Feature Extraction for Image Pattern Matching with Cellular Automata

Lynette van Zijl and Leendert Botha

Department of Computer Science
Stellenbosch University
South Africa
lvzijl@sun.ac.za, lbotha@cs.sun.ac.za

Abstract. It is shown that cellular automata can be used for feature extraction of images in image pattern matching systems. The problem under consideration is an image pattern matching problem of a single image against a database of LEGO bricks. The use of cellular automata is illustrated, and solves this classical content-based image retrieval problem in near realtime, with minimal memory usage.

Keywords: cellular automata, pattern matching, content-based image retrieval

1 Introduction

The use of cellular automata (CA) in image processing and graphical applications has received some attention over the past few years (for example, [3,6,10]). In this paper, CA are applied to the pattern matching of images in a content-based image retrieval (CBIR) system.

CBIR systems generally require the recognition of semantically equivalent subimages from a library of given images. For example, given a library of photographs, a requirement may be the retrieval of all photographs containing (any kind of) flower. The methods for solving this general problem, however, can be improved if specific sub-domains of images are considered. In addition, solutions for specific problems can then often be generalized to improve the general CBIR methods [4].

In this work, the specific domain of images is that of LEGO bricks. This seemingly frivolous domain contains many mathematically interesting aspects. For example, the image matching must be a semantically exact pattern matching, but the images themselves can differ in rotation, scale and color (for example, see figure 1). Moreover, a useful software implementation demands a realtime solution, which means that computationally expensive mathematical solutions are not appropriate. This work shows that the extraction of the semantic definition of a LEGO brick from a given image (the so-called feature extraction phase of this problem) can be implemented with CA. The CA solution allows for a direct parallel implementation, and is also implementable directly on the GPU – this implies that the use of the CA allows almost instantaneous feature extraction.

Section 2 contains the necessary background and definitions. The use of the CA for feature extraction is described in section 3. The results are analysed in section 4, and the conclusion is given in section 5.

2 Background and definitions

The relevant terminology and definitions for CA and CBIR are briefly summarized in this section.

Lynette van Zijl, Leendert Botha: Feature Extraction for Image Pattern Matching with Cellular Automata, pp. 3–14.
Proceedings of PSC 2009, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04403-2

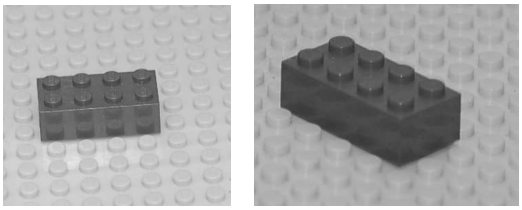

Figure 1. Two semantically equivalent LEGO bricks.

2.1 Cellular automata

We assume that the reader is familiar with CA as, for example, in [14]. We therefore summarize only the necessary definitions here.

A cellular automaton (CA) C is a k-dimensional array of automata. Each of the individual automata in the CA is said to occupy a *cell* in the CA. In the initial configuration of C, each automaton in C is in its initial state – this is typically refered to as time step $t = 0$. Each transition of C involves the *simultaneous* transitions of each of the individual automata. In addition, the individual automata are aware of the states of each of the other automata in the array, and the individual transitions may depend on the states of the other automata in the array. The global state of C thus evolves through time steps, where each time step describes the simultaneous changes in the individual automata.

In our case, CA are used to model images. Hence, only two-dimensional CA are considered, where each cell represents one pixel in the image plane. Furthermore, it is assumed that the individual automata in each cell are identical, and hence one transition function can be defined for the CA as a whole. Traditionally, each individual automaton is not dependent on all the other automata in the CA, but only on a subset of these automata. This subset is known as the *neighbourhood* of the automaton in the cell under consideration. We now formalize these intuitive concepts (see [8] for more detail):

Definition 1. *A 2D CA is a* 3*-tuple* $M = (A, N, f)$*, such that*

- A *is the finite nonempty state set,*
- $N = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$ *is the neighbourhood vector consisting of vectors in* $\mathbb{Z}^2$*, and*
- $f : A^n \to A$ *is the transition rule.*

Given a configuration c of the cells in the CA at a certain time t, the configuration c' at time $t + 1$ for each cell $\boldsymbol{x}$ can be calculated as

$$c'(\boldsymbol{x}) = f(c(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)) .$$

In such a 2D CA, specific neighbourhoods can be defined. For example, the so-called Von Neumann neighbourhood for a cell $x_{i,j}$ is defined as $\langle x_{i-1,j}, x_{i,j-1}, x_{i,j+1}, x_{i+1,j} \rangle$.

2.2 Content-based image retrieval

Given an image pattern p, CBIR requires that p is compared to a set of images P to find a set of semantically equivalent matches Q. To define semantic equivalence, certain characteristics (features) of the images must correspond within given boundaries.

The set of images P is preprocessed off-line to obtain a so-called feature vector for each image, and this feature vector is stored with each image. The search pattern p requires (realtime) preprocessing to obtain its feature vector, and the matching process then becomes a comparison of the feature vector of p against all the feature vectors in P. A distance measure between feature vectors is used to return all images in P which are semantically closely related to the search image p.

The preprocessing of p in the case of the LEGO domain requires a number of steps:

- **Baseboard elimination**: Each p is assumed to be an image of a LEGO brick on a so-called baseboard, which is a flat LEGO surface with studs (see figure 1). It is assumed that the baseboard has a color contrasting with the color of the brick. The first step then is to eliminate the baseboard from p.
- **Edge detection**: The brick itself is identified in p by finding all the edges belonging to the brick.
- **Stud location**: The positions of the studs are located in p.
- **2D**: From the stud locations, the top surface of the brick is identified by finding the edges closest to the studs.
- **Geometry**: Given the stud locations, the arrangement of the studs in a geometric pattern defines the final semantics of the brick.

This work covers the preprocessing of the search image p, and it is shown how to accomplish this task by using CA.

3 Feature extraction with CA

The aim of the preprocessing phase is to construct a feature vector, and this process is described in detail in this section.

3.1 Background elimination

To be able to calculate an accurate feature vector for p, all the pixels that correspond to the background must be eliminated. As stated before, the background in this case always consists of a LEGO baseboard which has a color distinguishable from the color of the brick. As an initial step, the color of the pixels on the edge of the image is subtracted from all pixels which have approximately the same color.

In figure 2, after the initial color subtraction, the reader may note that the baseboard studs are not fully eliminated. This is due to the fact that the studs form shadows, which are not of the same color as the baseboard. To eliminate these shadows, a CA is used.

Let p_1 be the image obtained from p after the baseboard color subtraction. Define a Von Neumann-type neighbourhood $\boldsymbol{n}_x$ for each cell $x_{i,j}$, such that $\boldsymbol{n}_x = (\boldsymbol{x}_N, \boldsymbol{x}_S, \boldsymbol{x}_W, \boldsymbol{x}_E)$, where

$$\begin{aligned}
\boldsymbol{x}_N &= \langle x_{i-\delta_L,j}, \ldots, x_{i-1,j} \rangle \\
\boldsymbol{x}_S &= \langle x_{i+\delta_L,j}, \ldots, x_{i+1,j} \rangle \\
\boldsymbol{x}_W &= \langle x_{i,j-\delta_L}, \ldots, x_{i,j-1} \rangle \\
\boldsymbol{x}_E &= \langle x_{i,j+\delta_L}, \ldots, x_{i,j+1} \rangle .
\end{aligned}$$

That is, the Von Neumann neighbourhood is taken in the usual four directions up to a distance of δ_L from the current cell. Suppose that a background pixel in cell $x_{i,j}$ is indicated by $x_{i,j} = 0$. A CA C_L can now be defined, with transition function

$$c'(x_{i,j}) = \begin{cases} 0, \text{ if } \exists k_N, k_S, k_W, k_E : x_{i-k_N,j} = 0 \ \& \\ \qquad\qquad\qquad\qquad\qquad\quad x_{i+k_S,j} = 0 \ \& \\ \qquad\qquad\qquad\qquad\qquad\quad x_{i,j-k_W} = 0 \ \& \\ \qquad\qquad\qquad\qquad\qquad\quad x_{i,j+k_E} = 0 \\ 1, \text{ otherwise,} \end{cases}$$

where $1 \leq k_N, k_S, k_W, k_E \leq \delta_L$. That is, if there is any background pixel found within a distance of δ_L in all four directions from the current cell, then the cell is taken to be a background cell and will be eliminated. The number of iterations required to eliminate background pixels with this CA method is linearly dependent on the size of the original image p, and the relative size of the brick against the size of the background baseboard.

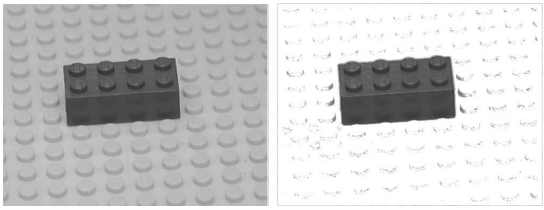

Figure 2. The original image on the left and the image after removing background pixels, based on their color. The small border around the picture indicates which pixels were used to determine the background color.

Some careful consideration will show that there is one instance where CA C_L will fail to remove all background pixels. This occurs under certain lighting conditions of p, when the background pixels form a straight line. In this scenario, there will be background pixels in only one or two directions from a given cell, and hence the line will not be removed. This is clear from the definition of the transition function of CA C_L above.

A second CA C_S can be constructed to eliminate the straight lines. This CA uses a smaller distance, δ_S, to look in all four directions. In contrast to C_L, a pixel is identified as background if *either* the horizontal or the vertical directions contain background pixels within the distance δ_S. Hence, let p_2 be the image obtained from p_1 after the background elimination described above. Again, define a Von Neumann-type neighbourhood $\boldsymbol{n}_x$ for each cell $x_{i,j}$, such that $\boldsymbol{n}_x = (\boldsymbol{x}_N^S, \boldsymbol{x}_S^S, \boldsymbol{x}_W^S, \boldsymbol{x}_E^S)$, where

$$\begin{aligned} \boldsymbol{x}_N^S &= \langle x_{i-\delta_S,j}, \ldots, x_{i-1,j} \rangle \\ \boldsymbol{x}_S^S &= \langle x_{i+\delta_S,j}, \ldots, x_{i+1,j} \rangle \\ \boldsymbol{x}_W^S &= \langle x_{i,j-\delta_S}, \ldots, x_{i,j-1} \rangle \\ \boldsymbol{x}_E^S &= \langle x_{i,j+\delta_S}, \ldots, x_{i,j+1} \rangle . \end{aligned}$$

That is, a Von Neumann neighbourhood in all four directions up to a distance of δ_S is used. Suppose that a background pixel in cell $x_{i,j}$ is indicated by $x_{i,j} = 0$. The transition function is then defined as

$$c'(x_{i,j}) = \begin{cases} 0, \text{ if } \exists k_N^S, k_S^S : x_{i-k_N^S,j} = 0 \ \& \ x_{i+k_S^S,j} = 0 \\ 0, \text{ if } \exists k_W^S, k_E^S : x_{i,j-k_W^S} = 0 \ \& \ x_{i,j+k_E^S} = 0 \\ 1, \text{ otherwise .} \end{cases}$$

Thus, a single subtraction of image pixels, followed by an application of CA C_1, followed by an application of CA C_2, yields the desired results, with all of the background pixels removed. The result is illustrated in figure 3.

Figure 3. The image after applying CAs C_L and C_S.

Once the background has been eliminated from the given image, one can proceed to find the edges of the brick itself.

3.2 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.

Our solution implements a CA-based method originally proposed by Popovici *et al* [10]. Let $\varphi(a, b)$ define a similarity measure between pixels a and b. The simplest example of such a similarity measure is the Euclidean distance in RGB-space [1], so that $\varphi(a, b) = \|a - b\|$. Hence, the value of $\varphi(a, b)$ decreases as the similarity between pixels a and b increases, so that $\varphi(a, a) = 0$.

Let ϵ be a specified lower threshold. Then define the CA C_e with the transition function as given below:

$$c'(x_{i,j}) = \begin{cases} 0, \text{ if} & \varphi(x_{i,j}, x_{i,j-1}) < \epsilon \;\&\; \varphi(x_{i,j}, x_{i,j+1}) < \epsilon \;\& \\ & \varphi(x_{i,j}, x_{i-1,j}) < \epsilon \;\&\; \varphi(x_{i,j}, x_{i+1,j}) < \epsilon \\ x_{i,j}, \text{ otherwise .} & \end{cases}$$

Again, the neighbourhood is clearly a Von Neumann neighbourhood, and in this case the distance is 1.

Sample output from C_e is shown in figure 4.

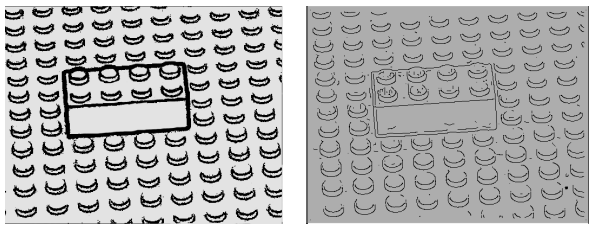

Figure 4. The edge detected images using cellular automaton C_e.

[1] Both Euclidean distance and vector angle were implemented as similarity measure, in both RGB space and YIQ space, in the software.

3.3 Feature extraction

The semantics of a LEGO brick is determined by its form, the number of studs and the arrangement of the studs[2]. For example, consider figure 5. Brick number 1 is a rectangular 2 by 4 brick. It has eight studs that are arranged in two rows of four studs each, in straight lines. There are also rounded bricks (brick number 3), macaroni bricks (brick number 4), and L-shaped bricks (brick number 6). Note that bricks number 2 and 3 have the same number of studs in the same arrangement, but their edges define the bricks to be semantically different. Also, brick number 2 and brick number 5 have the same number of studs (namely, four each), but in a different arrangement and hence these two bricks are also semantically different.

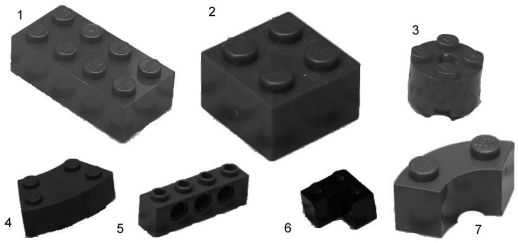

Figure 5. The semantic forms of LEGO bricks.

It is now necessary to find a feature vector that mathematically describes a LEGO brick, based on its form, number of studs, and stud arrangement. These characteristics are to be extracted and combined into a single feature vector for any given brick. These steps are discussed below.

Stud location The edges of a stud are difficult to find with standard shape-detection methods, as the edges have a distinctive halfmoon shape (see figure 6). A possible solution is to use template matching, where template shapes are moved around the image until a location is found which maximizes some match function. A popular match function is the squared error [13]:

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

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

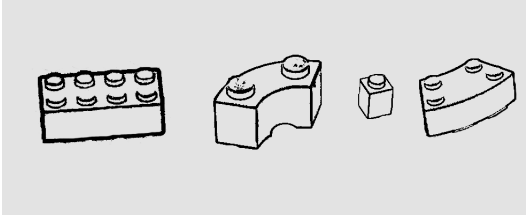

Figure 6. Four edge detected bricks showing similarity in the shape of the studs.

[2] In some user-defined cases, the color of the brick may also be used as an additional semantic feature.

The template used for the studs of the LEGO bricks is shown in figure 7. Note that templates of different size are provided, as scaling is not accurate in this specific case. The score of the matching function is scaled relative to the size of the template to prevent larger templates from getting higher scores.

Figure 7. The set of templates used in the template matching process.

Given the stud locations, the next step is to determine the stud formation.

Stud formation After the previous step, the center points of the locations of all the studs are available. The next step is to define the formation in which the studs occur. In other words, the number of rows and columns of the studs have to be extracted. For example, a brick with eight studs may have two rows of four studs each, or one row of eight studs.

Hence, it is necessary to find the minimal set of straight lines $\mathcal{L} = \{l_1, l_2, \ldots, l_n\}$ where each stud lies on exactly one l_i. Note that the stud location is point-specific, so that a point is deemed to lie on a line if it is within a given perpendicular small distance from the line.

Our algorithm is given below (see Algorithm 1), and is described in more detail in [1]. The output of Algorithm 1 gives the number of studs, and the number of lines needed to cover those studs, and these are used directly in the final feature vector.

Next, the form of the brick must be determined. A LEGO brick has a three-dimensional form, defined by both the inside and outside edges of the brick. The standard three-dimensional matching algorithms available in the literature would have been too computationally expensive in this case [13]. We therefore simplified the problem to a two dimensional problem, by the observation that all LEGO bricks are rectangular protrusions of the top surface of the brick [3]. Hence, it is only necessary to identify the top surface.

Identifying the top surface The top surface of the brick is identified by finding the edges surrounding the stud locations found in the previous step. This is a three step process: first, the edges of the studs themselves are removed by subtraction. This leaves random noise on the top surface, which is removed with a CA similar to the CA used to eliminate the background. Lastly, a CA is defined to flood in all directions, from the stud locations to the nearest edge.

The first step (removing the stud edges) is simply done by subtracting the matching template shape. This results in random noise, as the templates are not a perfect pixel-by-pixel match. The CA C_1 as defined previously, is used to remove this noise.

The flooding process to find the edges on the top surface of the brick, is again easily defined with a CA C_f. Initialize C_f so that there are four possible states in each cell: *background*, *edge*, *top surface* and *not top surface*. All the pixels where there were studs, are identified as top surface, and any cell that is not background, edge or top surface, is identified as *not top surface*.

[3] We only consider 'standard' LEGO bricks in this work. Other forms (such as sloped bricks, or bricks with a base larger than the top, such as cones) will be considered in subsequent work.

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
```

The neighbourhood to be used in C_f is a Moore neighbourhood [4], with a specified distance ns. The transition function then considers each cell. If it is not a top surface cell, then the cell changes into a top surface cell if it is adjacent to any top surface cell in the Moore neighbour and it is neither edge nor background. That is, from the stud locations, the neighbours of each cell are considered. Count the number of neighbours that are not edge or background. If this number exceeds a given threshold, then the current cell is top surface. Formally, let th be the threshold and ns the neighbourhood size. Let a top surface cell be represented by 0, edges by 1, and background by 2. Then C_f has the transition function

$$c'(x_{i,j}) = \begin{cases} 0, \text{ if} & c(x_{i,j}) \neq 0 \ \& \ c(x_{i,j}) \neq 2 \ \& \\ & (\Sigma_{m,n} x_{m,n} = 0) > th, \\ & \text{where } i - ns/2, j - ns/2 < m, n < i + ns/2, j + ns/2, \\ 1, \text{ otherwise .} & \end{cases}$$

An example of the stud edge, noise removal and flooding is shown in figure 8.

At this point, if color is to be used as a distinguising feature, a standard 256-bin histogram [11] is used to construct the color information for the brick.

It is now finally possible to encode the feature vector of a given LEGO brick, based on its number of studs, stud locations, form, and color. In our case, we used the Hu-set of invariant moments [7] to encode the feature vector with the information extracted from the image p.

[4] A Moore neighbourhood consists of all eight cells surrounding the current cell.

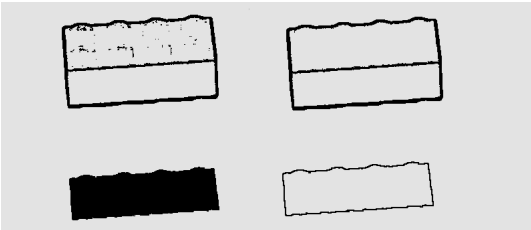

Figure 8. The output of the flooding process. The top left picture shows the edges after the studs have been removed. A CA is applied to remove the noise and the output is shown in the top right picture. The flooded region is shown on the bottom left and the boundary of this region, which is to be encoded into a feature vector, is shown on the bottom right.

Once the feature vector has been calculated for the search image p, that vector can be matched against all the pre-calculated images in the database. Our software can handle multiple search criteria on any of the elements in the feature vector, and hold a match score so that a set of best possible matches can be retrieved.

4 Analysis

This section illustrates some of the results in the final implemented CBIR system. More details, and comparative results with more traditional approaches, are discussed in [1]. In our initial experiments, bricks were correctly identified in almost 80% of our test cases.

An example of the shape-based retrieval is illustrated in figure 9. The bricks are shown in best match order (the lower the number, the better the match). In figure 9, note that an identical brick to the search image p was the best match, followed by two other curved bricks, while the rectangular bricks were the worst matches. Note that any shape is described by the Hu-set of invariant moments. In comparing two shapes, the Euclidean distance between the two shapes is calculated – the smaller the distance, the better the match. In figure 9 below, it therefore follows that the macaroni-shaped bricks are nearer to each other than to rectangular bricks.

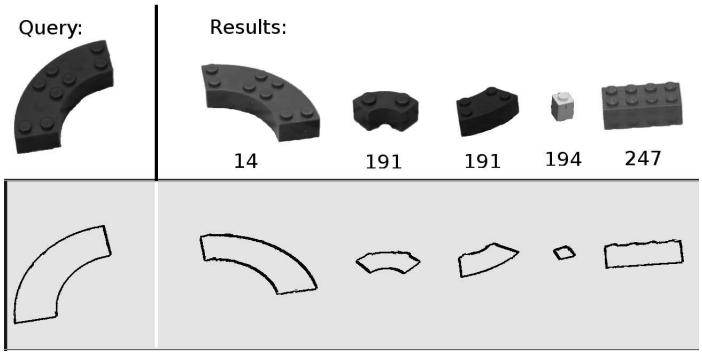

Figure 9. A sample shape retrieval query with match scores presented in thousands.

If an image is not of sufficient quality, the edge detection can result in discontinuous edges. This invalidates the flooding process. Recall that the flooding process terminates when an edge is encountered. Figure 10 shows an example of a brick for

which the flooding process fails. Here, note that the brick does not have a continuous edge separating the top surface from the rest of the brick. Hence, the entire brick is flooded, resulting in an inaccurate feature vector.

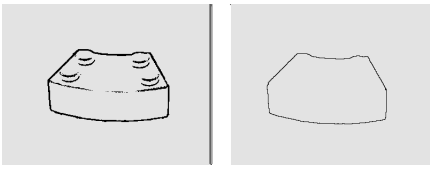

Figure 10. Shape extraction process fails for a brick that does not have a continuous edge separating the top surface from the rest of the brick.

To solve the problem, one can simply adjust the threshold of the edge detector CA (this is a parameter which can be set by the user in our software). As long as the image p is of sufficient quality, the increased threshold will always result in a continuous edge. Figure 11 illustrates a changed threshold and consequent successful edge detection.

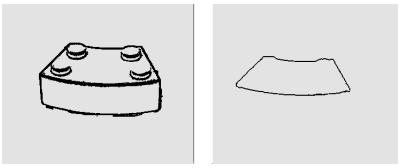

Figure 11. The same brick from figure 10, using a better threshold, and resulting in a correct identification of the top surface.

Almost all mismatches are due to an input image p where the edge detection fails. Failed or incorrect edge detection are due primarily either to a threshold that is not high enough for the picture quality (see below), or to distracting features which result in an incorrect edge detection. Figure 12 shows some examples of bricks that will not be correctly identified. The brick on the left is the same colour as the background, and hence is eliminated during the background elimination phase. The brick on the right results in false edges, due to the vertical stripes. This leads to a feature vector with an incorrect shape description for a 1×2 brick.

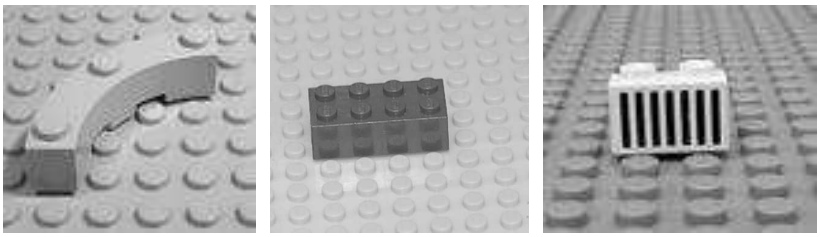

Figure 12. Bricks that cannot be recognized, due to background colour (left), lighting conditions (middle), and distracting features (right).

4.1 Comparison with existing systems

General content-based image retrieval systems cannot be directly compared to our system. Most CBIR system (such as the FIRE search engine [5]) classify images into

broad groups and find matching topics. For example, given an image of a LEGO brick, the FIRE engine would return a wide variety of images of toys, with at best a few LEGO bricks.

We can compare at least one part of the image pre-processing with other algorithms, namely, the edge detection. There are many different edge detection algorithms and systems. In general, the more accurate the edge detection, the longer the algorithm takes to execute. For example, the well-known Canny [2] and SUSAN [12] edge detectors are extremely accurate, but too slow for real-time analysis. Other less accurate methods have other issues that make their use difficult in this domain. For example, the Marr-Hildreth algorithm [9] lacks in the localization of curved edges, which is essential in the LEGO images. It is also interesting to note that the more accurate edge detectors result in thin edges (see figure 13), while the rest of our algorithms such as flooding and template matching, work best with thick edges.

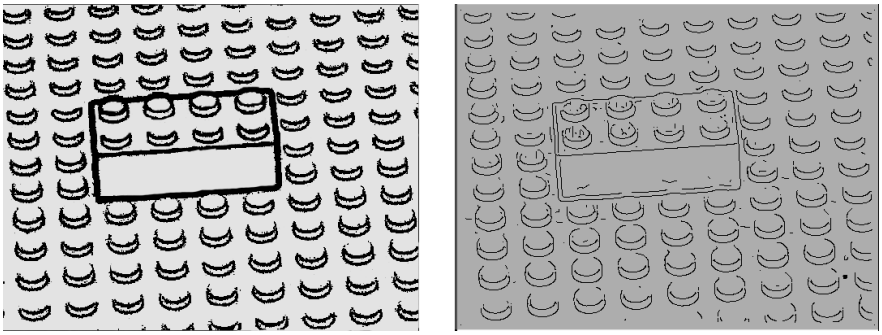

Figure 13. Results from he CA edge detector (left) versus the Canny edge detector (right).

5 Conclusion

We showed that cellular automata can be successfully applied for the realtime retrieval of LEGO brick images. The advantage of this approach is the limited memory use and fast execution time of a CA implementation.

We showed that it is possible to simplify the three-dimensional shape extraction problem to a two-dimensional case for the LEGO brick. We implemented a fully functional CBIR system based on the CA feature extraction, and illustrated the results.

For future work, we intend to extend this work to more general LEGO bricks. In particular, we want to consider those cases that are not simply protrusions of a brick with rectangular stud formations.

References

1. L. Botha: *A CBIR system for LEGO brick image retrieval*, tech. rep., Stellenbosch University, 2008.
2. J. Canny: *A computational approach to edge detection.* IEEE Trans. Pattern Anal. Mach. Intell., 8(6) 1986, pp. 679–698.
3. C. Chan, Y. Zhang, and Y. Gdong: *Cellular automata for edge detection of images*, in Proceedings of the Third International Conference on Machine Learning and Cybernetics, August 2004, pp. 3830–3834.

4. R. Datta, D. Joshi, J. Li, and J. Wang: *Image retrieval: ideas, influences, and trends of the new age.* ACM Computing Surveys, 40(2) April 2008.
5. T. Deselaers, D. Keysers, and H. Ney: *Features for image retrieval – a quantitative comparison*, in In DAGM 2004, Pattern Recognition, 26th DAGM Symposium, number 3175 in LNCS, 2004, pp. 228–236.
6. S. Druon, A. Crosnier, and L. Brigandat: *Efficient cellular automata for 2D/3D free-form modeling.* Journal of Winter School of Computer Graphics, 11(1) 2003, pp. 102–108.
7. K. Hu: *Visual pattern recognition by moment invariants.* IRE Transactions on Information Theory, IT-8 February 1962, pp. 179–187.
8. V. Lukkarila: *On undecidability of sensitivity of reversible cellular automata*, in AUTOMATA 2008, 2008, pp. 100–104.
9. D. Marr and E. Hildreth: *Theory of edge detection.* Proceedings of the Royal Society of London. Series B, Biological Sciences, 207(1167) 1980, pp. 187–217.
10. A. Popovici and D. Popovici: *Cellular automata in image processing*, in Proceedings of the 15th International Symposium on Mathematical Theory of Networks and Systems, University of Notre Dame, 2002.
11. S. Siggelkow: *Feature Histograms for Content-Based Image Retrieval*, PhD thesis, Albert-Ludwigs-Universität Freiburg, Fakultät für Angewandte Wissenschaften, Germany, Dec. 2002.
12. S. Smith and J. Brady: *Susan - a new approach to low level image processing.* International Journal of Computer Vision, 23 1997, pp. 45–78.
13. W. Snyder and H. Qi: *Machine Vision*, Cambridge University Press, New York, NY, USA, 2003.
14. S. Wolfram: *Cellular Automata and Complexity*, Westview Press, 1994.