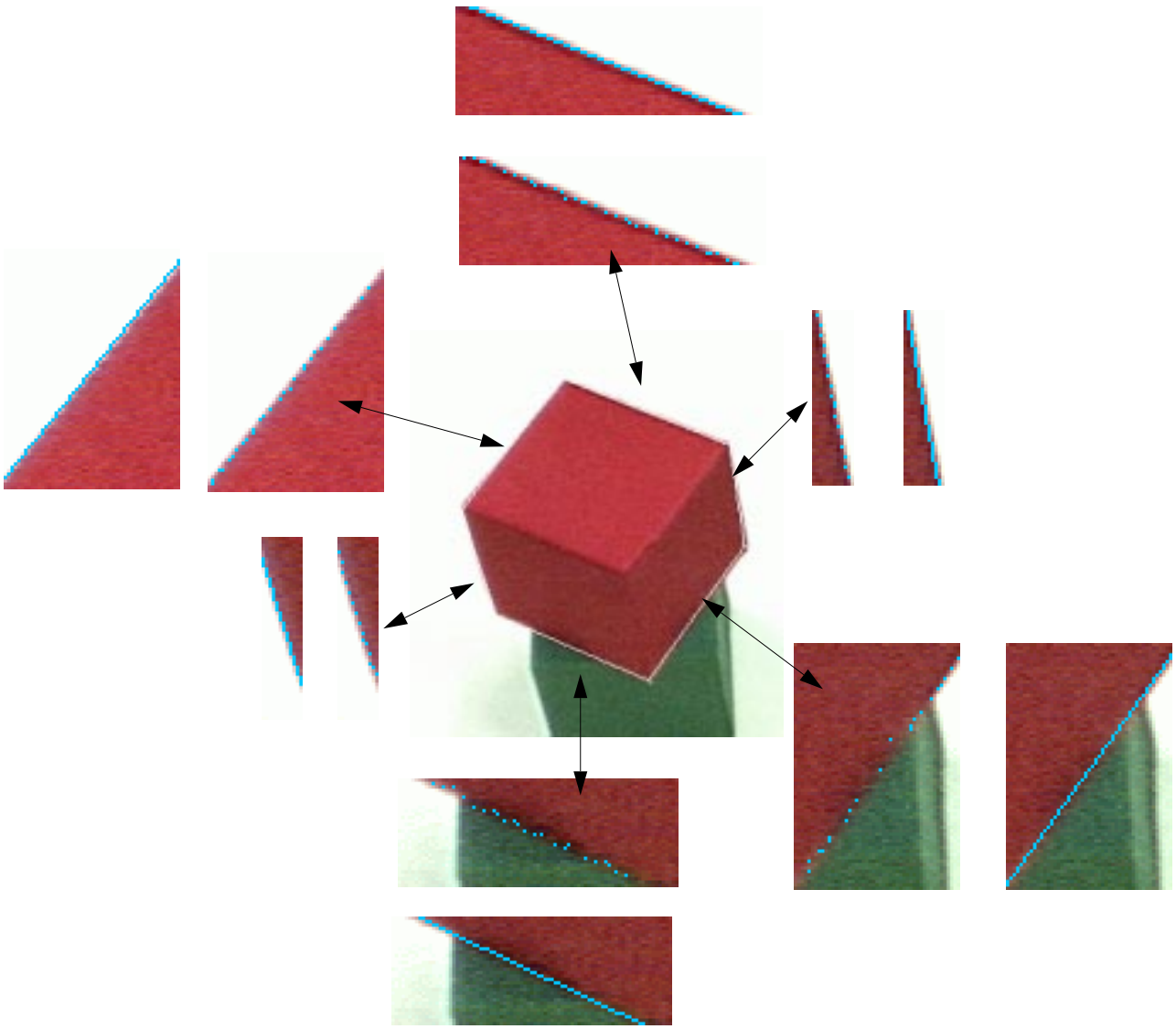
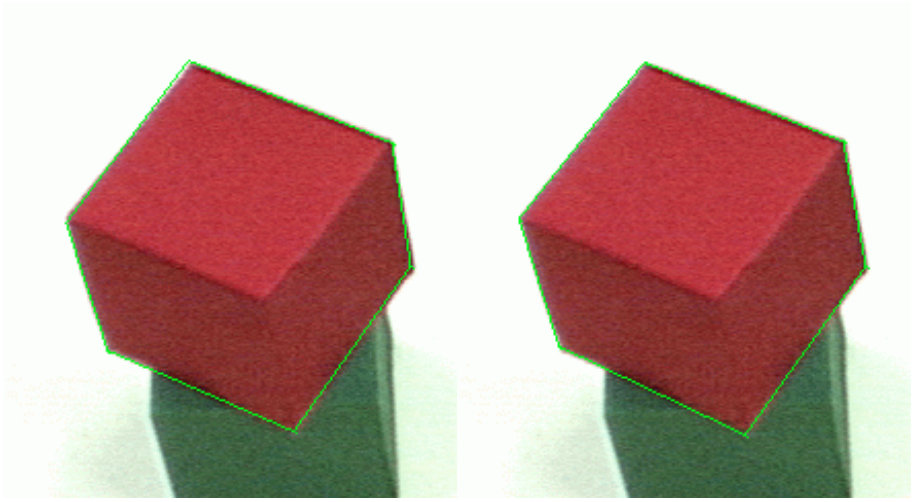


Figure 10: Edge detection using edge guides: The initial edge detect process produced vertices connected by the white lines shown (middle picture). They are used to constrain and guide the search for better edge points (inner ring of smaller pictures) and consequently edge lines (outer ring) are produced.



(a) (b)

Figure 11: (a) closed figure obtained using basic edge detection method, and (b) using guided method with information from (a)

5 3D-synthesis

The next step in the processing is to take the 2D closed figures of shapes from two calibrated cameras and combine them to form a 3D model of an object.

5.1 Calibrated cameras

In the following we refer to a point in the world coordinate system by $(x, y, z)^T$, and the continuous image coordinate system by (u, v).

Each head camera is fully calibrated relative to the world coordinate system within which the workspace and robot arms are also defined. The cameras are taken to approximate to the pinhole model, and the result of the calibration [2, 3] produces a matrix M decomposable as a product of two further matrices I and E:

$$M = I * E \tag{22}$$

$$I = \begin{bmatrix} \alpha_u & 0 & u_0 & 0 \\ 0 & \alpha_v & v_0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{23}$$

$$E = \begin{bmatrix} R & T \\ 0 & 1 \end{bmatrix} \tag{24}$$

I is the *intrinsic parameters* matrix: α_u and α_v are scaling factors in the two axes of the camera CCD plane, and (u_0, v_0) are the coordinates of the image center. E is the *extrinsic parameters* matrix: R is the 3×3 rotation matrix of the CCD plane and T is its translation vector relative to world coordinates.

The matrix M allows the following transformation to be made:

$$\begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = M \begin{pmatrix} sx \\ sy \\ sz \\ s \end{pmatrix} \tag{25}$$

where the vector $(sx, sy, sz, s)^T$ represents all points on a line in world coordinates intersecting the image center.

5.2 Generating a 3D point

Given the continuous pixel coordinates of corresponding points on an object from two cameras, (u_1, v_1) (from camera 1) and (u_2, v_2) (from camera 2), the 3D world coordinates (x, y, z) of this object point can thus be calculated from the two camera matrices M_1 (camera 1) and M_2 (camera 2) by finding the intersection of the respective lines.

This calculation is actually performed (see [2, 3]) by iteratively solving the linear system of 4 equations with 3 unknowns obtained from the two sets of matrix equations similar to eqn. (25).

The intersection is in general not going to be exact, and so there will also be an associated error e generated, that gives the uncertainty in world units of the 3D point suggested. Section 6.2 discusses the use for such a parameter.

5.3 Corresponding points

To generate a full 3D model a minimum number of points are required to constrain the cube location. We do not assume any particular orientation of the cube, and therefore need at least four non-planar points to specify a given cube.

The result of our detection process provides us with 2 sets of vertices, $\mathcal{V}_1$ and $\mathcal{V}_2$, containing N_{verts1} and N_{verts2} vertices respectively. The task now is to match up points between the sets that correspond to the same points on the actual object. Since we are locating the corners of the cube with the detection method we shall assume that both sets contain at least as many corresponding points as the set with the lower number of points. Thus every point in the set containing less points should be matched up with a point in the other set.

The point clusters are shifted so that their centroids coincide. Matching points are located using a distance metric in this relative space. Thus points closest in the superposed clusters are assumed to correspond. This assumption is only approximately feasible and only then for cases when the objects are sufficiently distant from the cameras.

A more rigorous way is to compare all points, and choose those pairs with the lowest values of error e from the 3D point generation. However, since the first method sufficed for our examples we used it for the experiments (and times) described in this paper.

5.4 Generating a 3D cube model

After the corresponding points have been found, a set of 3D points $\mathcal{P} = \{P_1, \ldots, P_{N_{corners}}\}$ can be generated as described in section 5.2. The final step in model generation is then to fit a 3D description of a cuboid to the set $\mathcal{P}$.

5.4.1 Cube iteration

A candidate cube is chosen by comparing the maximum extension of the set of points $\mathcal{P}$ with the long diagonals of each cube in the set $\mathcal{C}$ that have the correct color. The best match to $\mathcal{P}$ is then found using a simple iteration process. In the following the location (position and orientation) of the candidate cube is represented by the tuple $(\boldsymbol{p}, \boldsymbol{r})$ where $\boldsymbol{p} = (x_c, y_c, z_c)$ defines the position of the cube center and $\boldsymbol{r} = (\theta, \phi, \psi)$ the rotation of the cube with respect to the world coordinate axes (x, y, z).

The candidate cube's position $\boldsymbol{p}$ is first set to coincide with the centroid of $\mathcal{P}$. This results in cube location $(\boldsymbol{p}_{centroid}, \boldsymbol{r}_{start})$, where $\boldsymbol{r}_{start}$ is some initial random cube orientation (we set it to $(0, 0, 0)$).

The cube's location is then iterated to fit the point set:

$$\textbf{final\_loc}_{\ cube,\mathcal{D}} := (\textbf{final\_pos}\ (\boldsymbol{p}_{centroid}, \boldsymbol{r}_{start}, s_{trans\_start}),\ \textbf{final\_rot}\ (\textbf{final\_pos}\ (\boldsymbol{p}_{centroid}, \boldsymbol{r}_{start}, s_{trans\_start}), \boldsymbol{r}_{start}, s_{rot\_start}))$$

The recursive functions used above are defined as follows:

final_pos returns the final position obtained with the smallest step size $s \geq s_{trans\_stop}$ starting from $\boldsymbol{p}_{from}$ with a given initial step size s and a fixed orientation $\boldsymbol{r}_{const}$.

best_pos returns the best position $\boldsymbol{p}_{best}$ of the given cube with respect to $\mathcal{P}$ for a starting position $\boldsymbol{p}_{from}$, a fixed step size s and a fixed rotation vector $\boldsymbol{r}_{const}$.

$D_{all} = \{-1, 0, 1\}^3$ is the set of (27) displacement vectors.

$\boldsymbol{W} = \begin{pmatrix} w_1 & 0 & 0 \\ 0 & w_2 & 0 \\ 0 & 0 & w_3 \end{pmatrix}$ where w_1, w_2, w_3 are rotation angles about each world coordinate axis.

final_pos $(\boldsymbol{p}_{from}, \boldsymbol{r}_{const}, s) :=$
if $s < s_{trans\_stop}$
then $\boldsymbol{p}_{from}$
else **final_pos** (**best_pos** $(\boldsymbol{p}_{from}, \boldsymbol{r}_{const}, D_{all}, s, 0, \boldsymbol{p}_{from}), \boldsymbol{r}_{const}, s/2)$

best_pos $(\boldsymbol{p}_{from}, \boldsymbol{r}_{const}, D, s, bestfit, \boldsymbol{p}_{best}) :=$
if $D = \{\}$

then $\boldsymbol{p}_{best}$
else **if fitness** $(\boldsymbol{p}_{from} + s \cdot \textbf{sel}\ (D), \boldsymbol{r}_{const}) > bestfit$
then **best_pos** $(\boldsymbol{p}_{from}, \boldsymbol{r}_{const}, D \setminus \{\textbf{sel}\ (D)\}, s,$
fitness $(\boldsymbol{p}_{from} + s \cdot \textbf{sel}\ (D), \boldsymbol{r}_{const}), \boldsymbol{p}_{from} + s \cdot \textbf{sel}\ (D))$
else **best_pos** $(\boldsymbol{p}_{from}, \boldsymbol{r}_{const}, D \setminus \{\textbf{sel}\ (D)\}, s, bestfit, \boldsymbol{p}_{best})$

final_rot returns the final rotation (orientation) obtained with the smallest step size $s \geq s_{rot\_stop}$ starting from $\boldsymbol{r}_{from}$ with a given initial step size s and a fixed position $\boldsymbol{p}_{const}$.

best_rot returns the best rotation (orientation) $\boldsymbol{r}_{best}$ of the given cube with respect to the set of points $\mathcal{P}$ for a starting rotation $\boldsymbol{r}_{from}$, a fixed step size s and a fixed position vector $\boldsymbol{p}_{const}$.

final_rot $(\boldsymbol{p}_{const}, \boldsymbol{r}_{from}, s) :=$
if $s < s_{rot\_stop}$
then $\boldsymbol{r}_{from}$
else **final_rot** $(\boldsymbol{p}_{const}, \textbf{best\_rot}\ (\boldsymbol{p}_{const}, \boldsymbol{r}_{from}, D_{all}, s, 0, \boldsymbol{r}_{from}), s/2)$

best_rot $(\boldsymbol{p}_{const}, \boldsymbol{r}_{from}, D, s, bestfit, \boldsymbol{r}_{best}) :=$
if $D = \{\}$
then $\boldsymbol{r}_{best}$
else **if fitness** $(\boldsymbol{p}_{const}, \boldsymbol{r}_{from} + s \cdot \textbf{sel}\ (D) \cdot \boldsymbol{W}) > bestfit$
then **best_rot** $(\boldsymbol{p}_{const}, \boldsymbol{r}_{from}, D \setminus \{\textbf{sel}\ (D)\}, s,$
fitness $(\boldsymbol{p}_{const}, \boldsymbol{r}_{from} + s \cdot \textbf{sel}\ (D) \cdot \boldsymbol{W}), \boldsymbol{r}_{from} + s \cdot \textbf{sel}\ (D) \cdot \boldsymbol{W})$
else **best_rot** $(\boldsymbol{p}_{const}, \boldsymbol{r}_{from}, D \setminus \{\textbf{sel}\ (D)\}, s, bestfit, \boldsymbol{r}_{best})$

sel denotes the selection function of set theory. The selection function defined on the set of non-empty subsets of D_{all} returns an (arbitrary) element of an argument set $D \subseteq D_{all}$.

The function **fitness** $(\boldsymbol{p}, \boldsymbol{r})$ calculates the goodness of cube fit to the point set $\mathcal{P}$ in the following way:

Each point in $\mathcal{P}$ is matched progressively to the nearest of the 8 corners of the cube defined by the location $(\boldsymbol{p}, \boldsymbol{r})$ using a euclidean distance metric. "Progressively" means that the pair (point, corner) that is nearest is paired off first, removing the given corner from the set of possibilities for the other points in $\mathcal{P}$. The average euclidean distance of all the pairs defines then the fit.

Thus first the cube is shifted over to the general region of the points, then it is iteratively matched to the points using decreasing translational shifts along and then decreasing rotational turns about three orthogonal world axes.

Four parameters control the iteration: $s_{trans\_start}, s_{trans\_stop}, s_{rot\_start}$, and s_{rot_stop}. They determine how large the iteration step is to be for both translation and rotation iteration, and how small it can become before the iteration terminates. We introduce another parameter, fit_{good}, which is a lower bound for the cube fit. This allows the iteration procedure to be interrupted at any time if the fit is good enough for our purposes. This is typically only useful for saving time if the iteration has already reached a successful level.

6 Refinements to the algorithm

6.1 Incomplete 2D projections

An incomplete projection is one that has less than 6 corners, and can arise in the following situations:

- The object itself is partially obscured by another object, which made the edge identification either impossible or simply leads to the generation of "false" corner points:
  - A **true** corner point is one formed from the intersection of two adjoining edges that correspond to real edges of the cube.
  - A **false** corner point is one formed from the intersection of two edges where at least one does not correspond to a real edge of the cube

  Figure 12 illustrates this.
- The object is observed practically "head-on", so that the 2D projection is a 4– or 5–sided polygon, instead of the general case of 6. In such cases the missing corner(s) are normally not completely invisible – they have just very obtuse angles. We deal with the 5–sided case in the following way:
  - perform basic edge detection as usual, generate the 5-point figure
  - assume a symmetrical 2D projection
  - assume the missing point to lie somewhere between the points joining the current longest side
  - use the assumed symmetry to generate the new point by construction
  - use the resulting 6-point figure to continue with the guided edge detection

  This worked very well for our cases. The error resulting from assuming a perfect symmetry in the 2D projection is usually removed after the guided edge detection. The 4–sided case can not be handled using the symmetrical construction, but we found that for many cube positions, when the second camera can produce a 6-point figure, the 3D hypothesis is still quite good (only 4 non-planar 3D points are required to define the cube unambiguously).

6.2 Errors and estimates

At every stage of the processing, from blob generation to final 3D hypothesis, there is the possibility of collecting errors and estimates to do with the accuracy of processing in a particular stage. In this section we will list them and their meaning, and in section 9 show how they may be used to construct a *reflective actor*.

- **Edge points.** Each edge point generated using the two surfaces method has an associated error. This error represents the uncertainty that an edge is at that point.

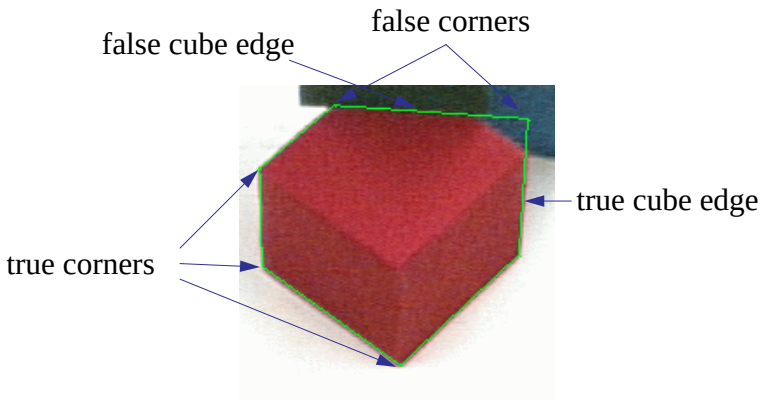


Figure 12: An object's closed figure after edge detection, showing the difference between a true corner point and a false corner point.

Thus if an edge is sharp there will be less uncertainty than if it is blurred. This is typified by the width of the edge, which in turn is given by the pixel distance between the points representing the two surfaces. Thus the profiles in Figure 6 are characterized not only by their height, but also by their width.

- **Edge lines.** An edge line is calculated from the edge points using a simple least squares fit method. The correlation of the line with the set of points is an indication of the error associated with it. This value should be suitably converted to provide an error for the line gradient (since that is the important indicator in this case).

- **Vertices.** A vertex is formed by the intersection between two lines. The uncertainty in the gradient in the lines can be used to estimate the possible deviation of the vertex from the value found using the best fit lines. This is done in the following way: a line is tilted in the negative gradient direction from the best fit by the gradient error, and intersected with the other line which is also tilted in the negative gradient direction. The same is done for the other three intersection combinations, and the average deviation of the four resulting intersection points from the original vertex point calculated. This represents then the estimated error value for the vertex point.

- **3D points.** The two sets of vertices from the two cameras are combined to form 3D points using the calibrated camera models. There are two ways of calculating the error of these points. One way is to calculate a hypothesis sample for a given 3D point by randomly shifting the two corresponding vertex points within the vertex error limits and then to estimate a mean error value from the variance of this sample. Another way is to use the ideal vertex points and the value e from section 5.2 as the error value. In section 9 we show how both may be used to provide different types of reflection values for the hypothesis procedure.

- **Cube fitting.** The cube fitting procedure fits a given cube into the set of 3D points. As for the 3D points there are two indicators of the cube fit error. Firstly the 3D point errors can be used to generate some sets of possible 3D points, cubes fitted to these points and then the deviation between these hypotheses used to calculate an overall error estimate. Secondly the error can be derived from the fit function (section 5.4.1) value.

7 Performance tests

7.1 Measurement of correctness

It is difficult to determine how correct the final hypothesis really is, since we do not know ourselves the exact location of the cuboid in world coordinates. The reason for this is that the world coordinates are for JANUS effectively defined by the manipulators, and the normal success criterion is the precision with which the robot can grasp the cuboid in question. This is however also no real indication of the correctness of the visual processing, because the precision of the manipulation is also a contributory factor.

For these reasons we used an indirect method for determining the correctness of the visual processing. The method works by projecting the corners of the hypothesized cube back through the two camera models, and comparing these with the original image. This can only be done by hand. The correctness is then given by the average pixel difference between all external vertices and their projections.

7.2 Standard cube results

In order to be able to compare different situations we define a "standard cube" as one that presents an optimal image for both cameras. Our standard cube was defined as follows:

- Color: red
- Size: 70 mm
- Orientation: such that the projected blob has six edges with approximately the same length
- Lighting: strong

This standard cube is like the one we have been using in this paper to illustrate various stages of the edge detection process. In the case of a standard cube it suffices to show a single result. Figure 13 shows the 2D vertices obtained from the guided edge detection, together with the projection of the hypothesis back onto the respective image planes for the two cameras. The slight discrepancies between 2D vertices and corners of the projected cubes can be traced both to the edge detection inaccuracy and also the camera calibration inaccuracy. The result is very good, and the generated hypothesis is more than accurate enough for our robot. The correctness value was calculated to be 1.5 units (pixel separation). The location precision is effectively order of millimeters (although this claim can only be derived from robot grasping experiments). This is however an ideal case, and the following experiments investigate behavior for non-ideal cases.

7.3 Dependence on orientation

The orientation of the cube is the most critical factor for this algorithm. As shown in Figure 14 there comes a point where the algorithm cannot produce useful hypotheses. This

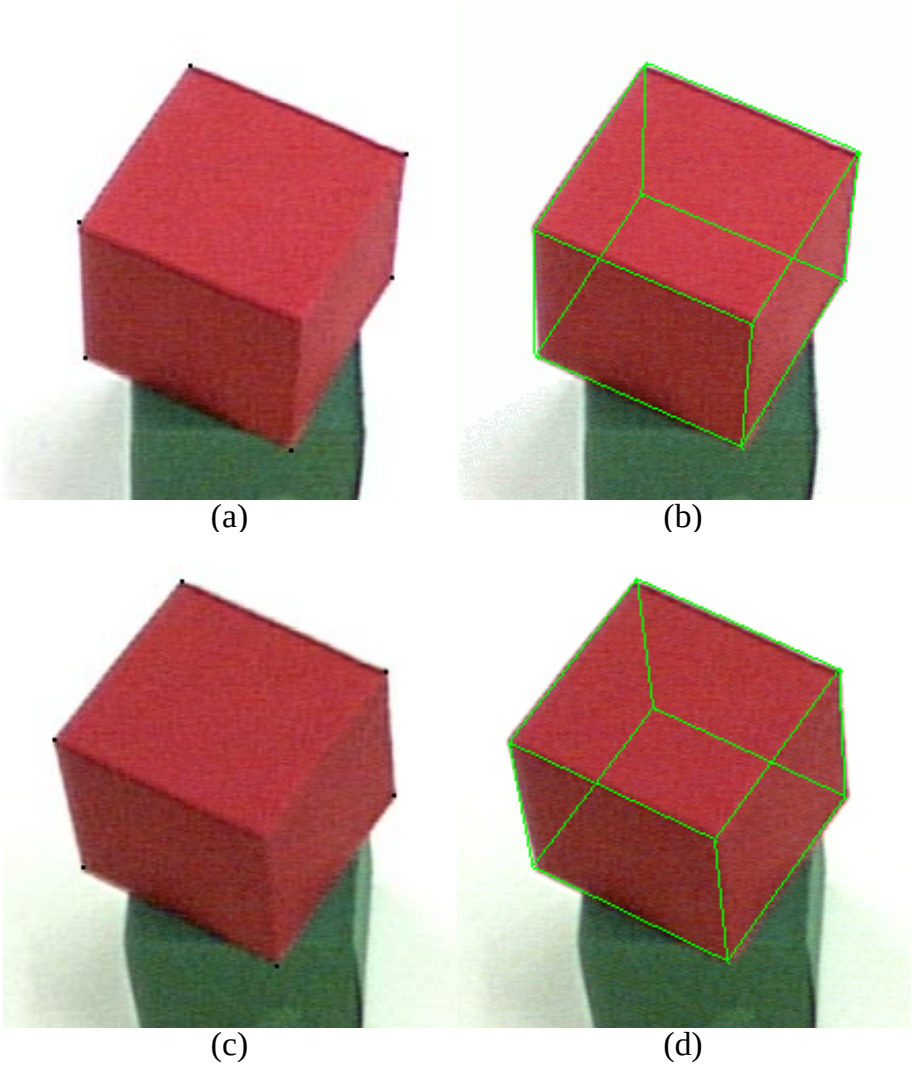


Figure 13: Result for a standard cube, left camera image: (a) 2D vertices, (b) hypothesized cube projected onto image plane, right camera image: (c) 2D vertices, (d) hypothesized cube projected onto image plane. The 2D vertices are black dots.

is in general when the cube is turned less than about 20 degrees to one of the cameras (although it may be possible to get usable hypotheses for hard cases too, as Figure 14d shows). As we explain in section 9 one of the most important characteristics of a useful algorithm is that such limiting cases can be recognized and appropriately handled.

7.4 Dependence on lighting

It was possible to produce three different artificial lighting conditions in our lab—all lights on, two-thirds of the lights on, and one third of the lights on. Results are shown in Figure 15 for the left camera (right camera is similar). It can be seen that lighting intensity is not terribly critical. Correctness for all three pictures is 1.5 (the bad point top left is counterbalanced by very good points for the other corners). For the low lighting condition it was however necessary to halve the surface parameters τ_{ocolor} and τ_{bg}. This may however be automized (section 8).

7.5 Dependence on blob size

As is to be expected, the larger the blob is in the image plane the better its resolution and the better too its edge detection. Figure 16 shows hypotheses for the standard cube for increasing blob size. We varied the blob size by changing the distance of the cube from the camera, which brings in another important effect.

As the cube nears the camera the camera model approximates less well the pin-hole model of a camera. This has the effect of introducing distortion into the 3D points. The effect is small but can be seen in the example. The last picture was for a distance of about 40 cm from the camera. The series of pictures also nicely tests the simple vertex matching method. No problems were encountered with the approximation, although the two camera images could become very different (Figure 17). In this figure can also be seen the different image quality obtained from the two cameras. The right camera was somewhat worse than the left one.

The main effect of blob size is the time taken to process it. Figure 18 shows the processing time as a function of box diagonal size. The time taken scales more or less linearly with box diagonal size, and not exponentially, as one would expect. The reason for this is that for the range of sizes considered the variable sample line spacing N_{sample} (to be described in section 8) scaled down the amount of work to be done for bigger blobs.

7.6 Dependence on cube color

Figure 19 shows results for standard cubes of different colors. We found that the only parameters that needed to be adjusted to achieve these good results for all colors were once again the surface parameters τ_{ocolor} and τ_{bg}.

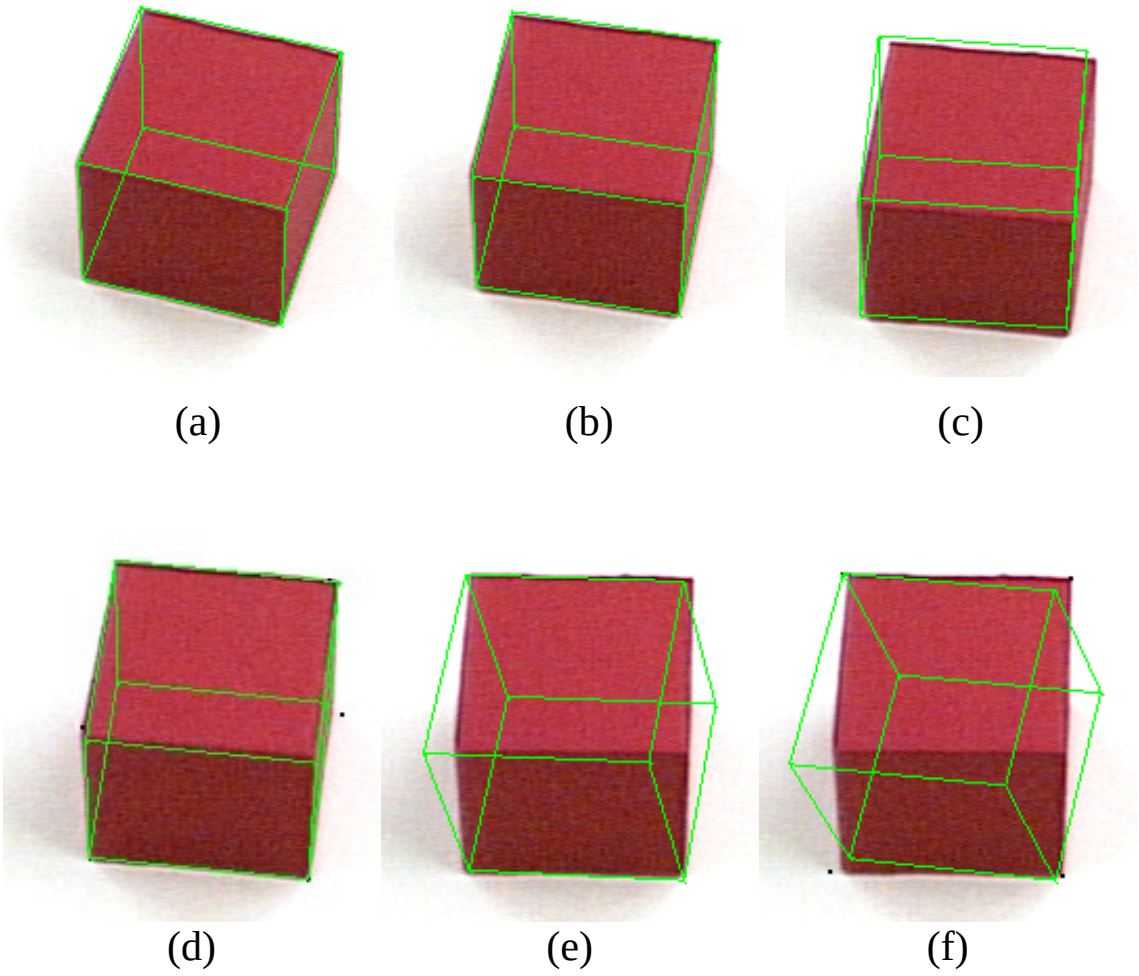


Figure 14: Examples of decreasing projection angle of cube on image plane. (a) and (b) are still good, but (c) brings in errors due to the uncertainty of the middle vertices, which sometimes goes well (picture (d) is the same cube with a different cube fit iteration result). The last cases (e) and (f) demonstrate the extreme where the middle vertices are invisible to the algorithm, and the only 3D points to go on are the remaining four. Since these lie on a plane, various almost equally valid hypotheses may be made. Correctness values: (a) 1.5, (b) 2.2, (c) 4.1, (d) 1.5, (e) 8, (f) 12.

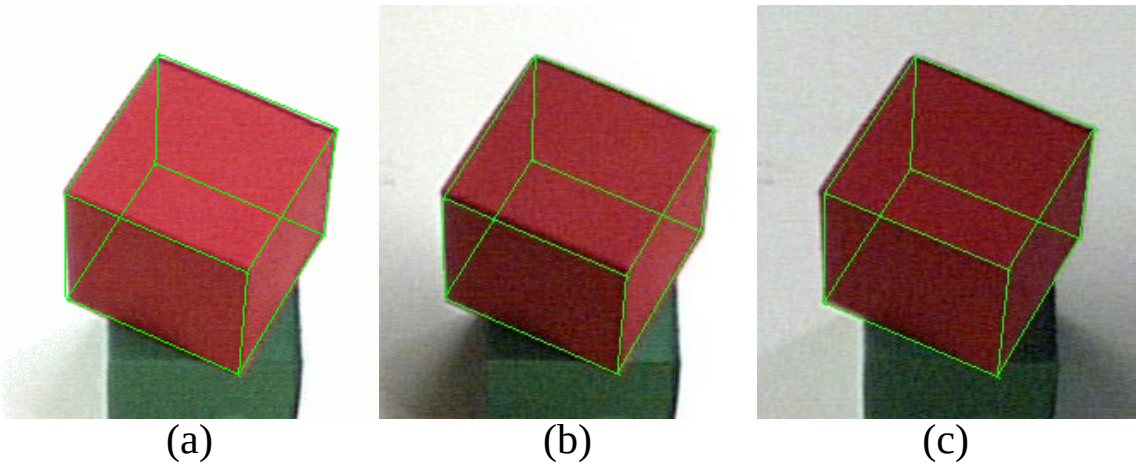


Figure 15: (a) full lighting, (b) two-thirds lighting, (c) one-third lighting. Shown are the 2D projections of the hypothesized cubes on the left camera image plane.

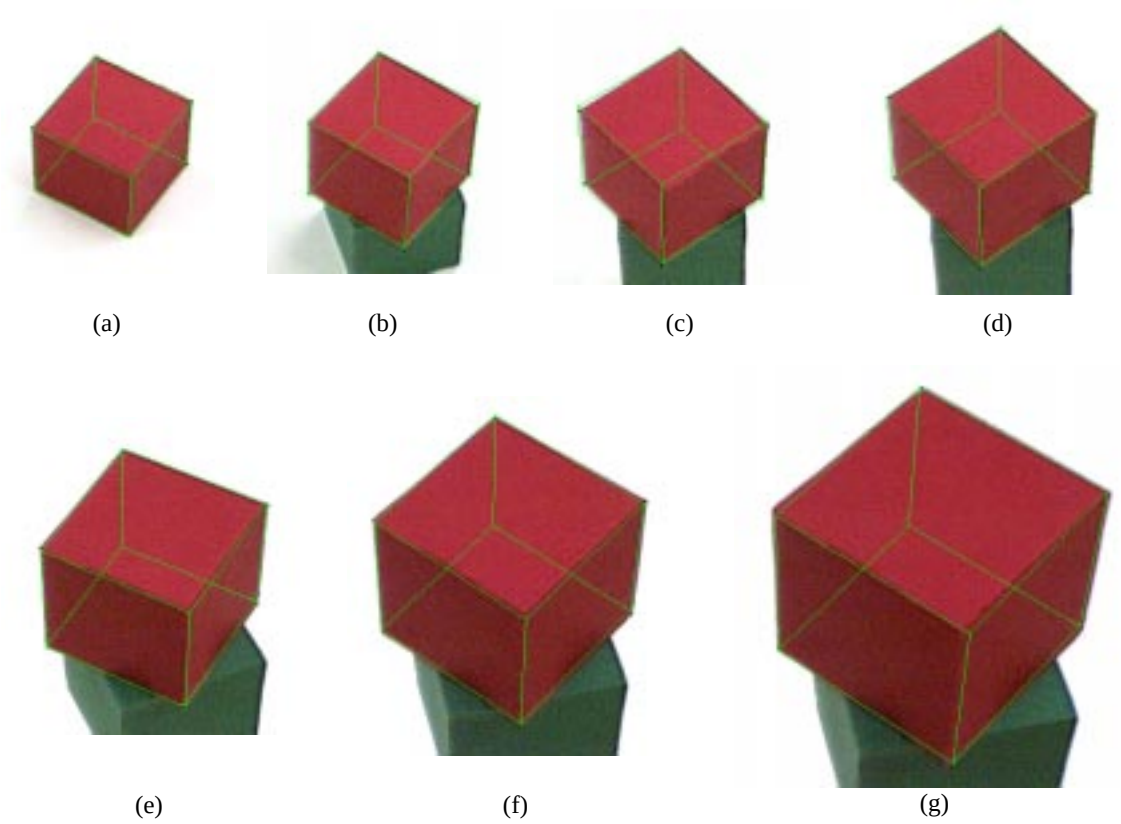


Figure 16: Dependence of edge hypotheses on blob size.

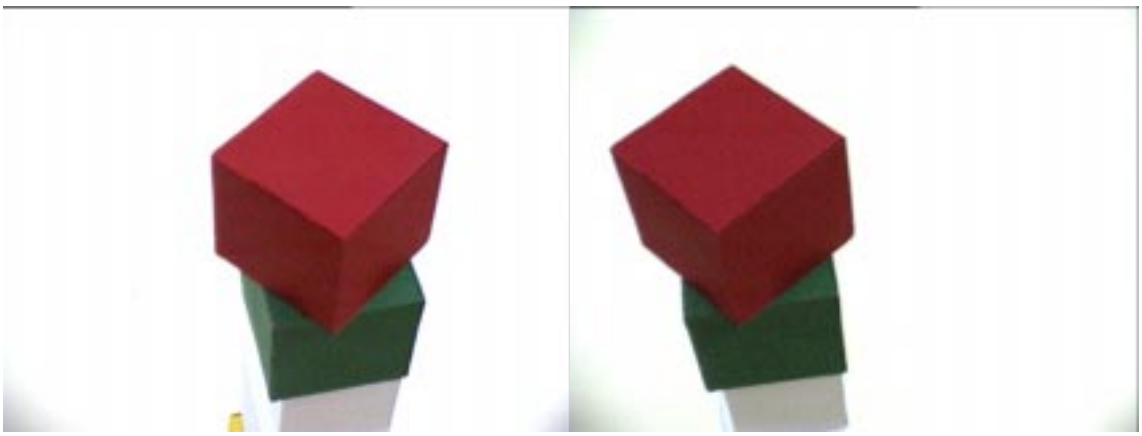


Figure 17: Left and right images for cube about 40cm distant.

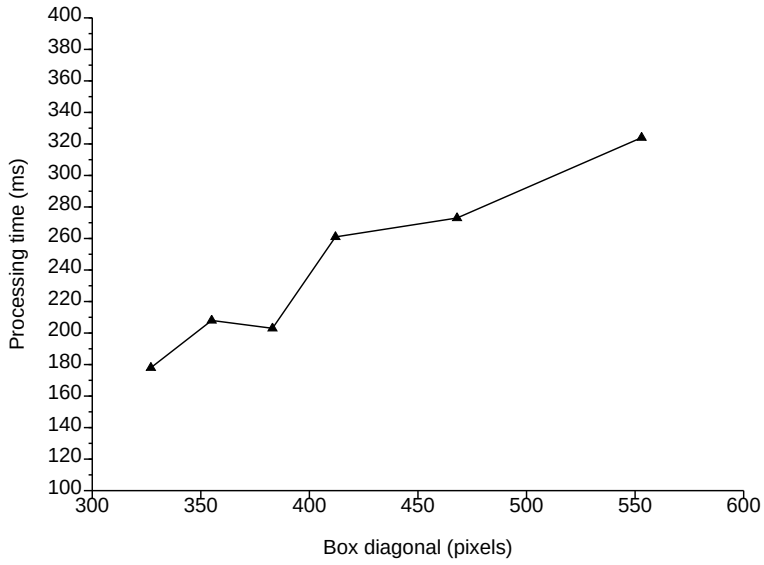


Figure 18: Dependence of processing time on box diagonal

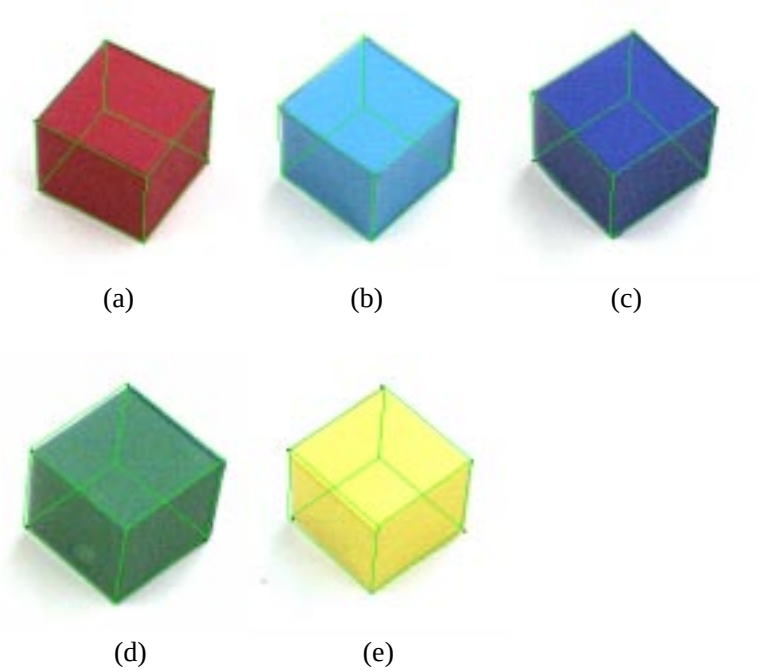


Figure 19: Results for five standard cube colors: (a) red (b) light blue (c) dark blue (d) green (e) yellow

7.7 Dependence on shadows

We found that the algorithm seems to be quite resistant to shadowy conditions (see Figure 20). The edges that overlap onto shadows may become more error-prone and broader, but the overall effect is, due to the color surfaces method, not greatly disturbing. For the overall position heavy shadows produce order of millimeter error.

8 Parameter estimation

The algorithm has a number of parameters, all of which must be determined in some way. There are two types of parameter: general and specific. The general parameters may be determined through trial and error and set for any type of picture that conforms to our restrictions (or the restrictions set by the algorithm as its "confident area" (section 9)). The specific parameters tend to have different optimal values depending on the image itself. These must be determined automatically during processing.

8.1 General parameters

Table 1 lists the general parameters and the values we used for all of our experiments.

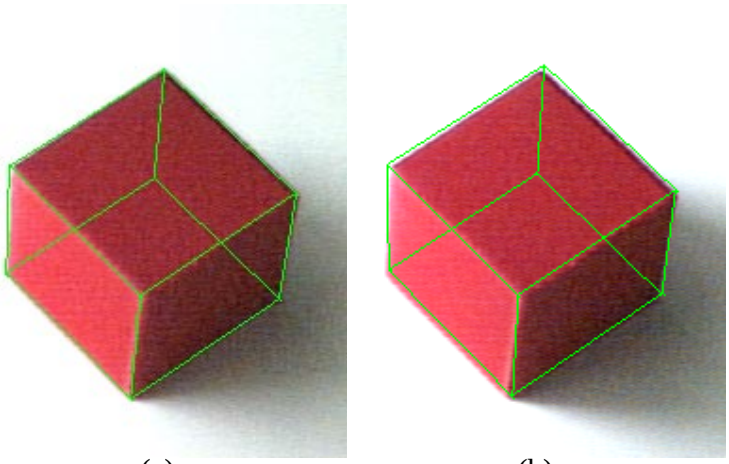


Figure 20: Two examples of shadowy conditions. (a) correctness value 1.5, (b) correctness value 3.0.

8.2 Specific parameters

The specific parameters are those that are sensitive to features of images, such as light intensity, color and blob size. The surface parameters, τ_{ocolor} and τ_{bg} actually only took on a few different values, as shown in Table 2.

The values of these parameters can be set appropriately on determining general intensity and color properties of a blob. The parameter N_{sample} determines most significantly the speed of the basic edge detection processing. The more scan lines that are used the slower the processing. Furthermore, we discovered that if too many scan lines are used the procedure is less sensitive to gradient changes and thus is less able to separate edge characterized by large corner angles (see section 4.2). The parameter must also not be too large, or there will be too few edge points to form lines. We set as minimum value $N_{sample} = N_{sample\_min} = 3$, and maximum value $N_{sample} = N_{sample\_max} = 7$. For each blob the maximum value is used first for all scan lines, and if too few (less than 4) vertices are attained the process is repeated after setting $N_{sample} = N_{sample} - 1$, and so on until enough vertices (at least 4) are found, or N_{sample_min} is reached. This method is not optimal, but allows good (clear or big) images to be processed quicker, while difficult images receive more concentrated attention.

9 Algorithm limitations and the concept of reflective teams

The algorithm described in this paper can be seen as one single global process of generating a 3D hypothesis from two camera images of a cuboid. On the other hand it is also clear that it is naturally decomposable into the processing levels described in the various sections, each level having its own set of parameters, extension possibilities and input/output

Symbol	Function	Value
ΔG	Grid size for blob extraction	30 pixels
τ_{blob}	Threshold angle of blob color with a cube color	10 °
$\tau_{surface}$	Defines a surface	4 °
D	Minimal pixel extent of a surface	4 pixels
N_{min}	Minimum number of edge points per edge line	5
$corr_{min}$	Minimum overall line correlation	0.8
s_{trans_start}	Initial step size for cube iteration (translational part)	20
s_{trans_stop}	Final step size for cube iteration (translational part)	0.5
s_{rot_start}	Initial step size for cube iteration (rotational part)	20
s_{rot_stop}	Final step size for cube iteration (rotational part)	0.2
fit_{good}	Lower bound on cube iteration error	0.5

Table 1: The general parameters used in all experiments

Parameter	Special case	Value
τ_{ocolor}	red, light blue, dark blue cubes; good lighting	20 °
τ_{bg}	red, light blue cubes, dark blue; good lighting	10 °
τ_{ocolor}	red, light blue, dark blue cubes; bad lighting	10 °
τ_{bg}	red, light blue, dark blue cubes; bad lighting	5 °
τ_{ocolor}	yellow/green cubes: all lighting	10 °
τ_{bg}	yellow/green cubes; all lighting	5 °

Table 2: Values of specific parameters τ_{ocolor} and τ_{bg}

characteristics. In short, each level of this algorithm can be seen as an *independent component operating on a particular set of input values to produce a set of output values, using certain assumptions, parameters, and methods, and which has limitations and accuracy that depend on the input data.*

The recognition of this fact, which applies to many algorithms, and is very often obvious through the algorithm's hierarchical form, is the first step to converting an algorithm into a building block for a larger system. Indeed we go further with this extension and convert it into a *reflective actor in a reflective team architecture.*

9.1 Reflective team architecture

More details about the reflective team architecture are given in [1, 5] and thus here we will only give a brief description. A reflective team can be considered as a unit consisting of a number of reflective actors[2] and acts itself as a reflective actor. The purpose of a team is to solve a problem at any given level by explicitly modelling the *problem decomposition* and its later *recombination.* Decomposition is typified by two major processes: *sub-task*

[2]Previously we used the word "agent" here, but due to overuse in and possible confusion with other areas of AI we adopted the word "actor".

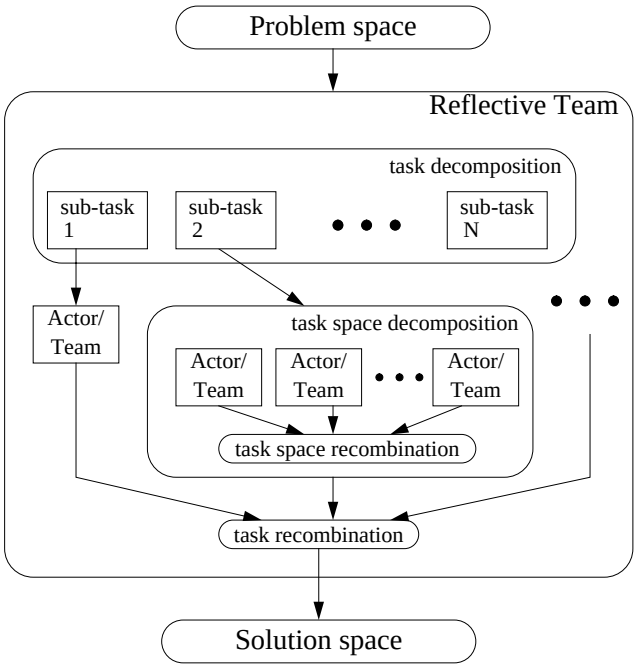


Figure 21: Illustration of a reflective team architecture. A problem is decomposed into sub-tasks, each of which is handled by an actor or a team. A team may itself perform task decomposition, a task space decomposition or both. All actors are reflective and may also be teams. Decomposition must be complemented by a recombination stage.

decomposition and *task space decomposition*. The first performs a problem-specific decomposition into sub-tasks which are then delegated to separate actors, and in the second the problem space of a given sub-task is divided into (perhaps overlapping) areas that are handled by different actors with each one performing the same task (see also [4]). Figure 21 illustrates these mechanisms.

The reflectiveness of each component of a team is important for the later recombination of the resulting sub-solutions and suggestions produced into a complete solution. Section 9.3 discusses this issue.

Hierarchical and iterative task decomposition may also be incorporated into the architecture. In order to represent this we extend the sub-task decomposition to include a shared-memory module (Figure 22). The actors that take part in the shared memory data exchange must also be able to handle time-dependent and temporarily missing data. Such a decomposition is in fact purely a parallel representation of the serial hierarchy using an implicit time iteration to hide the sequential processing. This means that the "recombination" module actually does not require any form of reflection in order to decide what to output as team result. The answer is clear: the most recent data emerging from the final stage of the hierarchy (actor N in this case) otherwise nothing. In this respect such a team representation does not automatically constitute a reflective team.

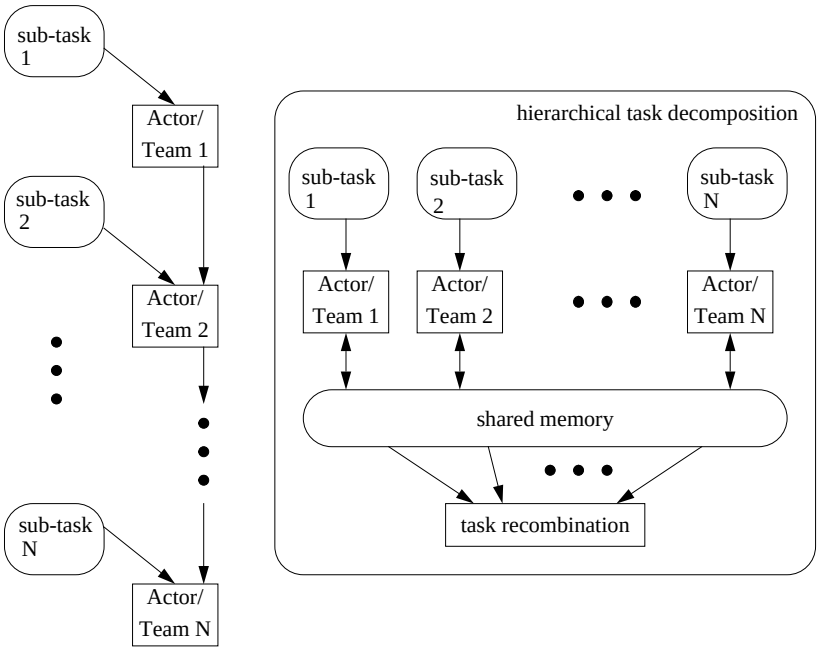


Figure 22: Incorporation of a serial hierarchical task decomposition into the reflective team architecture, through use of a shared memory module

9.2 3D cube algorithm as a non-reflective team

The algorithm described in this paper is task decomposed as illustrated in Figure 23. As can be seen, the inherently serial hierarchy is represented in a parallel architecture consisting of one main *cube hypothesis* team with a shared memory module. This team consists of two further *2D points* teams for left and right images, and two actors that deal with 3D point synthesis and cube fitting respectively. The 2D points teams also have a shared memory module and actors dealing with various aspects of the edge detection process described above.

9.3 Introducing reflection

As mentioned in section 9.1 no reflection is necessary to perform the recombination in Figure 23. However each component of the system can itself be reflective, in two different ways.

1. **Output data reflection.** An actor can reflect about the correctness or validity of the data it is writing as output. For example the vertices generation actor can output for each vertex in the set a corresponding estimate of the error associated with it (see section 6.2).

2. **Input data reflection.** An actor can process data that also have reflection values associated. These reflection values can be used to influence the weight or validity of each data value in the actor processing. They may also be used to provide further

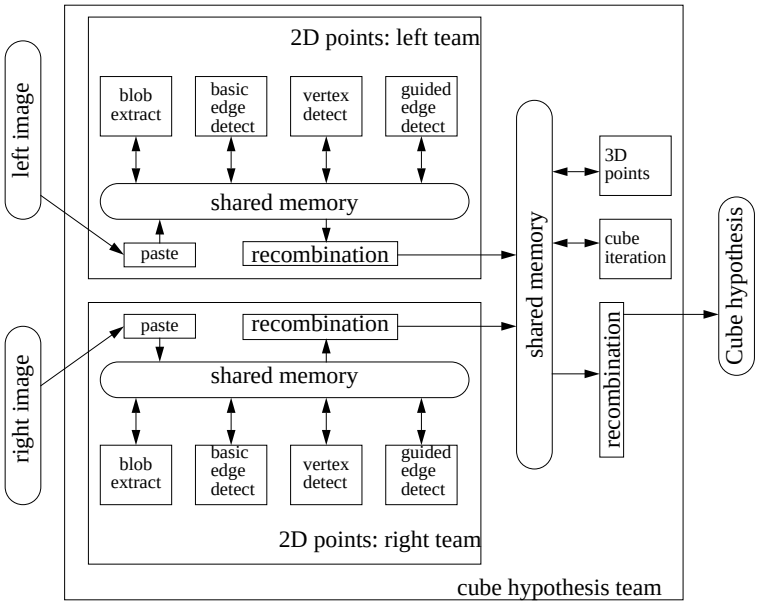


Figure 23: Actor/team task decomposition of the cube hypothesis algorithm

estimates for output data reflection. For example the vertices actor receives data about the edge lines, and these may each have reflection values about the accuracy of gradient or offset. Intersection jiggling using this information provides further estimates of intersection confidence.

But that is not all. Reflection values may also *actively* be used in order to *improve data generation*. An example: The 3D points actor has generated one point that has an unusually high error. All other points have acceptable error values. This information could be used to place more importance on the better points during the following cube iteration. It could however also be used to trace the error source. Depending on the way in which such information can be processed one could envisage the following:

- The 2D points used to generate the 3D points are jiggled within their error ranges to produce better 3D synthesis. The vertices actors for the 2D point with greater error is informed.
- The vertices actor informed locates the edge line with greater error used in its generation and demands analysis by the edge lines actor.
- The edge lines actor attempts to improve the line generation on the basis of the edge points it has. If this is not possible the edge points actor is informed.
- The edge points actor checks the edge points concerned and either corrects them or tries to improve generation of edge points for the given image area

This process only involved actors concerning themselves with things they knew about and were able to control. There was a further reverse sense of the processing (in the original

hierarchy from bottom to top) which is hardly noticeable in the shared memory team representation. Furthermore there emerges the exciting possibility of actively *improving* the processing of any given actor, either by internal parameter adjustment or by situation learning. In our example the edge points actor could learn when to use a particular set of $\{\tau_{ocolor}, \tau_{bg}\}$ through feedback from other actors and teams.

9.4 Self-assessment

There is one further aspect of reflection that needs to be mentioned. Although each specific actor/team may provide information on the accuracy of any particular output it emits, it is also useful for each actor to generate a *confidence value*. Confidence is to be understood as a general expression of satisfaction[3] by the actor that the generate output is a "good answer" for the current input. Such general information is most useful for the team recombination of information from actors that handle the same input and output data. It is also the basis of recombination of data from teams that divide up the task space, as in the Pandemonium team [4]. Thus there may be expert actors that really only know about the special case of blue cubes on a green background, and can handle these very well. Such an actor should have low confidence when faced with a red cube on a white background, and if another actor is present that can handle such cases it should be used instead.

In this way a multitude of different processing methods and more or less constrained expertness may be combined into a large hetereogenous system.

10 Conclusions

This paper was mainly concerned with the description of a fast 3D cube hypothesis algorithm for use with a robot manipulation system. In order to speed up the processing we made a number of assumptions about the nature of objects in the robot's world. Using simple heuristic methods we were able to construct a flexible and robust algorithm that generated typical cube sizes at a rate of 5 per second. This is a nice rate (although still too slow) for a real-time robot manipulator. The approximations we made along the way were characterized by a number of parameters, most of which could be given fixed values. Some parameters were however dependent on the input data (images), and so special cases had to be introduced.

The paper was also concerned with outlining how such an algorithm could be embedded into a reflective team architecture. This was demonstrated not only to be a useful structural simplification but also one that allowed the insertion of (a) more intelligent forms of processing the data in a complex hierarchy, and (b) extension to a large hetereogenous system that may be improved to handle wider data variations and, notably, less explicit assumptions.

[3] Excuse the anthropomorphism!

Acknowledgements

The author would like to thank Gernot Richter for long, fruitful and patient reading and re-reading of the manuscript, and for many useful comments and suggestions.

References

[1] U. Beyer and F. J. Śmieja. Learning from examples, agent teams and the concept of reflection. *International Journal of Pattern Recognition and Artificial Intelligence*, 10(3):251–272, 1996.

[2] F. Dornaika and C. Garcia. Robust camera calibration using 2d to 3d feature correspondences. In *Proceedings of the International Symposium SPIE – Optical Science Engineering and Instrumentation, Videometrics V, Volume 3174*, pages 123–133, San Diego, Ca., July 1997.

[3] Christophe Garcia. Complete calibration of the autonomous hand-eye robot janus. internal report, GMD - German National Research Center for Information Technology, October 1997.

[4] F. J. Śmieja. The Pandemonium system of reflective agents. *IEEE Transactions on Neural Networks*, 7(1):97–106, 1996.

[5] F. J. Śmieja and H. Mühlenbein. Reflective modular neural network systems. Technical Report 633, GMD - German National Research Center for Information Technology, Sankt Augustin, Germany, February 1992.