Journal of Universal Computer Science, vol. 15, no. 14 (2009), 2765-2785
submitted: 28/6/09, accepted: 29/7/09, appeared: 1/8/09 © J.UCS

Realtime LEGO Brick Image Retrieval with Cellular Automata

Leendert Botha
(Stellenbosch University, South Africa
lbotha@cs.sun.ac.za)

Lynette van Zijl
(Stellenbosch University, South Africa
lvzijl@sun.ac.za)

McElory Hoffmann
(Stellenbosch University, South Africa
mcelory@cs.sun.ac.za)

Abstract: We consider the realtime content-based image retrieval of LEGO bricks from a database of images of LEGO bricks. This seemingly simple problem contains a number of surprisingly the image signature, and corresponding feature set, and illustrate cellular automaton-based methods for the whole feature extraction phase.

Key Words: Content-based image retrieval

Category: H.2.8 Image databases I.5 Pattern recognition

1 Introduction

Content-based image retrieval (CBIR) has received renewed attention over the past decade [Datta et al. 2008]. Real-world image retrieval systems for specific domains have been investigated for diverse domains, such as photo album management, medical imaging, astronomy, and many others. Datta *et al* point out the importance of such studies [Datta et al. 2008], and remarks that specific techniques can often be generalized to other domains.

In this work, we consider the domain of images of LEGO bricks. We implemented a content-based image retrieval (CBIR) system where a user submits an image of a given LEGO brick, and the set of closest-matching bricks is retrieved in realtime from a database of images of LEGO bricks. The practical application here is for a user to identify an unknown LEGO piece and obtain metadata about it.

This seemingly frivolous problem poses some interesting challenges. For example, the images in the database must be identified by means of form rather than primarily by color, as the same type of brick can have many different colors. As the form of a brick is a three-dimensional feature, this requirement increases the difficulty of the problem. Moreover, the images in the database and the user-supplied input image can have different sizes, colors, perspectives and lighting conditions. For example, in figure 1, the brick on the left-hand side is a so-called 2x4 brick, and is semantically the same as the brick on the right-hand side. The next challenge in this problem is the fact that realtime recognition is required, which means that the matching process must be fast.

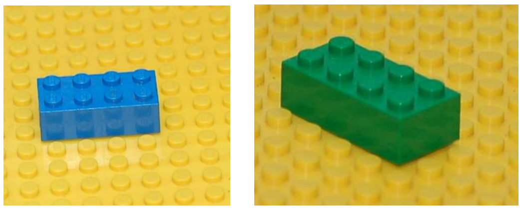

Figure 1: Two semantically equivalent LEGO bricks.

The focus of our work was to find a 'good enough' solution that would accurately recognize semantically equivalent LEGO bricks in realtime. We designed a novel solution to the realtime LEGO brick matching problem, where the whole feature extraction phase is solved with the use of cellular automata, while the feature matching phase is solved mostly with known CBIR algorithms. We note that this allows us to execute the feature extraction in total on a GPU, or alternatively, at least in parallel on several CPUs.

We describe our solution in section 2, and then analyse our results in section 3. We conclude in section 4.

2 The LEGO domain CBIR system

The major components of our LEGO CBIR system are modules for edge detection, for feature extraction, for database operations and queries, and for the user interface and results viewer.

The edge detection module takes as input the original image, and then performs background elimination and edge detection on the image. The results are returned as a binary image (see figure 6) with the non-zero pixels corresponding to the edges of the LEGO brick.

The feature extraction module extracts the color, shape and stud formation of the LEGO brick. It takes as input the background eliminated image (see figure 5) and the binary edge detected image (see figure 6) and gives as output three histograms containing the color information of the brick, two integers describing the stud formation, and a feature vector encoding the shape of the LEGO brick.

The database module performs all database transactions, such as storing images to and retrieving images from the database. Every search query effectively browses through existing images, eliminating and sorting them according to the specified search criteria.

2.1 Edge detection

The edge detection algorithm for the LEGO brick problem needs to isolate the inside and outside edges of the brick, as well as the pattern of studs on the top of the brick. The need for realtime analysis of the image effectively eliminated general edge detectors such as the SUSAN edge detector [Smith and Brady 1997] and the Canny edge detector [Canny 1986]. Instead, we investigated other simpler and less accurate edge detectors which could provide realtime analysis. This included the Sobel operator [Ballard and Brown 1981], the Roberts cross operator [Roberts 1965], the Prewitt operator [Prewitt 1994] and the Marr-Hildreth algorithm [Marr and Hildreth 1980]. Each of these had its own problems for this specific domain. For example, the Marr-Hildreth algorithm has problems with the localization of curved edges, which occurs frequently in our data set.

Our final solution implements a method proposed independently by Popovici *et al* [Popovici and Popovici 2002] and Chang *et al* [Chan et al. 2004], based on cellular automata (CA). A CA is a k-dimensional array of cells. Given a cell in a CA, one considers its neighbouring cells to decide its next state. All cells are evaluated simultaneously (in parallel). In our case, the CA is 2-dimensional, and each cell represents one color pixel in the image plane.

Following [Popovici and Popovici 2002], we define the next state of the CA based on a transition function $\delta : S^5 \to S$, with

$$\delta(s_1, s_2, s, s_3, s_4) = \begin{cases} 0, & \text{if } \varphi(s, s_i) < \epsilon, i = 1, \ldots, 4 \\ s, & \text{otherwise} \end{cases} \quad .$$

Here, the states s_i represent the Von Neumann neighbourhood of the current cell s (see figure 2). Also, $\varphi(s, s_i)$ is a similarity measure between the pixels s and s_i which decreases as the similarity increases, so that $\varphi(s, s) = 0$. The simplest method is to define φ as the Euclidean distance in RGB-space, thus $\varphi(s, s_i) = \|s - s_i\|$.

It is important to note here that CA typically evolve over several time steps to a stable configuration. In Popovici *et al*'s method, the CA performs one time step only, which makes the use of a CA apparently superfluous. However, the important point is that the evaluation of the new state of each pixel is done in parallel as with all CA, and this parallel evaluation gives results superior to a standard sequential traversal of the 2D array of pixels.

The CA was implemented so as to use either the vector angle or the Euclidean measure as a distance measure, and it also offers the option of performing the

	s_1	
s_2	s	s_3
	s_4	

Figure 2: The Von Neumann neighbourhood of cell s in a CA.

edge detection in either the YIQ or RGB color space [1]. Sample output is shown in figure 3.

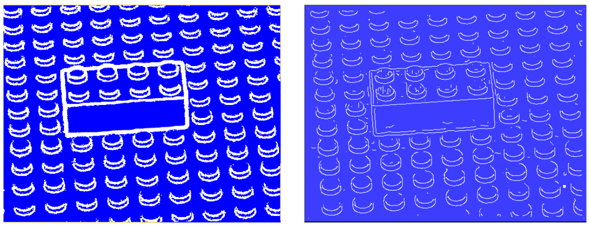

Figure 3: The edge detected images using cellular automata (left) and the Canny edge detector (right).

2.2 Background elimination

To be able to calculate accurate feature vectors for the image of the LEGO brick, all of the pixels that correspond to the background need to be eliminated. The problem of identifying the background is quite specific to this project, and we designed custom algorithms to solve the problem. We again made use of CA.

2.2.1 Elimination based on color

All the images in the database are composed of objects photographed on a background of a LEGO baseboard. The color of the baseboard is generally distinguishable from the color of the object. However, the studs on the baseboard

[1] Using the YIQ color space can increase search accuracy[Patel and Tonkelowitz 2003], since the histogram distance measures in this color space corresponds more to human perception.

form shaded areas which are much darker and are hence harder to recognise. The first part of the background elimination process identifies the color of the baseboard by taking the colors at the edge of the image (see figure 4). All pixels which have approximately the same color are then removed. After this process, the only background pixels that remain are the pixels that correspond to the shaded regions under the studs. To eliminate the shaded regions, we used two CA (described below). An example of the result from this phase is shown in figure 4.

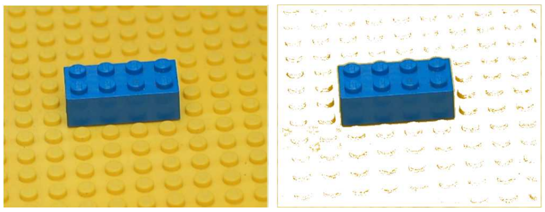

Figure 4: The original image on the left and the image after removing background pixels, based on their color. The small border around the picture indicates which pixels were used to determine the background color.

The first CA effectively eliminates pixels which are surrounded on all four sides by background pixels. This CA uses a neighborhood distance of δ_L in all four directions of the grid. If background pixels are encountered within the δ_L distance in all of the four directions, then the objective pixel is identified to be a background pixel. After a few iterations of this CA, most of the background is eliminated, with the exception of straight lines that could occur in some lighting situations. Note that the transition rule of this CA cannot remove straight lines since no neighboring background pixels will be encountered in the direction of the line.

A second CA is used to 'break' the straight lines. This CA uses a smaller distance, δ_S, to look in all four directions, but this time, a pixel is identified as background if either the horizontal or the vertical directions contain background pixels within the distance δ_S.

Combining the above three steps yields the desired results as all of the background pixels are removed, except for those immediately adjacent to the object, as can be seen in Figure 5. This is no problem, however, since no false edges occur in that region and the desired result can be obtained by subtracting all of the identified background pixels from the edge-detected image. An example result is shown in figure 6.

The reader may note that there are indeed other algorithms to eliminate

Figure 5: The image after applying the two CA.

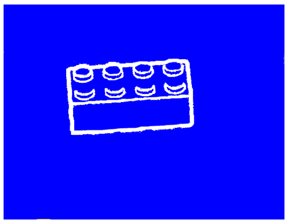

Figure 6: The final output after edge detection and background elimination.

background color. However, as noted earlier, we want to use only CA in the feature extraction phase for potential GPU implementation.

2.3 Feature extraction

A LEGO brick is semantically defined by its form, studs and to a lesser extent its color. Both the number of studs and the arrangement of the studs is significant. For example, in figure 13, the top left brick is a rectangular 2 by 4 brick. It has eight studs, arranged in a pattern of two rows of four studs each in straight lines. Other brick forms include rounded bricks (top right), 'macaroni' bricks (bottom right), and L-shaped bricks (bottom row, third from left). Note that the 2×2 brick (top row, second from left) has the same number of studs as the 1×4 brick (bottom row, second from left), but these two bricks are clearly semantically different.

The feature vector describing a LEGO brick therefore has to mathematically describe its form, number of studs, stud arrangement, and color. Below, we describe how we extract each of these different features into a single feature vector for any given brick.

2.3.1 Stud location

The reader may notice that the edges of a stud has a distinctive shape due to the projection of the images, and hence standard shape detection methods (such as ellipse-detection) cannot be used here. Finding such non-standard shapes within an image is typically done using template matching.

A template is a subimage which is itself a picture. Template matching is the process of moving this template around the image until a location is found which maximizes some match function. A popular match function is the squared error [Snyder and Qi 2003]:

$$\mathrm{SE}(x,y) = \sum_{\alpha=1}^{N}\sum_{\beta=1}^{N}\left(I(x-\alpha, y-\beta) - T(\alpha,\beta)\right)^2$$

where I is the image and T is the $N \times N$ template.

Since we are (possibly) looking for more than one stud, we will be searching for local maxima of the match function. A major problem with template matching is that although the size of the template is constant, the size of the studs in the image depend on the scale of the image. We therefore use a set of templates, as a dynamic resizing of the template to fit the scale of the image is time intensive and an inaccurate resizing of the template could cause the algorithm to fail.

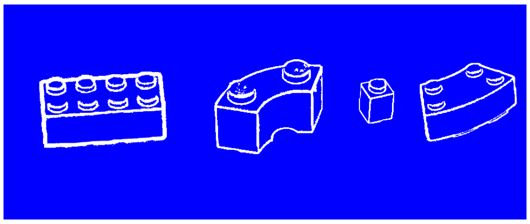

Figure 7: Four edge detected bricks showing similarity in the shape of the studs.

The first requirement for the template matching algorithm is to create a template. Figure 7 shows four edge detected LEGO bricks. Notice the similarity in the shape of the studs, and also the invariance to slight rotation. Based on these similarities, we finally developed the set of templates in figure 8.

Figure 8: The set of templates used in the template matching process.

Since the sizes of the individual templates differ, the score of the matching function is scaled relative to the size of the template, in order to prevent large templates from accumulating large scores in images of small scale. In order to decrease the runtime, the centre point of the template is chosen in a way that allows us to calculate the match function only for pixels that lie on edges, without affecting the accuracy of the procedure. This reduces the runtime by up to a factor of 20. This approach is valid, since we know that any pixel on the template that is surrounded by a large number of white pixels (edges) is highly likely to correspond to a white pixel (edges) on the image, when a stud is found.

The template matching process employs the following steps:

1. For each of the 12 templates:
   - Place the template with its centre on each of the pixels corresponding to the edges.
   - Calculate the value of the match function (squared error) at every one of these locations.
   - Identify the highest score (most likely stud location).
2. Pick the template with the highest high score. This template will be used to identify the studs.
3. Re-calculate the match values for the chosen template.
4. All of the pixels with a match value smaller than the threshold, are identified as locations for studs.
5. Examine the chosen locations and eliminate multiple detections of the same stud (two locations close to each other).

Notice that in step 3 above, the match is re-calculated. The alternative is to store all of the 12 calculated match values for each pixel, but this could require excessive memory for large images.

2.3.2 Stud formation

Given the locations of the centre points of all of the studs, the objective is to determine in what formation they lie. Recall that we assume that all of the studs lie on a grid, and the formation refers to the number of rows and the number of columns in this grid. A brick with a total of eight studs divided into two rows of four studs each, has a formation of 2×4.

This problem reduces to finding the set of straight lines $\mathcal{L} = \{l_1, l_2, \ldots, l_n\}$ of minimum cardinality such that each stud lies on exactly one l_i. Note that, since the exact positions of the studs cannot be determined, a stud is judged to "lie on" a line if it is within a certain perpendicular distance from the line. This problem has been shown to be APX-hard [Kumar et al. 2000], which means that there exists polynomial-time approximation algorithms with approximation ratio bounded by a constant. However, the number of studs is always small (typically less than 24), so the performance gain of using an approximation algorithm over an exact approach is not large enough to warrant the possible loss in accuracy.

Our algorithm starts off by choosing all possible combinations of two studs and constructing lines between them. Assuming there are n studs, this step constructs $\frac{n(n-1)}{2}$ lines and is thus $O(n^2)$. For each line, the number of studs that lie on the line is calculated. Since there are $O(n^2)$ lines, and for each line $O(n)$ calculations are made, this has a complexity of $O(n^3)$. A greedy approach is then used, repeatedly removing the line which covers the most studs and eliminating the studs it covers, until no studs are left.

The algorithm for determining the stud location is presented below (see Algorithm 1). Note that the output of the algorithm is the number of studs and the number of lines needed to cover them. These two values are stored in the database.

Given the stud locations and the stud arrangement, we need to find the form of the brick. This is a three-dimensional problem, as the form is defined not only by the outlying edges, but also the inside edges of the brick. Since we do not have a calibrated setup, a three-dimensional matching based on standard methods would be prohibitively expensive from a computational point of view. Our solution was to simplify the problem to a two-dimensional problem, based on the symmetry of the LEGO brick shapes. That is, all the standard LEGO bricks are protruded shapes of the top surface of the brick. Therefore, if it is possible to identify the top surface of the brick, the problem reduces to a two-dimensional problem.

2.3.3 Identifying the top surface

The top surface is identified by flooding in all directions, starting at the stud locations. Again, we implement this as a CA, which in essence floods from stud locations to the nearest edge. Since the output from this phase will be encoded into a feature vector, the flooded pixels themselves are not given as output but rather the pixels that directly surround the flooded area.

Before the flooding can start, the edges that correspond to the studs need to be removed. This is done by removing all edges that are covered by the template at the identified locations. This typically leaves random noise on the top surface, which is eliminated using a CA similar to the one used to eliminate the background. An example of this whole process is shown in figure 9.

The final information needed to construct the feature vector of a given brick, is its color information. We simply used the standard way of summarizing color

Algorithm 1 Determining the formation of the studs

```
procedure GET_FORMATION(Set of studs 𝒮 )
    ℒ ← ∅                                        ▷ Initialize the set holding the lines
    for i ← 1 to size(𝒮) do                      ▷ Add all possible lines
        for j ← i + 1 to size(𝒮) do
            ℒ ← ℒ + new Line{𝒮.get(i), 𝒮.get(j)}
        end for
    end for

    for each line l in ℒ do ▷ Determine how many studs covered by each line
        for each stud s in 𝒮 do
            if distance(l, s) < δ then           ▷ If s lies very close to l
                l.coveredStuds.add(s)            ▷ Add s to set covered by l
            end if
        end for
    end for

    count ← 0                                    ▷ The cardinality of the covering set
    while size(𝒮) > 0 do
        l ← ℒ.removeMax()                        ▷ Remove line that covers most studs
        𝒮 ← 𝒮 − l.coveredStuds
        count+ = 1
    end while

    return count, size(𝒮)                        ▷ The formation is count by size(𝒮)/count
end procedure
```

values, namely, color historgrams [Siggelkow 2002, Deselaers et al. 2004]. To create a color histogram, one constructs a separate 256-bin histogram for every color channel [Kotoulas and Andreadis 2003]. We note that, in the LEGO brick application, the color histograms are calculated on an object which has mostly uniform color as compared to a whole image containing a wide variety of objects and colors. For this reason, the RGB-color space may offer accurate enough retrieval and avoid the conversions between color spaces. We leave the choice of color space (RGB vs YIQ) as an option to the user. Note that we normalize the histograms, so that every bin indicates the percentage of values that fall in that region, rather than the total number of values that fall in that region. The normalized color histograms, in the RGB space, for the LEGO brick of figure 6 is shown in figure 10.

We are now finally able to encode the feature vector of a given LEGO brick,

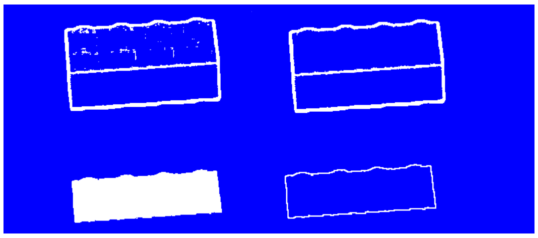

Figure 9: The output of the flooding process for the brick of figure 6. The top left picture shows the edges after the studs have been removed. A CA is applied to remove the noise and the output is shown in the top right picture. The flooded region is shown on the bottom left and the boundary of this region, which is to be encoded into a feature vector, is shown on the bottom right.

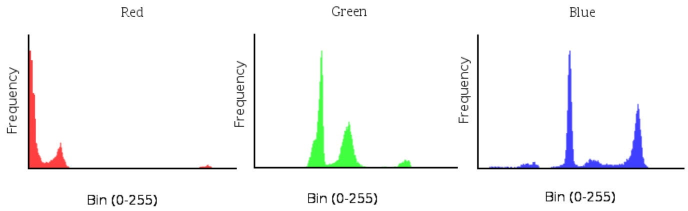

Figure 10: The normalized color histograms, in the RGB space, for the LEGO brick of figure 6.

based on its number of studs, stud locations, form, and color.

There are standards methods for encoding shape feature vectors, such as chain codes, ψ-s curves, Fourier descriptors, and geometrical or spatial moments [Davies 2004]. In our case, we settled on using the Hu-set of invariant moments [Hu 1962], as it is computationally inexpensive, and invariant to changes in rotation, scale and translation.

2.4 Feature matching

An interesting design choice made for the feature matching was the handling of multiple search criteria. Searching for studs with the same stud formation is straightforward since the database can clearly be divided into two mutually exclusive groups: those that have the exact same stud formation, and those that do not. However, for the color and shape comparison, the similarity is given by a continuous match score (we implemented both bin-to-bin and cross-bin quadratic color comparison algorithms). A threshold can be applied to this match score, to determine which images are “close enough” to the query image.

A relatively accurate threshold can be determined for the color matching, but for the shape matching the match score can assume a rather wide range of values, and thresholding may eliminate shapes that are similar. To achieve the best possible search results, the matching process is handled as follows:

- If color is the only search criterion, then the entries in the databases is sorted according to the color match, and the n best entries are displayed.
- If shape is the only search criterion, then the entries in the databases is sorted according to the shape match, and the n best entries are displayed.
- If both color and shape are search criteria, then thresholding is applied on the color match score, eliminating the entries with a significant color difference from the query image. The remaining entries are sorted according to the shape match score and the n best entries are displayed.
- If stud formation is one of the search criteria, the entries without the required stud formation is eliminated and the remaining entries are processed as above.

3 Analysis

The analysis of our system served to test each of our design decisions, and compares different approaches to achieve the best result for this specific CBIR application.

3.1 Edge detection

Recall that we used Euclidean distance in the similarity measure for our CA. As vector angle [Davies 2004] is another standard measure, we implemented both and compared the results (see figure 11). The Euclidean distance was computationally faster, and resulted in less noise in the top surface of the bricks, where correct stud location is critical in later phases.

The next point in edge detection was to compare other standard edge detectors to Popovici's CA approach. When using the optimal parameters, both the Canny edge detector and the CA edge detector deliver good results. However, we found the Canny edge detector to be more sensitive to noise than the CA edge detector (see again figure 3). The edges generated by the Canny edge detector are much thinner than those generated by the CA. This can cause the template matching phase to fail, as it is difficult to define a template that overlap with the stud shape in all cases. The CA edge detector also requires substantially less processing. Figure 12 shows the runtimes of the two methods for various image sizes. By default, the system now uses the CA edge detector.

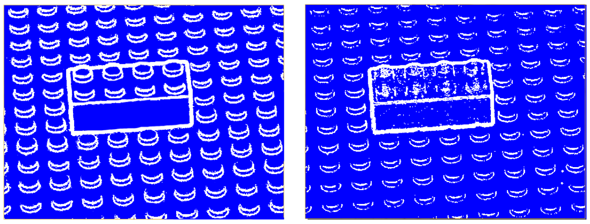

Figure 11: Edge detection using Euclidean distance (left) and vector angle (right) as similarity measures.

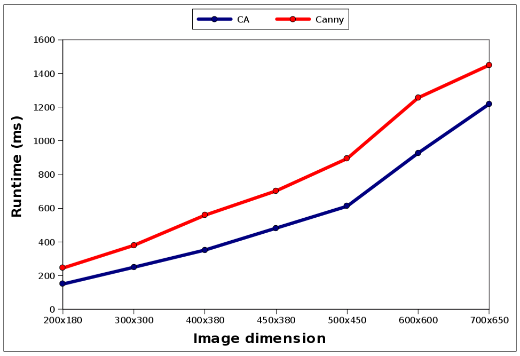

Figure 12: A graph showing the runtimes of the two edge detection methods, for various images sizes.

3.2 Background elimination

The background elimination phase is clearly successful, leaving only minor traces of the baseboard where it makes contact with the brick. It accurately removes the background and it does so in minimal time. Figure 13 shows more sample output from this phase.

3.3 Feature extraction

3.3.1 Color histogram

The successful construction of the color histogram only depends on the accuracy of the background elimination phase and as such, is deemed to be an accurate color descriptor. Test results for the retrieval based on color is given and discussed in section 3.4.1.

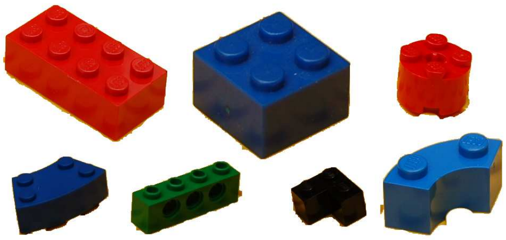

Figure 13: Sample output from the background elimination phase.

3.3.2 Stud formation

The success of this phase depends to a large extent on the quality and lighting conditions of the image. For high quality images with good uniform lighting, the studs are accurately located using the template matching approach. Reflections, noise and blur in the images can influence the process and may result in falsely detected studs or studs that go undetected. Figure 14 shows a blurry image and its edge detected variant, with the location of the detected studs marked with red crosses. The template that was used for the matching is shown on the right hand side. In this case, the failure is due to the blur in the original image, which causes false edges to be detected. The two studs that went undetected were particularly influenced by these false edges, to such an extent that the template did not match.

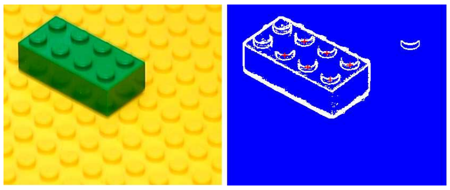

Figure 14: A blurry image for which the stud detection process failed. The template used in the process is shown on the right.

The reader may note that the stud detection process is crucial in the subsequent correct identification of matching bricks. Our experiments show that our stud detection is correct in more than 80% of cases, even for such difficult situations as depicted in figure 15.

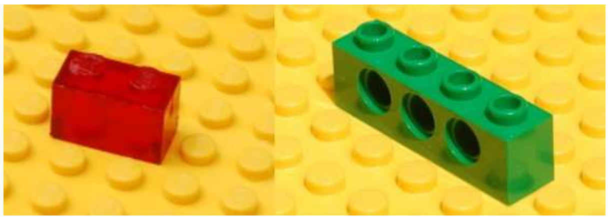

Figure 15: Difficult studs that are correctly detected.

3.3.3 Shape descriptor

The extraction of the shape feature vector depends on the accurate location of the studs. As described in the previous section, the results of this phase is in general satisfactory. The only other requirement for the successful completion of this phase is that the boundary of the top surface must be continuous on the sides where it is not in direct contact with the baseboard. Recall that the flooding process stops when an edge or a background pixel is encountered. Figure 16 shows an example of a brick which does not have a continuous edge separating the top surface from the rest of the brick. The result is that the whole brick is flooded, resulting in an inaccurate feature vector.

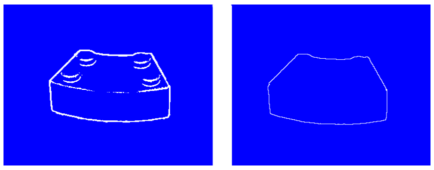

Figure 16: An illustration of how the shape extraction process fails for a brick that does not have a continuous edge separating the top surface from the rest of the brick.

This problem is solved by simply adjusting the threshold of the edge detector, which is a user adjustable option in the software. Figure 17 shows the same brick, but with a better threshold used in the edge detection process.

3.4 Color-based retrieval

3.4.1 Bin-to-bin matching versus cross-bin matching

In general, the color matching was found to be accurate. The best results are obtained using cross-bin matching, but this comes at a cost (see the runtime

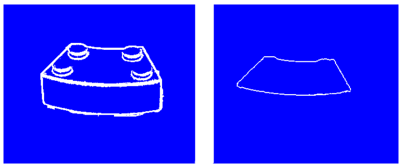

Figure 17: The same brick from figure 16, using a better threshold, and resulting in a perfect identification of the top surface.

analysis in section 3.6.2). To illustrate the difference in accuracy between the bin-to-bin approach and the cross-bin-approach, a search query was performed on a small set of images containing two red bricks, two green bricks and one blue brick. A semi-transparent red brick was used as the query image. The search results for bin-to-bin matching and cross-bin matching is shown in figures 18 and 19 respectively. From these figures, two major advantages of cross-bin matching can be identified. Firstly, the good matches (according to human perspective) generally tend to have lower color difference than the bad matches. Notice the huge difference between the second and third bar in figure 19, whereas for bin-to-bin matching the values are much closer, making it difficult to identify a suitable threshold to separate the 'good' matches from the 'bad' matches. Secondly,

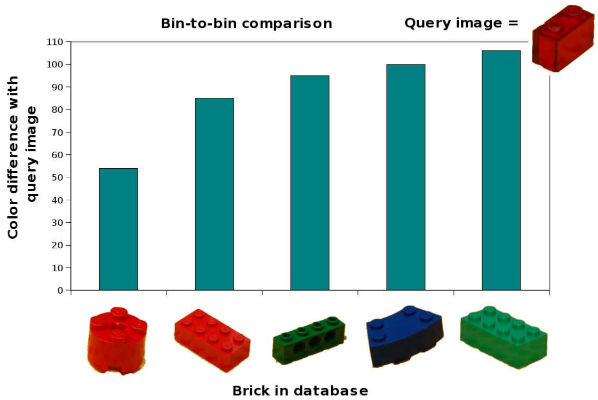

Figure 18: The color differences between a query image (top right) and five of the images in the database. The values indicate the Euclidean distance measure between the two histograms.

notice the order of the search results. The last three search results in the case of bin-to-bin matching are $green, blue, green$ which is not quite intuitive, since

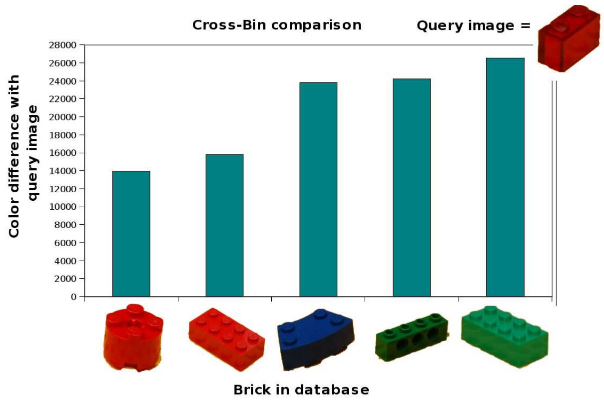

Figure 19: The color differences between a query image (top right) and five of the images in the database. The cross-bin distance measure is calculated using the quadratic form.

human perspective would rate the two green bricks as equally close to a red brick. No human would rate the one green brick as a better match than the blue brick and the other green brick as a worse match than the blue brick.

3.4.2 RGB space versus YIQ space

The experiment in the previous section was done using the RGB color space. In general, the RGB space was found to be more accurate than the YIQ space. When performing the same search query in the YIQ space, the blue brick is identified as a better match than one of the red bricks, as is indicated in figure 20. As noted earlier, the color histograms in this application differ from standard applications, since they describe objects of approximately uniform color. This is, in our opinion, the reason why the RGB space outperforms the YIQ space in this specific application. In its final version, the system has been configured to use the RGB color space by default.

3.5 Shape-based retrieval

The shape-based retrieval is highly dependent on the quality of the shape features extracted. If the top surface was identified correctly, then the shape-based retrieval is accurate. An example is shown in figure 21, where a shape query was made in a database containing five different shapes. The results are in the order of best match, with the match score (lowest being best match) indicated underneath each brick. An identical brick with totally different orientation came up as

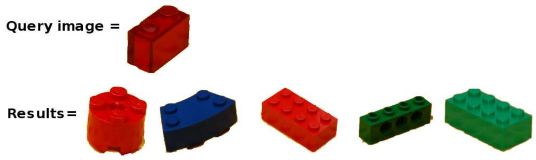

Figure 20: The inaccurate search results when searching by color using the YIQ color space.

the best match by far, followed by two more curved bricks with the rectangular bricks being the worst matches. The top surface that was used to construct the feature vectors for each brick is also shown.

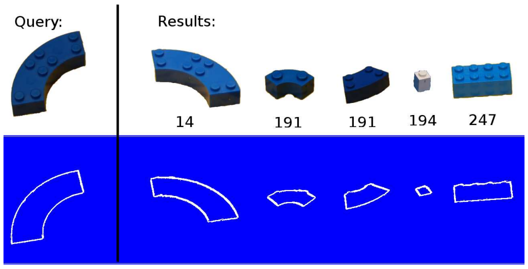

Figure 21: A sample shape retrieval query. Note that the match scores are presented in thousands. The match score for the best match is actually 14,000.

3.6 Runtime analysis

3.6.1 Runtime for each phase

Figure 22 shows a complete profile of the processing time required for every phase in our system. From the graph it is clear that no phase totally dominates the processing resources. Note the interesting memory/processing trade-off in the feature extraction phase: although minimal processing is required to construct the color histograms, they require approximately 100 times more storage space than the invariant shape features phase, which takes considerably longer to calculate.

3.6.2 Database query speed

Figure 23 shows the time required to query the database for both cross-bin and bin-to-bin color matching. Clearly, bin-to-bin matching is faster than cross-bin

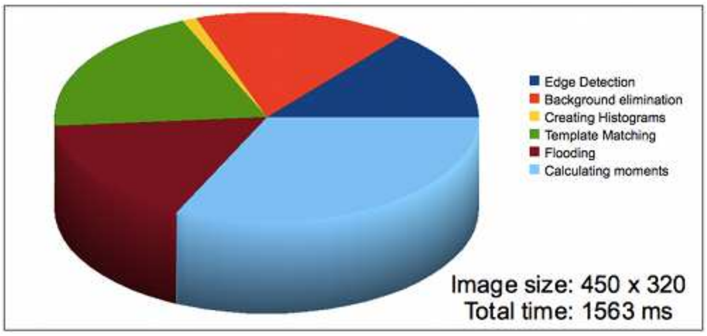

Figure 22: An analysis of the proportion of time spent within each phase.

comparison, as is to be expected.

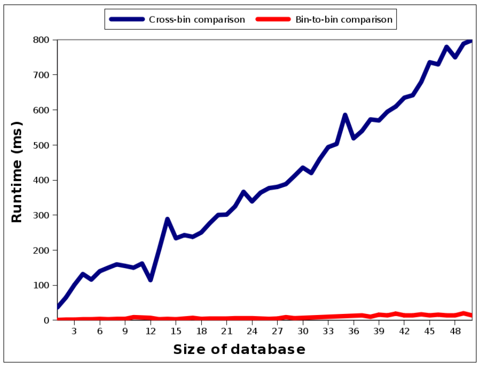

Figure 23: The amount of time needed to query the database using cross-bin and bin-to-bin comparison respectively.

3.7 Comparison with existing systems

A valuable measure of performance is comparison with existing systems. In this case, however, there are no directly comparable systems available. Most systems, such as the FIRE search engine [Deselaers et al. 2004], operate on a wide variety of images and focus on clustering the images into broad groups. Specifying an image of a LEGO brick as search criteria will in most cases return a wide variety of toys and, at best, a few LEGO bricks, regardless of whether their shapes/colors are similar to that of the query image.

4 Conclusion

We successfully implemented a CBIR system for the realtime retrieval of LEGO images, based on cellular automata. We compared different approaches, and in particular considered the use of CA. It was determined that the widely-used YIQ color space, does not perform well for this application. The expensive, optimal Canny edge detector was replaced by an inexpensive and simple CA edge detector, which gave a solid foundation on which the later stages were built successfully. It was also determined that the Euclidean distance measure is a sufficient similarity measure between pixels and using the more complex vector angle measure decreased the performance of the edge detector.

For the feature extraction, it was found that three 256-bin color histograms are sufficient to provide accurate retrieval based on color. The locations of the studs have been successfully determined using template matching with a set of 12 templates of various sizes. From this, the minimum number of lines that cover the studs are determined, since this reveals the formation in which the studs lie.

The feature vector to describe the shape of the LEGO brick was obtained by a semantic simplification from the 3D shape into a 2D shape. The optimal 'signature' for a LEGO brick was defined as the outline of the surface that contains the studs. This feature was calculated and encoded using the Hu set of invariant moments. The retrieval based on these invariant features are successful and the time needed to match the shape feature to the database is minimal.

We conclude that the use of cellular automata for feature extraction is possible and allows for realtime retrieval for this application. Future work could include the porting of the CA implementation to the GPU to measure performance improvements. In addition, other applications where feature extraction can be fully implemented with CA, could be investigated.

References

[Ballard and Brown 1981] Ballard, D. H., Brown, C. M.: Computer Vision; Prentice-Hall Inc., Cliffs,E., 1981.

[Canny 1986] Canny, J.: "A computational approach to edge detection"; IEEE Trans. Pattern Anal. Mach. Intell.; 8 (1986), 6, 679–698.

[Chan et al. 2004] Chan, C., Zhang, Y., Gdong, Y.: "Cellular automata for edge detection of images"; Proceedings of the Third International Conference on Machine Learning and Cybernetics; 3830–3834; 2004.

[Datta et al. 2008] Datta, R., Joshi, D., Li, J., Wang, J.: "Image retrieval: ideas, influences, and trends of the new age"; ACM Computing Surveys; 40 (2008), 2.

[Davies 2004] Davies, E.: Machine Vision: Theory, Algorithms, Practicalities; Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[Deselaers et al. 2004] Deselaers, T., Keysers, D., Ney, H.: "Features for image retrieval – a quantitative comparison"; In DAGM 2004, Pattern Recognition, 26th DAGM Symposium, number 3175 in LNCS; 228–236; 2004.

[Hu 1962] Hu, K.: "Visual pattern recognition by moment invariants"; IRE Transactions on Information Theory; IT-8 (1962), 179–187.

[Kotoulas and Andreadis 2003] Kotoulas, L., Andreadis, I.: "Colour histogram content-based image retrieval and hardware implementation"; Circuits, Devices and Systems, IEE Proceedings -; 150 (2003), 5, 387–93+.

[Kumar et al. 2000] Kumar, V., Arya, S., Ramesh, H.: "Hardness of set cover with intersection 1"; ICALP '00: Proceedings of the 27th International Colloquium on Automata, Languages and Programming; 624–635; Springer-Verlag, London, UK, 2000.

[Marr and Hildreth 1980] Marr, D., Hildreth, E.: "Theory of edge detection"; Proceedings of the Royal Society of London. Series B, Biological Sciences; 207 (1980), 1167, 187–217.

[Patel and Tonkelowitz 2003] Patel, A., Tonkelowitz, I.: "Whassupp: A novel approach to query-by-sketch using wavelet coefficients and color histograms"; Harvard University; (2003).

[Popovici and Popovici 2002] Popovici, A., Popovici, D.: "Cellular automata in image processing"; Proceedings of the 15th International Symposium on Mathematical Theory of Networks and Systems; University of Notre Dame, 2002.

[Prewitt 1994] Prewitt, J.: "Object enhancement and extraction"; Picture Processing and Psychopictorics; 3 (1994), 231–262.

[Roberts 1965] Roberts, L.: "Machine perception of three-dimensional solids"; Optical and Electro-optical Information Processing; (1965).

[Siggelkow 2002] Siggelkow, S.: Feature Histograms for Content-Based Image Retrieval; Ph.D. thesis; Albert-Ludwigs-Universität Freiburg, Fakultät für Angewandte Wissenschaften, Germany (2002).

[Smith and Brady 1997] Smith, S., Brady, J.: "Susan – a new approach to low level image processing"; International Journal of Computer Vision; 23 (1997), 45–78.

[Snyder and Qi 2003] Snyder, W., Qi, H.: Machine Vision; Cambridge University Press, New York, NY, USA, 2003.