Computational Design of LEGO® Sketch Art

MINGJUN ZHOU and JIAHAO GE, The Chinese University of Hong Kong, Hong Kong, China
HAO XU, Qianzhi Technology Inc., China
CHI-WING FU, The Chinese University of Hong Kong, Hong Kong, China

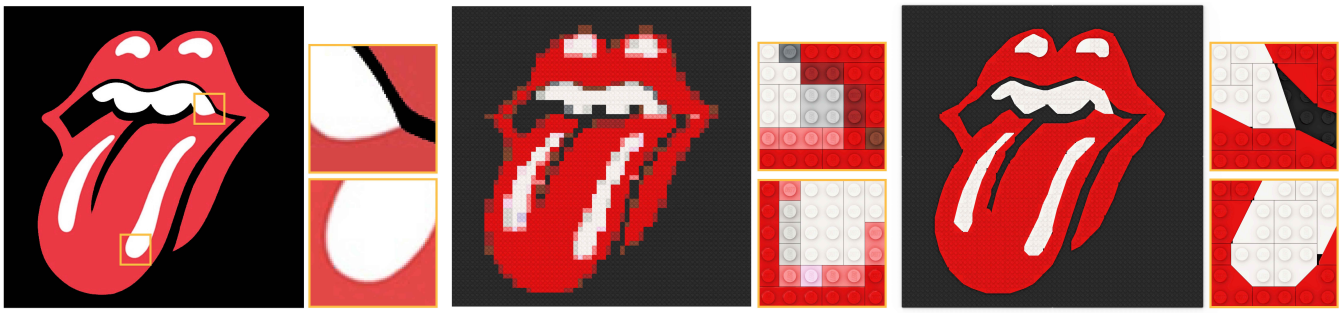


Fig. 1. A free-form LEGO® sketch art (c) automatically generated by our approach. The result comprises not only rectangular bricks but also bricks with sloping and rounding edges for reproducing the input image (a), which features the Rolling Stones' Tongue and Lips logo ®Musidor B.V. Compared with the generic LEGO® mosaic (b) created by the commercial software Stud.io [Bri 2023], our result better preserves the details, smooth curves, and sharp features on the same 64 × 64 LEGO® baseplate. See also Figure 2 (d) for the official design. Its resolution is 62 × 72 vs. ours with 47 × 56 (black margins in (c) excluded).

This paper presents computational methods to aid the creation of LEGO®[1] sketch models from simple input images. Beyond conventional LEGO® mosaics, we aim to improve the expressiveness of LEGO® models by utilizing LEGO® tiles with sloping and rounding edges, together with rectangular bricks, to reproduce smooth curves and sharp features in the input. This is a challenging task, as we have limited brick shapes to use and limited space to place bricks. Also, the search space is immense and combinatorial in nature. We approach the task by decoupling the LEGO® construction into two steps: first approximate the shape with a LEGO®-buildable contour then filling the contour polygon with LEGO® bricks. Further, we formulate this contour approximation into a graph optimization with our objective and constraints and effectively solve for the contour polygon that best approximates the input shape. Further, we extend our optimization model to handle multi-color and multi-layer regions, and formulate a grid alignment process and various perceptual constraints to refine the results. We employ our method to create a large variety of LEGO® models and compare it with humans and baseline methods to manifest its compelling quality and speed.

CCS Concepts: • **Computing methodologies** → **Shape modeling**.

[1]LEGO® is a trademark of the LEGO® Group, which does not sponsor, authorize or endorse this work. All information in this paper is collected and interpreted by its authors and does not represent the opinion of the LEGO® Group. Permission granted by LEGO®. All rights reserved.

Authors' addresses: Mingjun Zhou, mingjunzhou@link.cuhk.edu.hk; Jiahao Ge, spikege@163.com, The Chinese University of Hong Kong, Hong Kong, China; Hao Xu, hao.xu@maic.fun, Qianzhi Technology Inc., China; Chi-Wing Fu, cwfu@cse.cuhk.edu.hk, The Chinese University of Hong Kong, Hong Kong, China.

0730-0301/2023/12-ART201 $15.00
https://doi.org/10.1145/3618306

Additional Key Words and Phrases: LEGO®, computational design, fabrication, assembly

ACM Reference Format:
Mingjun Zhou, Jiahao Ge, Hao Xu, and Chi-Wing Fu. 2023. Computational Design of LEGO® Sketch Art. *ACM Trans. Graph.* 42, 6, Article 201 (December 2023), 15 pages. https://doi.org/10.1145/3618306

1 INTRODUCTION

LEGO® has emerged as a unique medium for artistic expression. By skillfully arranging bricks of various forms and colors on the baseplate, one may create LEGO® models to depict images, shapes, and artworks. A common approach to create LEGO® art is by downsampling a digital image then representing each pixel by a unit-square LEGO® plate of similar color as the pixel. The result is known as the LEGO® mosaic, which can be easily realized by computer software and has been integrated into many commercial websites, *e.g.*, [me 2022; Xplicator 2020]. Figure 1 (b) shows an example LEGO® mosaic generated by the widely-used software Stud.io. [Bri 2023].

However, LEGO® mosaics have limited expressiveness. For instance, a 64 × 64 LEGO® mosaic already involves more than three thousand LEGO® pixels, reaching 51.2cm × 51.2cm in physical size, as shown in Figure 1 (b). Hence, considering the manual assembly workload and space for displaying the mosaic, the resolution of LEGO® mosaics cannot be too large. Therefore, they often suffer from loss of image details and have limited capability of reproducing free-form smooth curves and sharp features.

To improve the expressiveness of LEGO® arts in a limited physical space, LEGO® designers, leveraging their professional experience, are able to skillfully use both rectangular and non-rectangular LEGO® bricks to create designs with free-form boundaries. Such form of LEGO® art is called the LEGO® sketch; see Figure 2. Specifically, LEGO® sketch employs coherently-connected bricks with

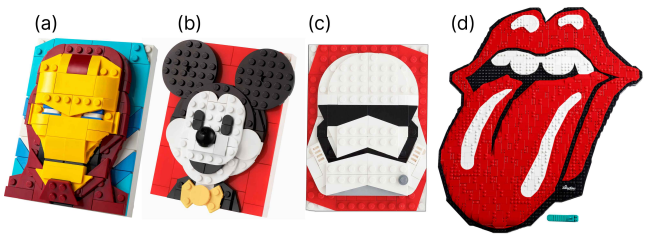


Fig. 2. LEGO® sketch models ©LEGO® Group designed by human experts. Each model reflects a deep understanding of available brick types, careful brick placement, and numerous iterations to explore different potential brick layouts.

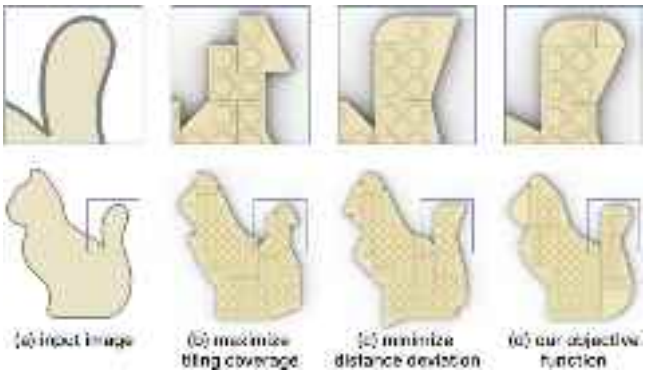


Fig. 3. Given the shape in (a), simply maximizing the tiling coverage or minimizing the distance deviation between shape and tiled boundary cannot produce LEGO® models (b & c) that well preserve the overall perception of the shape, as compared with our carefully-designed approach (d).

sloping and rounding edges to approximate free-form curves. Compared to LEGO® mosaics, LEGO® sketches are hard to design. They require tremendous manual effort to iteratively refine a design, through trial and error, by testing the construction feasibility and perceptual effect. At present, LEGO® sketch products are mostly sold as off-the-shelf items. Personalized designs remain costly.

In this work, we aim to develop computational methods to help create customized LEGO® sketch models from simple images. Like generic sketch models, we target images with simple structures, *e.g.*, clip arts, cartoon characters, and logos. As demonstrated in our result shown in Figure 1 (c), our tool should automatically generate a well-connected LEGO® model that resembles the input, while respecting the smooth curves and sharp features in the input.

Fundamentally, the main task is to create a tiling with LEGO® tiles/plates that best approximates each color region in the input image, without overlaps or holes; see Figure 1(c) for an example result. When there is only a single color region in the input, the task is very like a general tiling problem, yet with the following deviations: (i) we have limited planar LEGO® tiles/plates that can be used; (ii) the tiling placement locations have to follow the studs on top of the LEGO® baseplate, rather than being arbitrary and continuous in space; and (iii) we need to create not only a seamless tiling but also approximate the shape of the input color region. Also, we should handle inputs with multiple color regions, and we may arrange tiles over multiple layers; see Section 7 for examples.

This task is non-trivial. First, we have to carefully utilize limited LEGO® bricks of simple geometry to form non-regular structures, while trying to maintain the visual perception. The challenge is amplified when considering the limited baseplate space, in which a slight modification in any brick could cause significant ripple effect in the whole design. Second, the search space is immense and convoluted; we have to face the combinatorial explosion in LEGO® brick placements. Further, the objective is multifaceted, encompassing boundary smoothness, sharp features preservation, and layout seamlessness. We cannot simply maximize the tiling coverage or minimize the distance deviation; see Figure 3.

To approach this complex combinatorial optimization problem, we propose the following ideas. (i) We propose to decouple the LEGO® construction process into two steps. Given a shape/region to be approximated by LEGO® bricks, we first construct a LEGO®-buildable *contour polygon* on the baseplate that resembles the input shape; we call this step the "contour approximation"; then, we can fill the polygon with non-overlapping LEGO® bricks to produce the LEGO® model. (ii) To achieve the contour approximation, we propose to formulate it into a graph optimization. Procedure-wise, we first enumerate all potential brick placements around the input shape's boundary and collect all edge segments of the bricks that may contribute to form the contour. Then, we represent the candidate edge segments and their connections using a graph data structure and solve for the contour polygon (which is a loop in the graph) that best approximates the input shape/region. (iii) Further, we show that our graph optimization model can be extended to handle multiple regions of different colors and also regions over multiple layers. (iv) Lastly, we propose further strategies to improve the quality of our results: a grid alignment step to better align the input with the baseplate's grid structure and various user-specified perceptual constraints that can be built into the graph optimization to account for symmetry and small salient features in the input.

We demonstrate the capability of our tool by creating a wide array of LEGO® sketch models of varying sizes and complexities, and construct physical assemblies for some of them. We evaluate our method by comparing it with general users and experienced designers, demonstrating its compelling quality and speed, though our results consume significantly less time to create. Last, we compare our method with three baselines and evaluate the robustness of our method on brick set and baseplate resolution.

Overall, this work has the following contributions.

(i) We introduce the first computational framework to aid the design and creation of LEGO® sketch models. It is a complete pipeline that can automatically select and arrange LEGO® bricks (including regular rectangular bricks and also bricks with sloping and rounding edges) to construct LEGO® sketch models from simple images.
(ii) We meticulously model the representational capacity of various LEGO® bricks, decouple and formulate the construction process into a graph optimization, and model the construction requirements into quantifiable objectives.

(iii) We extend our optimization model to handle regions of different colors and regions over multiple layers. Besides, we further enhance our results by developing a grid alignment process and various perceptual constraints.

2 RELATED WORK

In this section, we discuss assorted areas of related works.

Automated LEGO® generation. First, we discuss computational methods for automatically generating LEGO® models. [Gower et al. 1998] formulate the task as a combinatorial optimization; since then, there are several follow-ups by exploring, e.g., graph grammar [Peysakhov et al. 2000], cellular automata [Smal 2008], genetic algorithm [Lee et al. 2015], and metaheuristic [Kollsker and Stidsen 2021]. Considering model stability and brick colors, Luo et al. [2015] devised a force-based analysis for LEGO® models composed of regular bricks. Hong et al. [2016] studied engraved LEGO® models for computational efficiency. Yun et al. [2017] used silhouette fitting to improve the voxelization and thus the quality of the generated LEGO® models. Please refer to [Kim et al. 2014] and [Kollsker 2020] for more comprehensive surveys on the earlier works.

Some variants to the task have been explored in recent years. Kuo et al. [2015] present Pixel2Brick, a computational framework to automatically construct brick sculptures from pixel art. Xu et al. [2019] designed a method to automatically generate LEGO® Technic models from sketches, considering LEGO® Technic beams, pins, and axles, as well as their connections. Liu et al. [2022] computationally improves the rigidity of Technic models during the design process. Zhou et al. [2019] considered sloping and circular bricks in addition to regular cuboid bricks for generating houses with pillars and sloping rooftop. Kollsker and Malaguti [2021] studied the optimization of 2D LEGO® constructions built from cuboid bricks for forming a planar vertical LEGO® wall that can reproduce a given 2D digital image.

So far, existing works consider mainly regular cuboid bricks, due to their simplicity in modeling and computation. In this work, we go beyond regular cuboid bricks, considering many non-rectangular tiles and plates to build the LEGO® designs. Compared with existing pixel-based approaches, *e.g.*, LEGO® mosaic, our approach can enhance the expressiveness of the LEGO® models. It can automatically generate LEGO® designs that better approximate the inputs with smooth curves and sharp features in the same physical dimension.

Apart from LEGO® models generation, Kohama et al. [2021] computed feasible sequences to assemble a LEGO® model in an additive manufacturing manner. Wang et al. [2022] explored a neural network to predict a machine-executable construction plan from a LEGO® instruction manual. Besides, LEGO® bricks are ubiquitous in various research contexts, e.g., STEM education [Gao et al. 2022], material-economic 3D fabrication [Chen et al. 2018], etc.

Tiling for patterns generation. Tiling patterns with repetitive units have a long history. Many problem instances have been studied and categorized [Grünbaum and Shephard 1987]. Similar to our work, many of these works focus on tiling the plane, for example, Escherization [Kaplan and Salesin 2000; Nagata and Imahori 2021], decorative mosaics [Hausner 2001; Smith et al. 2005], collages [Kwan et al. 2016], checkerboard patterns [Peng et al. 2019], and wooden parquetries [Iseringhausen et al. 2020], *etc.* Recently, Xu et al. [2020] developed a graph neural network, trying to produce non-periodic tilings of arbitrary 2D shapes. On the other hand, tilings on two-manifolds have been studied in [Chen et al. 2017; Deuss et al. 2014; Eigensatz et al. 2010; Fu et al. 2010; Gavriil et al. 2020; Jiang et al. 2021; Liu et al. 2021], which aim to cover freeform surfaces with tile instances from a small set of *fixed* panels or blocks.

Our task deviates from standard tiling tasks with additional constraints and objectives, focusing more on the perceptual aspects. To this end, we develop a novel formulation to meet the needs of the task for effectively creating artistic LEGO® sketch designs.

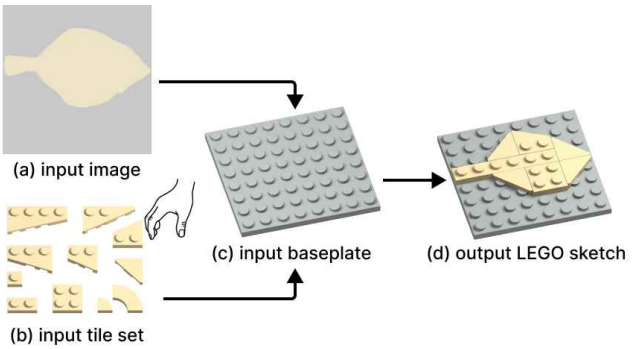


Fig. 4. From a given image (a), our goal is to arrange LEGO® tiles (*e.g.*, from the tile set in (b)) on a baseplate (c) (8x8 in this example) to produce a LEGO® sketch model (d) that resembles the shape in the given input.

2D shape descriptors. To evaluate the quality of LEGO® sketch models requires comprehensive measures, *e.g.*, shape descriptors for measuring how similar the result is to the input. Global shape descriptors, such as Fourier descriptor [Persoon and Fu 1977], wavelet descriptor [Chuang and Kuo 1996], beam angle statistics [Arica and Yarman Vural 2003], chordiogram [Toshev et al. 2012], and shape vocabulary [Bai et al. 2014] compactly extract global characteristics of a shape and measure the overall similarity between shapes. Local descriptors, such as [Asada and Brady 1986; Belongie et al. 2002; Kwan et al. 2016], describe local regions of shapes using various local features such as curvatures. Yet, existing shape descriptors are designed mainly for shape matching, thus tending to miss slight local variation and deformation that may undermine the user perception. In response to this, we formulate an objective in our approach that considers both global visual perception and local curve characteristics, striving to offer a comprehensive and efficient evaluation for supporting the generation of LEGO® sketch models.

3 OVERVIEW

Problem definition. Figure 4 illustrates the goal of our tool. The inputs include (i) a simple image with one or few regions; (ii) a LEGO® baseplate of given dimensions; and (iii) a tile set of LEGO® plates and tiles; see the full tile set in Figure 6. Using both regular rectangular bricks and non-regular bricks with sloping/rounding edges, we aim to create a LEGO® model that resembles the input

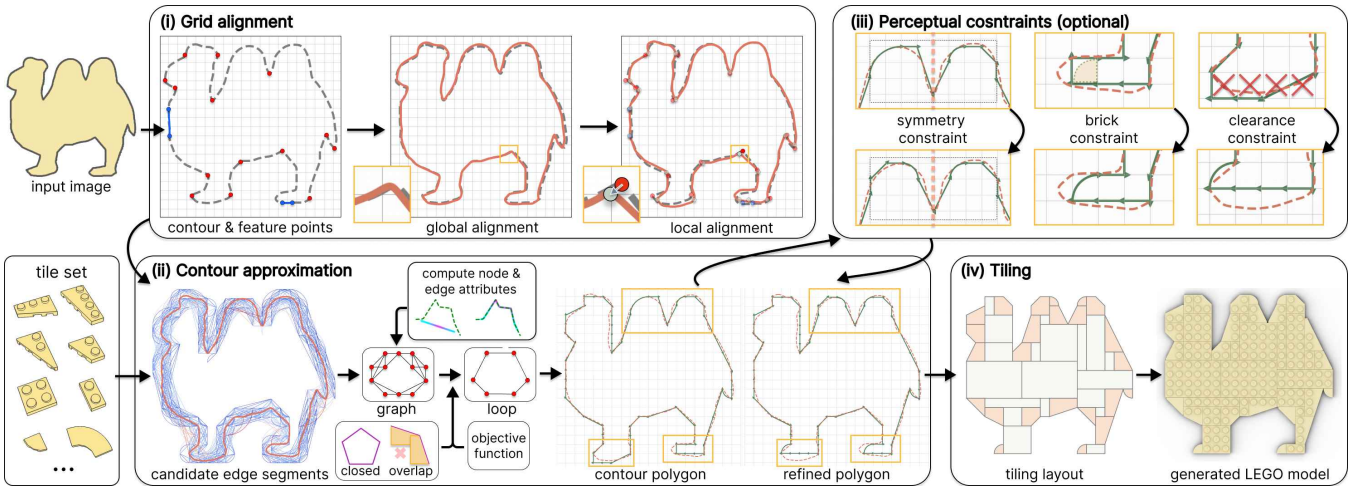


Fig. 5. Overview of our approach. (i) Extract the shape's contour, detect feature points (red & blue dots), and slightly deform the contour to align it with the baseplate's grid structure. (ii) Compute a LEGO®-buildable contour polygon on baseplate that best approximates the grid-aligned contour by enumerating all possible candidate edge segments that can be covered by LEGO® tiles around the contour, then formulating a graph optimization to solve for the contour polygon. (iii) Optionally, provide perceptual constraints to refine the polygon. (iv) Arrange LEGO® tiles to first cover the sloping and rounding edges on the contour polygon then fill the remaining interior with minimum number of rectangular LEGO® tiles.

image, preserving the smooth curves and sharp features in the input, without brick overlaps and holes in the tiled region.

To aid the understanding of our method, in Sections 4 to 6, we first present our method on an input image with only a single simply-connected region. Then, in Section 7, we discuss extensions to handle regions of multiple colors and regions over multiple layers.

Challenges. First, LEGO® plates and tiles are mostly simple shapes such as rectangles, trapezoids, and quarter circles; see again Figure 6. They cannot create structures that are too thin or too small. Also, they cannot perfectly reproduce any boundary line, due to the limited orientations provided by the sloping and rounding edges.

Second, the LEGO® baseplate has a limited resolution for brick placement. Hence, losing the image details in the input is often unavoidable. Yet, we should strive to find the best approximation of the input by carefully planning the bricks in this confined space.

Third, brick arrangement is essentially a combinatorial optimization problem. We have an immense and discrete solution space, due to the flexible brick choices and brick orientations, as well as extensive combinations of adjacent brick placements. Here, a minor alteration to a single brick can substantially impact the adjacent ones, potentially triggering a complete redesign of the assembly.

Lastly, the task involves multiple objectives, requiring simultaneous consideration of several design factors. Beyond the fundamental ones on brick connections and assemble-ability (*e.g.*, collision-free), we should try to account for the perceptual resemblance and boundary smoothness, as well as the salient features in the input.

Our approach. Figure 5 provides an overview of our approach, which consists of the following four major steps:

(i) *Grid alignment* aims to slightly deform the contour of the input shape to align it with the baseplate's grid structure. As Figure 5(i) shows, we first slightly shift and scale the whole contour (global alignment), then locally adjust different segments (local alignment), to better align it with the grid for constructing a LEGO® model that better approximates the input shape; see Section 4.

(ii) *Contour approximation* aims to produce a boundary polygon on baseplate to best approximate the input contour; see Figure 5(ii). First, along the contour, we enumerate all candidate edge segments, which are reproducible by bricks in the tile set, to cover the entire contour. Then, we formulate a graph data structure to represent these edge candidates and their connections, remove infeasible edge connections not realizable by the LEGO® tiles, and formulate a graph optimization to solve for the contour polygon that best approximates the input shape's contour; see Section 5.

(iii) *Perceptual constraint* is an optional step (Figure 5(iii)), in which perceptual constraints, *e.g.*, a partial symmetry constraint, can be provided to further refine the result; see Section 6.

(iv) The *Tiling* step further fills the contour polygon with LEGO® bricks; see Figure 5(iv). In detail, we first arrange bricks to cover the sloping and rounding edges on the polygon, then fill the remaining interior with minimum number of rectangular

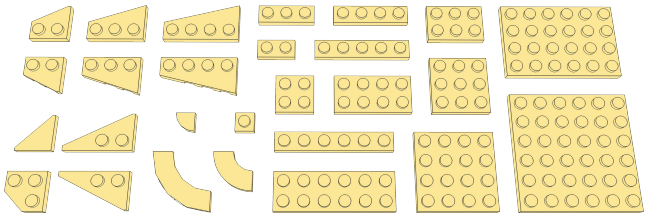


Fig. 6. The tile set we employed includes the above LEGO® plates and tiles.

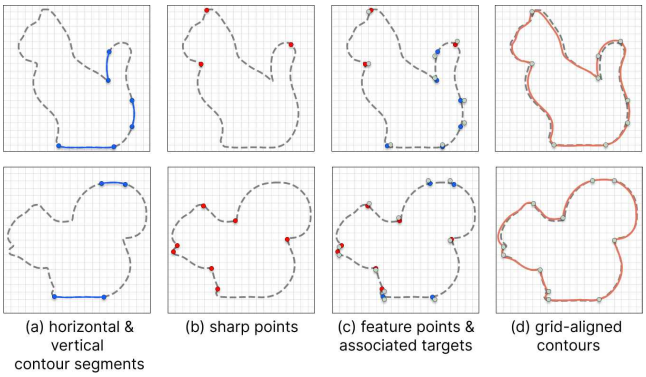


Fig. 7. (a)-(c) Preparation before adjusting the shape's contour, for aligning it with the baseplate's grid structure. (d) Grid alignment results.

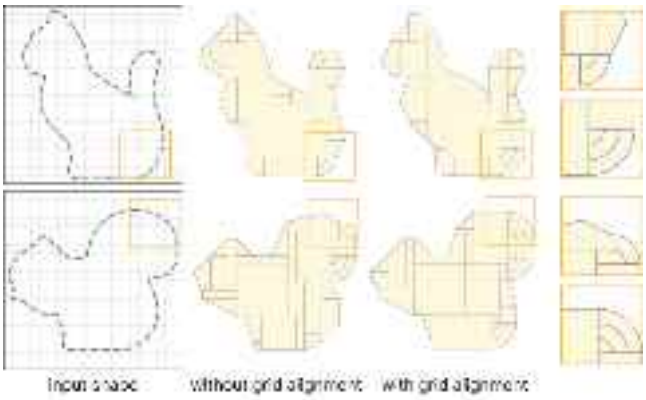


Fig. 8. Comparing LEGO® models generated with and without grid alignment. Grid alignment helps align the contours with the baseplate grid, such that the generated LEGO® models can better approximate the input shapes.

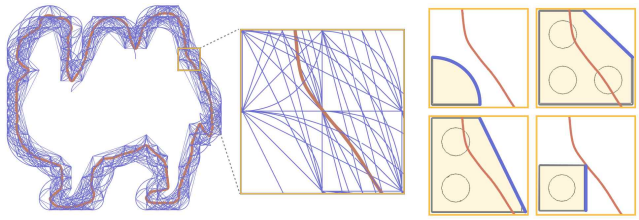


Fig. 9. Left: candidate edge segments enumerated around a grid-aligned contour. Right: we associate each edge segment with LEGO® tile(s) that may cover it. Candidate edge segments are marked in blue in this paper.

bricks. Also, LEGO® bricks have a fixed set of colors, known as the the LEGO® color palette. We pick the color in the palette that is the closest to the associated color in the input as the color of the bricks.

Further, we extend this pipeline to handle inputs with multi-color regions and multiple layers by formulating additional constraints in our optimization model; see Section 7. In the end, we employ our pipeline to produce diverse colorful LEGO® models and conduct various experiments to demonstrate the capability of our method, as well as its compelling quality and speed.

4 GRID ALIGNMENT

The baseplate has a grid structure, in which each grid cell encloses a single LEGO® stud; see Figure 4(c). Inspired by [Xu et al. 2010], we propose to first align the contour of the shape in the input image with the baseplate's grid structure, such that the generated LEGO® model can better approximate the shape. In detail, we have the following considerations: (i) we should try to move sharp points on the contour close to the grid vertices, such that we may better reproduce sharp points by LEGO® bricks; (ii) similarly, we should try to align nearly horizontal/vertical segments in the contour with horizontal/vertical grid lines in the baseplate; and (iii) the global and local deformations on the contour should be as small as possible to maintain the overall appearance of the shape.

Step (i): Preparation. To start, we extract the contour of the shape in the input image and locate nearly horizontal/vertical curve segments on the contour (contour segments) by thresholding; see Figure 7(a). To extract the contour, we densely sample points along the shape's boundary in the input image and employ Laplacian smoothing. Then, we locate points with sharp turning angle on the contour (see Figure 7(b)) and treat these sharp points and the endpoints of the identified horizontal and vertical contour segments as feature points on the contour. Further, for each feature point, we locate its associated target on the grid, *e.g.*, nearest grid vertex; see Figure 7(c).

Steps (ii) & (iii): Global & local alignment. We slightly translate and scale the contour by formulating a linear regression that minimizes the sum of distances between the feature points and the associated targets. Then, we slightly deform local segments of the contour between successive pairs of feature points, by solving for an affine transformation using gradient descent to move the feature points closer to the associated targets. For the case of endpoints from horizontal/vertical contour segments, we have to keep the endpoint pair to remain horizontal/vertical after the deformation.

Figure 7(d) presents two example shapes (contours) adjusted by the above procedure. From the results, we can see that the feature points in the adjusted contours can better align with the grid structure. Also, the adjusted contours are deformed only slightly to maintain the overall appearance, without obvious distortion. Figure 8 gives two examples on how grid alignment helps produce LEGO® models that better approximate the input shapes. We present the detailed mathematical formulation of the grid alignment step in Part A of the supplementary material.

5 CONTOUR APPROXIMATION

After obtaining the grid-aligned contour, we next should find a polygon on the baseplate that best approximates the contour (see Figure 5(ii)), while ensuring the polygon to be reproducible with the LEGO® bricks in the tile set. Our key idea to achieve this is to first enumerate all possible LEGO® brick placements around the contour and formulate a graph optimization to solve for the polygon.

Step (i): Enumerate candidate edge segments. Looking at the non-rectangular LEGO® bricks shown on the left side of Figure 6, we can see the limited types of sloping and rounding edges that LEGO® bricks may reproduce. So, we first enumerate *sloping and rounding edge segments* around the grid-aligned contour, then take these edge segments together with *one-unit grid line segments* around the contour to form a set of candidate edge segments; see Figure 9 (left). Also, we associate each of these edge segments with LEGO® brick(s) that may reproduce it; see Figure 9 (right) for some examples.

Step (ii): Graph Initialization. Next, we model the candidate edge segments and their relations by constructing graph $\mathcal{G} = \{\mathcal{V}, \mathcal{E}_1, \mathcal{E}_2\}$ with node set $\mathcal{V}$ and edge sets $\mathcal{E}_1$ and $\mathcal{E}_2$. Denoting $\{s_i\}$ as the set of candidate edge segments from step (i), we define $\mathcal{V} = \{s_i\}$, so each node in $\mathcal{V}$ denotes a unique candidate edge segment. Further, we employ $\mathcal{E}_1$ to represent connection relations and $\mathcal{E}_2$ to represent conflicting relations among the candidate edge segments:

- we create *connecting edge* $(s_i, s_j) \in \mathcal{E}_1$, if candidate edge segments s_i and s_j share a common endpoint at a grid vertex; see an example in Figure 10 (a); and
- we create *conflicting edge* $(s_i, s_j) \in \mathcal{E}_2$, if we cannot arrange LEGO® tiles to simultaneously cover both s_i and s_j without tile overlap; see examples in Figure 10 (b & c).

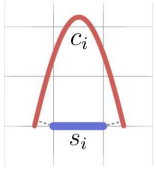
Using graph $\mathcal{G}$, we can formulate the problem of finding the contour polygon into a graph optimization. That is, we aim to (i) find a cycle through edges in $\mathcal{E}_1$, such that the series of edge segments (nodes) in the cycle best approximates the input shape, while (ii) employing $\mathcal{E}_2$ to ensure overlap-free LEGO® building, *i.e.*, no two nodes in the cycle are connected by a conflicting edge in $\mathcal{E}_2$. Please see supplementary material Part B for the details on graph initialization.

Step (iii): Compute graph node and edge attributes. We compute node and edge attributes for supporting the graph optimization. For each graph node (candidate edge segment) $s_i \in \mathcal{V}$, we compute:

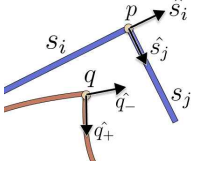
(i) *Distance deviation* L_d between s_i and the grid-aligned contour by projecting s_i onto the contour, finding the associated nearest contour segment c_i, and computing the Chamfer distance between points uniformly sampled on c_i and s_i.

(ii) *Distance variation* L_v measures the standard deviation of point-wise distances between uniformly-sampled points along s_i and contour segment c_i; when the distance variation is small, it means s_i has a similar shape as c_i, and vice versa.

Further, we prune candidate edge segments with low capability of approximating the contour by comparing lengths $|s_i|$ and $|c_i|$, *i.e.*, when $|c_i|/|s_i| >$ threshold δ_1; see the right inset figure. Besides, we compute the following attribute for each connecting edge $(s_i, s_j) \in \mathcal{E}_1$:



(iii) *Sharpness matching* L_s measures how well the turning angle between candidate edge segments s_i and s_j matches the local sharpness on contour. Here, we denote point p as the endpoint between s_i and s_j and point q as the projection of p on the contour. At point p, we obtain normalized tangential vectors $\hat{s_i}$ and $\hat{s_j}$ along the two candidate edge segments, whereas at point q, we fit its left and right contour segments separately with a B-spline curve to obtain normalized tangential vectors $\hat{q_-}$ and $\hat{q_+}$; see the above inset figure. Then, we can compute the *sharpness matching* term as the sum of deviations between $\hat{s_i}$ and $\hat{q_-}$ and between $\hat{s_j}$ and $\hat{q_+}$. Further, we prune (s_i, s_j) from $\mathcal{E}_1$, if its sharpness matching value exceeds threshold δ_2. Using this term, we can help preserve both sharp corners and smooth boundaries on the contour of the input.



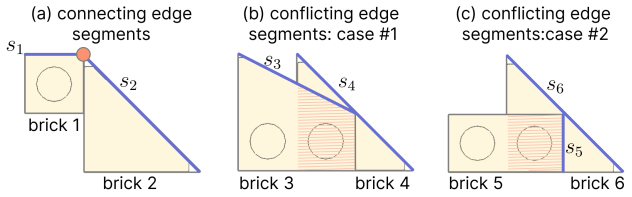


Fig. 10. (a) Connecting edges in $\mathcal{E}_1$ are formed between candidate edge segments (blue) that share a common endpoint (red dot); see, *e.g.*, (s_1, s_2). (b,c) Conflicting edges in $\mathcal{E}_2$ mark candidate edge segments (blue) that cannot be simultaneously chosen, since their associated LEGO® bricks overlap one another; see *e.g.*, (s_3, s_4) in (b) and (s_5, s_6) in (c).

Please refer to supplementary material Part C for the formal mathematical definitions of each of the above attributes.

Step (iv): Solve for the contour polygon. Now, we are ready to formulate the graph optimization. First, we denote $\mathcal{V}' \subseteq \mathcal{V}$ as an ordered list of vertices (candidate edge segments). Second, we define two constraints: (i) to form a cycle, every pair of successive vertices, including the first and the last, in $\mathcal{V}'$ should be a connecting edge in set $\mathcal{E}_1$; and (ii) to ensure overlap-free LEGO® building, no two vertices in $\mathcal{V}'$ should be contained in the conflicting edge set $\mathcal{E}_2$. Third, we formulate an objective function, $\min_{\mathcal{V}'} [w_d\mathcal{L}_d + w_v\mathcal{L}_v + w_s\mathcal{L}_s + w_m\mathcal{L}_m]$, as a weighted sum of the following four terms:

- *Distance deviation* $\mathcal{L}_d$ sums the distance deviation L_d of each $s_i \in \mathcal{V}'$ and normalize the sum by the total contour length.
- *Distance variation* $\mathcal{L}_v$ sums the distance variation L_v of each $s_i \in \mathcal{V}'$ and normalize the sum by the total contour length.
- *Sharp-points preservation* $\mathcal{L}_s$ measures how well the polygon represented by $\mathcal{V}'$ retains the sharp points detected on the contour; see Section 4. For each sharp point on contour, we find a pair of adjacent candidate edge segments with the lowest sharpness matching value L_s; a lower value means a better preservation of the sharp point. Here, $\mathcal{L}_s$ is the mean sharpness matching value over all detected sharp points.
- *Smoothness preservation* $\mathcal{L}_m$. To avoid undesired sharp points on the generated LEGO® model that do not exist in the input contour, we visit every connecting point between successive edge segments in $\mathcal{V}'$, sum its sharpness matching value L_s, and normalize the sum by the total contour length.

Note that we present the detailed formal mathematical definition of each term in supplementary material Part D.

To solve the above graph optimization for the contour polygon, we first define a binary decision variable x_i for each graph node $s_i \in \mathcal{V}'$ and denote vector $\mathbf{x}$ as $\{x_i\}$. Then, we can model the

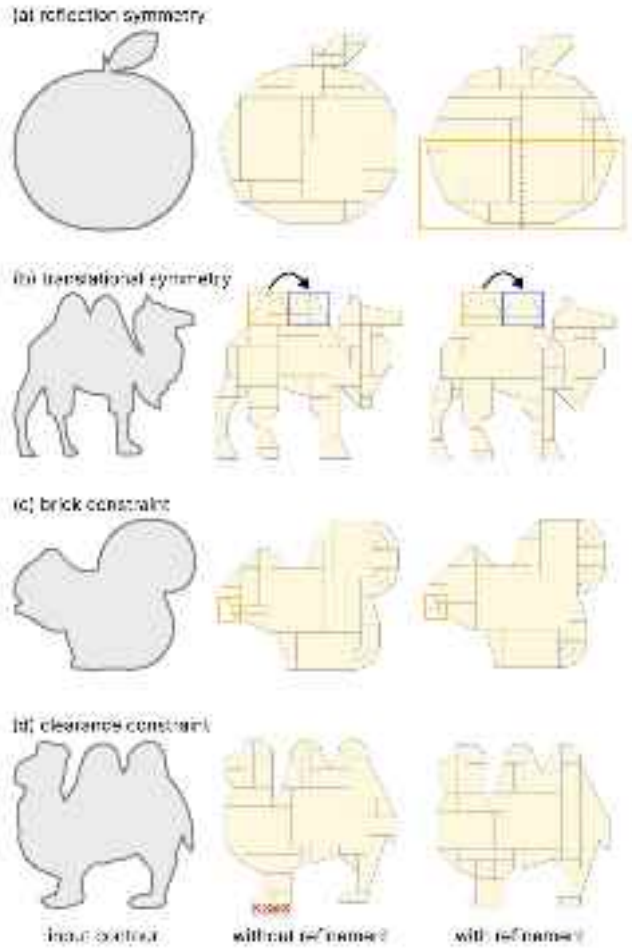


Fig. 11. Optional perceptual constraints for refining the contour polygon. The models from top to bottom are Apple, Camel2, Squirrel, and Camel1.

connection constraint and the conflicting constraint as a function of $\mathbf{x}$, and constrain $\mathcal{V}' \neq \emptyset$ to avoid trivial solutions. Further, we can compute the objective function based on $\mathbf{x}$. Therefore, we can formulate a constrained optimization using integer programming (IP) and solve it using an IP optimizer. Details about the formulation can be found in supplementary material Part D.

6 PERCEPTUAL CONSTRAINTS

Optionally, perceptual constraints can be introduced into the optimization model to help refine the results (see Figure 5(iii)):

Symmetry constraint. We may specify *reflection symmetry* by marking an axis-aligned box with a symmetry x-/y-axis (Figure 11(a)) and *translational symmetry* by marking a pair of boxed regions (Figure 11(b)). To achieve these, we formulate additional constraints in our optimization model to enforce the co-selection of associated candidate edge segments. Then, our optimization model can solve for the refined contour polygon with the symmetry property.

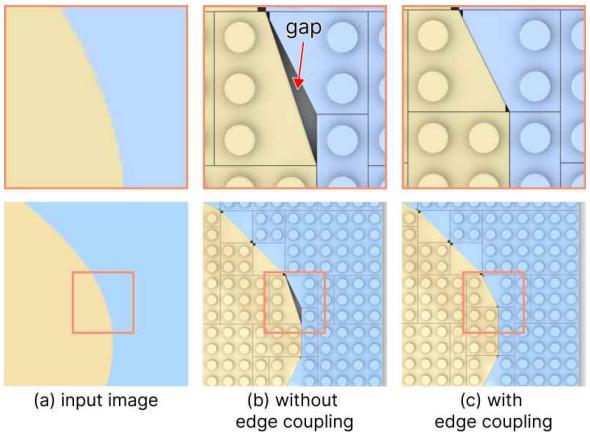


Fig. 12. Our edge coupling strategy helps to ensure a coherent boundary between LEGO® bricks in adjacent regions; compare (b) vs. (c).

Brick constraint. On a low-resolution baseplate, small but salient shape features could be missed in the contour polygon. We may explicitly arrange some brick(s) on the baseplate to make up certain desired features in the contour polygon (Figure 11(c)). Once a brick is added, decision variables associated with the brick's edge segments would be flagged (selected) to constrain the optimization.

Clearance constraint. Conversely, we may keep certain grid cells unoccupied; see Figure 11(d) for an example. In detail, we achieve this effect by constraining the non-existence of the candidate edge segments associated with the bricks that may occupy the cells.

7 EXTENSIONS

After obtaining the polygon that approximates the input, we next follow the procedure in step (iv) of Section 3 to fill the polygon with LEGO® bricks to produce a LEGO® model; see Figure 5(iv) and supplementary material Part F.

So far, we present our pipeline for the case of a single *simply-connected* shape in the input. Below, we present our extensions to enable the pipeline to produce LEGO® models with (i) regions of different colors and (ii) regions over multiple layers.

Extension (i): Multiple regions of different colors. To achieve this extension, the challenge is to ensure a coherent boundary between adjacent regions. As Figure 12(b) shows, if we arrange LEGO® bricks independently for the two regions, their boundaries may not match and may even overlap. Hence, we propose to couple the associated candidate edge segments in adjacent regions and solve for candidate edge segments in multiple regions together by jointly performing the optimizations; see Figure 12(c) for an example result.

In detail, given an input image with multiple regions (see Figure 12(a)), we first segment the regions, extract their contours, and identify shared contour segments between each pair of adjacent regions. Then, we amend our pipeline: (i) co-adjust the feature points and contours shared between regions in the global and local alignment; (ii) impose additional constraints in the graph optimization to couple the matched candidate edge segments along the shared

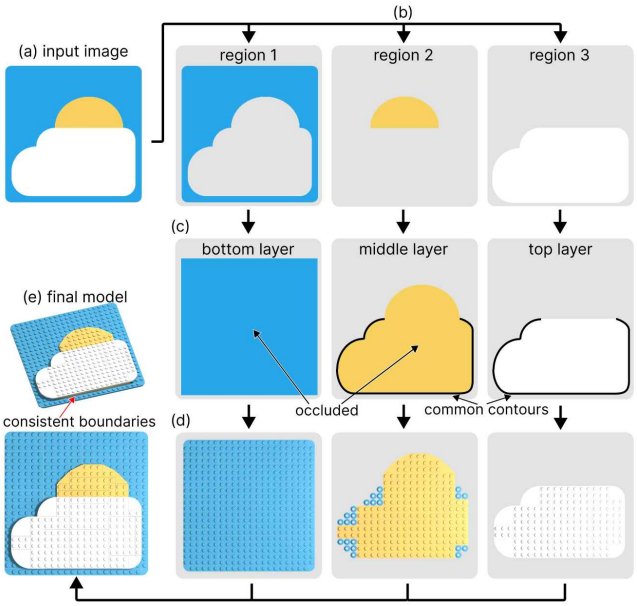


Fig. 13. A running example of creating a multi-layer LEGO® model from (a) an input image. (b) Segmented regions. (c) User-assigned layers. (d) Models generated in each layer. (e) Final composed LEGO® model.

boundary, such that the matched candidate edge segments are always co-selected (see Figure 12(c)); (iii) identify conflicting candidate edge segments across regions to ensure overlap-free LEGO® bricks in the results; and (iv) jointly solve for the contour polygons of multiple regions, such that the above cross-region constraints can be enforced by the graph optimization.

Extension (ii): Multiple layers. Further, we can extend our pipeline for producing LEGO® sketch models composed of multiple layers; see, *e.g.*, Figure 13(e). Such a model exhibits depth perception effect. Figure 13 shows a running example to illustrate the procedure:

- Following the procedure as extension (i), we first segment the input image (Figure 13(a)) into regions (Figure 13(b)).
- Then, to create a multi-layer model, users can assign each region to a different layer (Figure 13(c)). In the future, we plan to study visual perception techniques to explore T-junctions and machine learning to generate layering automatically.
- Importantly, given the layering, we examine regions from top to bottom layers to expand each region to fill the occluded areas under the upper layer (Figure 13(c)), thus promoting brick connections and supports in the final model. Also, we identify common contours in regions of adjacent layers (Figure 13(c)) and add constraints in the optimization to ensure consistent boundaries between layers.
- After that, we solve the graph optimizations layer by layer (top to bottom) to generate the contour polygons, produce the LEGO® model in each layer (Figure 13(d)) and stack them together (Figure 13(d) & (e)).
- In the tile set, some LEGO® pieces do not have studs on top; see Figure 6. So, when these bricks are arranged in a middle layer, they cannot provide connections for the bricks above them. Therefore, we replace such bricks with 1×1 round plates (whose colors follow the lower-layer bricks); see, *e.g.*, the second sub-image from the right of Figure 13(d).

Table 1. Statistics of our results: (i) number of nodes (candidate edge segments), connecting edges, and conflicting edges in graph, and baseplate resolution; (ii) number of edge segments in contour polygon and number of LEGO® bricks; and (iii) our method's running time. Models are sorted in ascending order of solving time. Multi-color models are highlighted in yellow. Models without figure numbers can be found in Figures 14 or 17.

	graph size & baseplate resolution				generated model size		running time (sec.)	
model	$\lvert\mathcal{V}\rvert$	$\lvert\mathcal{E}_1\rvert$	$\lvert\mathcal{E}_2\rvert$	res.	#polygon edges	#bricks	graph construction	solving
Fish (Fig. 4)	301	2353	21026	8x8	12	7	2.92	1.02
Cat (Fig. 8)	1020	7930	55842	20x20	38	36	13.57	3.03
Apple (Fig. 11)	936	7079	48981	20x20	36	31	12.70	3.29
Squirrel (Fig. 8)	997	7860	55433	20x20	34	32	11.72	3.43
Cattle	953	6482	51049	20x20	51	26	13.50	3.43
Cat (Fig. 3)	792	6166	45960	16x16	29	25	8.92	3.49
Camel2 (Fig. 11)	1305	8723	64705	20x20	80	44	19.61	4.04
Deer	984	6656	46914	20x20	61	34	13.60	4.34
Spring	1903	12350	100627	32x32	113	67	39.15	5.15
Bunny	1178	8892	63349	20x20	49	39	13.84	5.23
Sky (Fig. 13)	2170	14490	114477	24x24	169	81	25.59	5.57
Horse	1640	10877	75592	32x32	84	58	33.10	5.80
Kangaroo	1473	10661	73490	32x32	63	49	27.30	6.70
Dog	1060	8010	57087	20x20	44	29	12.54	6.91
Helicopter	1550	9663	68604	32x32	149	52	31.48	7.22
Camel1 (Fig. 5)	1428	10999	82916	20x20	62	43	19.85	7.59
Dolphin	975	7703	53012	20x20	40	28	11.66	9.02
Bat	787	6334	46149	20x20	28	28	9.64	9.04
Beetle	2869	21357	196377	32x32	132	55	50.53	10.18
Dragon	2682	19420	139986	32x32	127	89	63.28	11.31
Android logo	1478	10392	68359	32x32	82	52	26.15	14.34
Butterfly	3784	43728	463806	32x32	57	49	41.67	20.44
Ironman (Fig. 18)	8020	62835	425076	64x64	507	293	197.01	20.78
Landscape	13294	94121	655143	64x64	664	465	293.15	32.26
Sydney opera (Fig. 18)	11137	75644	671541	64x64	728	349	324.03	34.74
Octopus	3556	26397	213526	32x32	156	86	86.81	40.14
Australia terrain	9742	75560	501828	64x64	304	444	212.76	75.45
Smiley face	9793	73279	529716	64x64	472	301	207.82	99.23
SIGGRAPH	8554	62588	541060	120x30	359	191	101.68	121.51
Rolling Stones (Fig. 1)	11283	87701	837818	64x64	331	367	150.32	129.75
Brickhead	7978	58173	424726	64x64	619	386	215.69	268.80
Australia map	12136	120549	1544256	64x64	302	196	181.90	756.39

More results on single- and multi-layer LEGO® sketch models produced by our method are presented in Section 8.

8 RESULTS AND EXPERIMENTS

8.1 Implementation Details

We implement our tool using Python 3.10 and employ Numpy [Harris et al. 2020] to manipulate arrays. We employ Shapely [Gillies et al. 2023] for planar geometric computation, and Gurobi [Gurobi Optimization, LLC 2023] for solving the graph optimization.

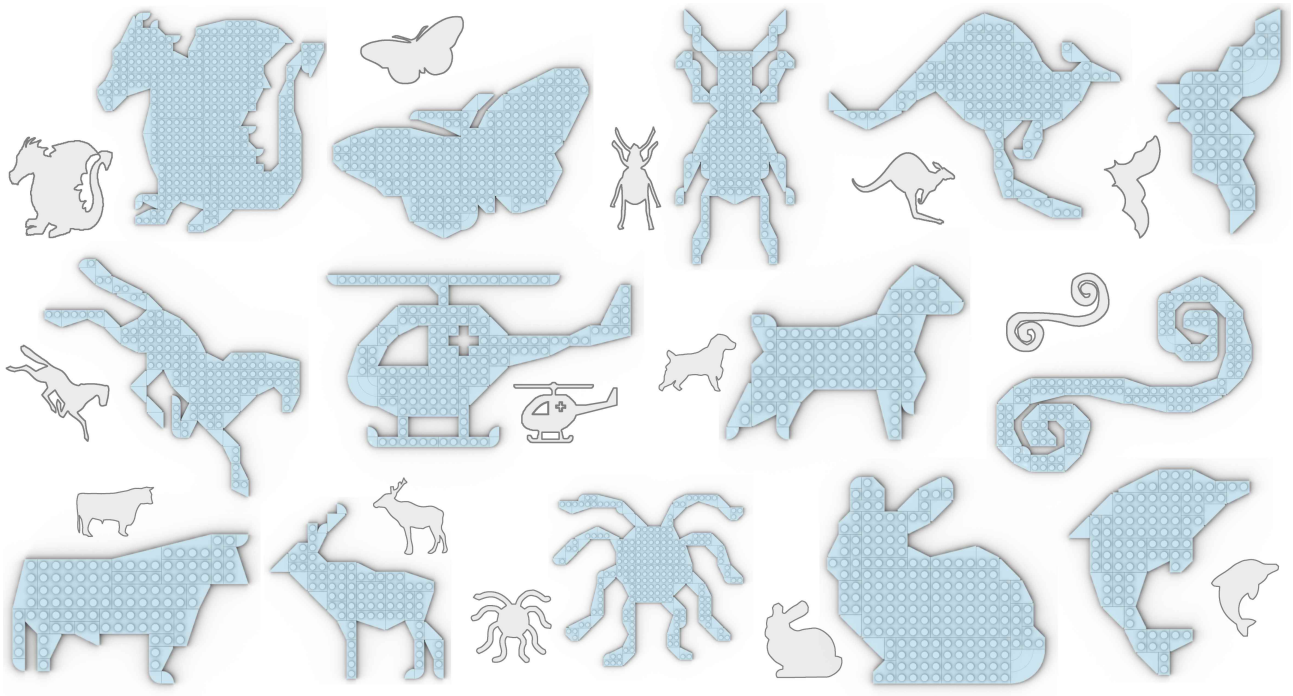


Fig. 14. A gallery showcasing various LEGO® models generated by our approach on 14 different input images. From left to right and then top to bottom, the models are Dragon, Butterfly, Beetle, Kangaroo, Bat, Horse, Helicopter, Dog, Spring, Cattle, Deer, Octopus, Bunny, and Dolphin.

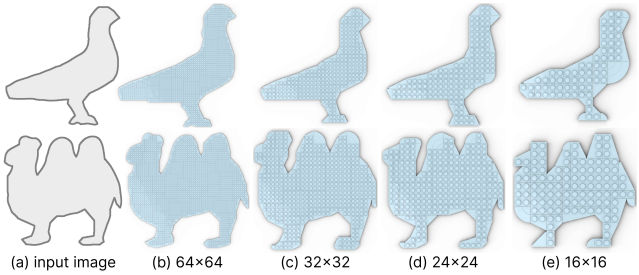


Fig. 15. Bird and Camel1 in different resolutions. From input (a), we generate LEGO® models on baseplates of resolution from 64×64 (b) to 16×16 (e). All models are automatically generated without perceptual constraints.

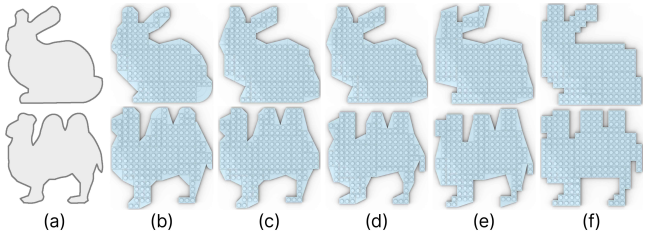


Fig. 16. Results generated on Bunny and Camel1 using the full tile set shown in Figure 6 with 13 non-rectangular bricks (b) to using a reduced set with 10 (c), 6 (d), 3 (e), and zero (f) non-rectangular bricks.

We adopt the KNN algorithm using OpenCV [Bradski 2000] to extract shapes and contours in input images, and remove high-frequency noises via Laplacian smoothing for 20 iterations. Then, the extracted contours are resized to fit into the baseplate coordinate system. We compute a truncated distance field for each contour to accelerate the identification of candidate edge segments, as elaborated in Section 5. To facilitate the computation of the objective functions in Step (iv) of Section 5, we uniformly sample points along the boundary of the extracted contour and the candidate edges every d LEGO® units, where we empirically set d as $\frac{1}{20}$ to balance between the precision and running time. In addition, we empirically set thresholds δ_1 and δ_2 in Section 5 as 3 and 4, respectively.

8.2 Results

We employ our method to create a large number of LEGO® models of varying sizes and complexities. Table 1 lists their statistics. Figure 14 presents a gallery of single-region models that have not been shown earlier in the paper. Among them, we impose reflection symmetry on Beetle and local reflection symmetry on the skid landing gear part of Helicopter. The other models are generated from the inputs without perceptual constraints. These results demonstrate the capability of our method to produce LEGO® models that resemble the inputs, while striving to preserve the smooth curves and sharp features in the inputs. It is worth to note that in many models, our method has striven to best use the available bricks with sloping and rounding edges to mimic various sharp, curvy, and complicated structures in the inputs. Notably, it attempts to replicate thin and

Fig. 17. A gallery of seven multi-color LEGO® sketch models, alongside with their input images. From left to right and then top to bottom: Australia map, Android logo, Brickhead, Siggraph, Australia terrain, Smiley face, and Landscape.

Fig. 18. Physical assembles of Dragon, Ironman, and Sydney opera.

sharp structures in various models, such as in Octopus and Beetle, as well as the challenging spirals in Spring. Also, it attempts to preserve the detailed fine features on various models, as evident on the antler of Deer and around the contour of Bat.

On the other hand, we employ our method to create eleven multi-color multi-layer LEGO® models; see Figures 1, 13, 17, and 18. Figure 17 shows seven of them, and Figure 18 shows a photograph of three physical assemblies that are being hung on the wall. These results showcase the remarkable versatility of our tool in producing different kinds of images, covering clip arts, movie characters, maps, logos, iconic buildings, and landscape painting, and demonstrate the practicality as home or office decor. Particularly, note the seamless borders between regions of different colors, showing that our extended optimization model can help enforce the connections not only within individual regions but also across adjacent regions.

Timing performance. The rightmost part of Table 1 reports the running times of our method: about ten seconds for models of standard LEGO® set sizes (20×20 or less), thanks to the *contour approximation* and *graph optimization*. The process for constructing the graph typically takes up most of the running time, as it needs to exhaustively enumerate all potential brick placements and all candidate edge segments, as well as their connections and conflicts. Overall, the whole method is able to complete in less than two minutes for standard-size LEGO® models, and it takes only a few minutes for the larger models like Rolling Stones and Landscape.

Results on varying resolutions. Our method is able to generate LEGO® models in different resolutions. Figure 15 shows the results on Bird and Camel1, where we progressively reduce the baseplate resolution from 64×64 to 16×16. We can see that as the resolution decreases, our method strives to use sloping and rounding pieces to keep the overall perceptual resemblance but may simply skip the intricate shape details, such as the beak of Bird.

Results with varying brick sets. We study our method's performance on different brick sets. We start with the full tile set shown in Figure 6 to generate Bunny and Camel1, and progressively remove non-rectangular bricks from the tile set, regenerate the model, and repeat this process until only rectangular bricks left. From

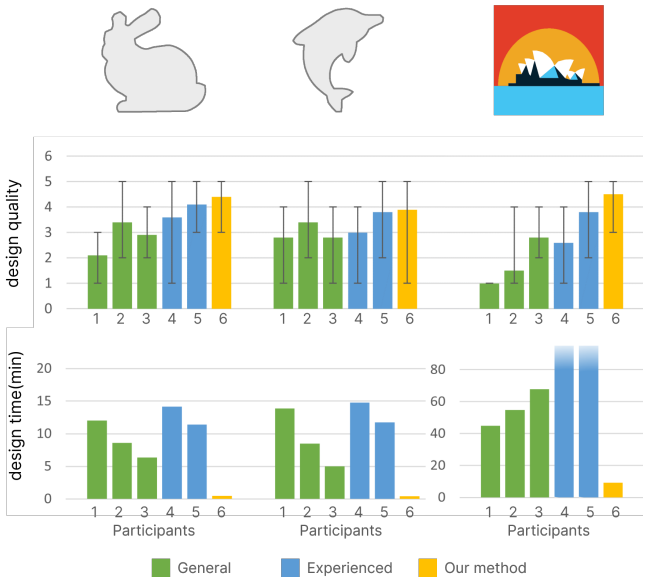


Fig. 19. Exploring human performance. We recruited five participants to design LEGO® models for the three images shown on top and compared their performance with our method through a user study. Results show that our method can rival the experienced designers, while using much less time.

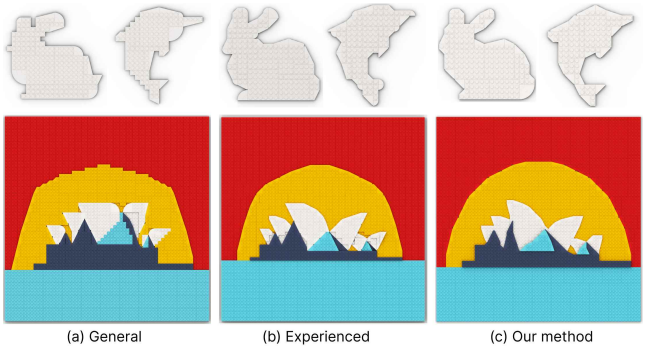


Fig. 20. Renderings of the designs created by (a) a general LEGO® fan, (b) an experienced designer, and (c) our method.

the results shown in Figure 16, we can see that our method can still produce results that perceptually resemble the original inputs when using different tile sets. Also, we can see that our method can skillfully realize the shapes using different bricks. Besides, these result demonstrates the crucial role of non-rectangular bricks in maintaining smooth curves and sharp features.

8.3 Evaluations

Comparison with human performance. To obtain a sense of how human performs in LEGO® sketch design, we recruited five participants and asked them to design LEGO® sketch models for the same input images as our method. Two of them are experienced LEGO® designers with around two years of full-time working experience on LEGO® design, while the other three are general LEGO® fans. Figure 19 (top) shows the three input images employed in this experiment and the baseplate resolution for them are 20×20 for Bunny, 20×20 for Dolphin, and 64×64 for Sydney Opera.

Instead of physical assembly, we let our participants create the designs in a commercial software (which is Stud.io [Bri 2023]) to accelerate the brick retrieval, brick manipulation, and brick coloring. Before the experiment starts, we respectively prepare three scene files (with .io extension) for the three input images, where we put into each scene (i) the baseplate and (ii) the available bricks (i.e., the brick set shown in Figure 6) next to the baseplate, such that the participants can readily copy and paste the bricks and drag the bricks to quickly form a design. We also put the associated image as texture on the virtual baseplate, such that the participants can easily evaluate the design similarity and the design dimension. After a basic training to ensure the LEGO® fans learn how to use the software, we instruct the participants to try their best to create a design that resembles the input using only the given brick set. Also, we set a time limit of 20 minutes for the simpler Bunny and Dolphin and 120 minutes for Sydney opera. We collected the designs of each participant and recorded the time taken to create each design.

To evaluate the quality of the designs, we recruited ten general-background volunteers to score the designs by the participants and by our method via questionnaires. We show an image of each design on the questionnaire. The volunteers were asked to rate the overall perceptual resemblance and visual aesthetics on a one-to-five scale. Figure 19 plots the design quality scores (middle) and time taken (bottom) for all the 18 designs, with participant IDs marked below the bar charts. Figure 20 shows the renderings of the designs by participants No.2 and No.5. From the results, we can see that our method is able to produce LEGO® models that rival the experienced human designers, while using significantly less time. The models designed by general LEGO® fans typically exhibit zigzag effects and unregulated shape distortions. While the experienced designers may produce models with smooth boundaries, they often require more time to try to minimize the distance deviations from the input images. Our results, in contrast, possess smooth boundaries, sharp corners, minimized distance deviations, and less distortions.

Comparison with baseline methods. We compare our method with three baseline methods. The first is a Greedy method that iteratively selects brick pieces around the shape boundary, where we minimize our objective function in each selection. The second method replaces the objectives in Section 5 by the metric of *integral square error* (Ise). The Ise metric is widely used in the research community of *polygonal approximation*, with which we simply measure and minimize the *distance deviation* between the input contour and the contour polygon. The third one adopts a Tiling algorithm that regards LEGO® pieces as tiles and aims to maximally cover a 2D region while avoiding holes and tile overlaps. This algorithm is a modification of the public code of [Xu et al. 2020].

We conduct the comparison by employing the baseline methods and our method to create LEGO® designs for 20 simply-connected input shapes, yielding a total of 80 results, where the resolution of the baseplates are all 20×20. To quantify the design quality

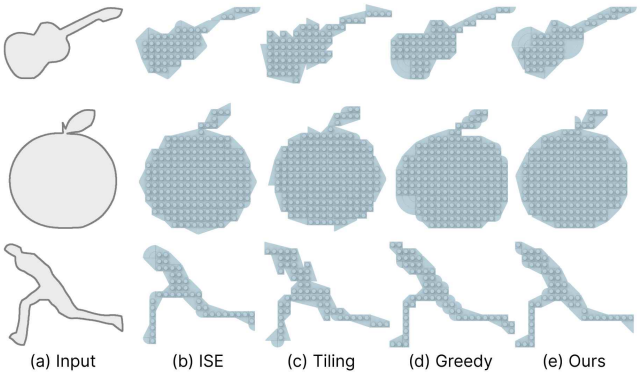


Fig. 21. Visual comparison between LEGO® models generated by baseline methods (b)-(d) and our method (e). All resolutions are 20×20 .

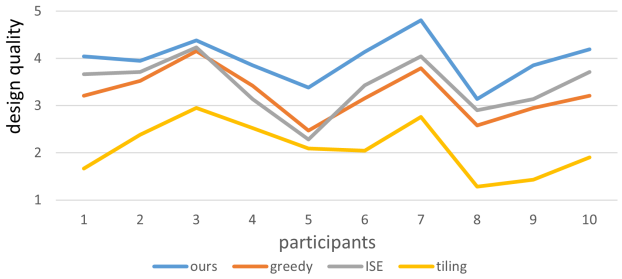


Fig. 22. User study results on comparing our method with the three baseline methods. The line charts show the average design quality scores on the four methods assessed by the ten participants, revealing that our method consistently produces better results than the baseline methods.

of these models, we evaluate the results using the following four metrics: *distance deviation*, *distance variation*, *sharpness matching*, and *gradient*. The metrics of *distance deviation*, *distance variation*, *sharpness matching* are directly adopted from our objective function (Section 5), and the *gradient* metric is defined as the mean difference between tangential vectors measured on associated sample point on the input contour and the approximated polygon.

We recruited 10 participants of general background to conduct the questionnaires, using the same format as the human performance experiment, collecting 80 scores (20 models x 4 methods) from each participant. Figure 22 plots the average scores, while Figure 21 shows the rendering of 12 results on three inputs. We also report in Table 2 the running time and the metric values. From the scores provided by the participants and the visual results, we can see that our approach clearly outperforms the baselines. Besides, the line charts also reveal that the participants consistently prefer our results over others. Please also refer to supplementary material Part I for the line charts with error bars.

Specifically, while the Greedy algorithm is fast, it lacks global consideration when arranging each brick. The greedy approach iteratively selects bricks that best match the local shapes, but it will likely cause unpleasant effects elsewhere, including non-smooth

Table 2. Running time and perceptual metrics measured on the LEGO® sketch models generated by our method and by the three baselines. Note that the Ise method solely minimizes the *distance deviation* (so it is a lower bound of this metric), while ignoring all other perceptual considerations. Results generated by our method perform the best on all the other metrics. See also Figure 21 for the visual comparisons.

Model	Method	Running time (seconds) ↓	Distance variation ↓	Sharpness matching ↓	Gradient ↓	Distance deviation ↓
Guitar	ise	9.27	0.65	0.08	0.24	**0.04**
	tiling	54.09	1.76	0.14	0.51	0.08
	greedy	**0.01**	0.62	0.06	0.25	0.08
	ours	13.53	**0.25**	**0.02**	**0.16**	0.06
Apple	ise	12.59	0.66	0.07	0.32	**0.04**
	tiling	471.71	0.89	0.14	0.35	0.05
	greedy	**0.04**	0.40	0.13	0.25	0.06
	ours	14.58	**0.26**	**0.04**	**0.19**	0.05
Running human	ise	11.27	0.44	0.32	0.23	**0.03**
	tiling	21.24	20.58	0.46	0.46	0.13
	greedy	**0.01**	0.43	0.07	0.21	0.03
	ours	14.78	**0.33**	**0.05**	**0.15**	0.04

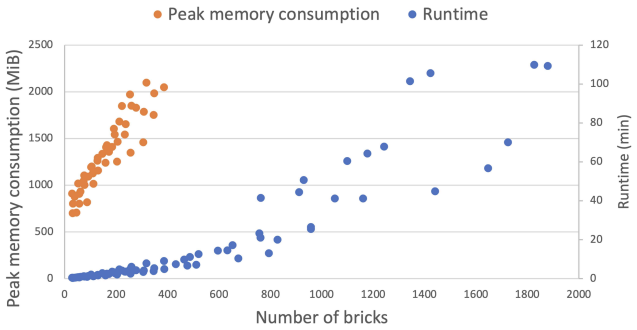


Fig. 23. Runtime and memory consumption analysis. These blue scatterplot shows the runtime and the orange scatterplot shows the peak memory consumption of our method when we employ it to generate LEGO® designs with different number of bricks.

zigzags and over-dilated thin features, as shown in Figure 21 (d). The Ise approach, commonly employed for *polygonal approximation* and typically effective for general tasks such as *polygonal simplification*, yields unsatisfactory results in our task. This is because designing LEGO® sketch models necessitates multi-faceted visual perception considerations, such as boundary smoothness and sharp features, not merely distance deviation. The results of Ise also demonstrate the necessity of the objective terms $\mathcal{L}_v$, $\mathcal{L}_s$, and $\mathcal{L}_m$ presented in Section 5. Lastly, the tiling algorithm is the most time-consuming one, while getting the lowest quality scores. The global consideration for the entire design slows down the computation and the unpleasing visual effect is due to the constraint that enforces each brick not to extrude beyond the shape region.

Runtime and Memory Consumption Analysis. We analyzed the runtime and memory consumption of our method on five shapes

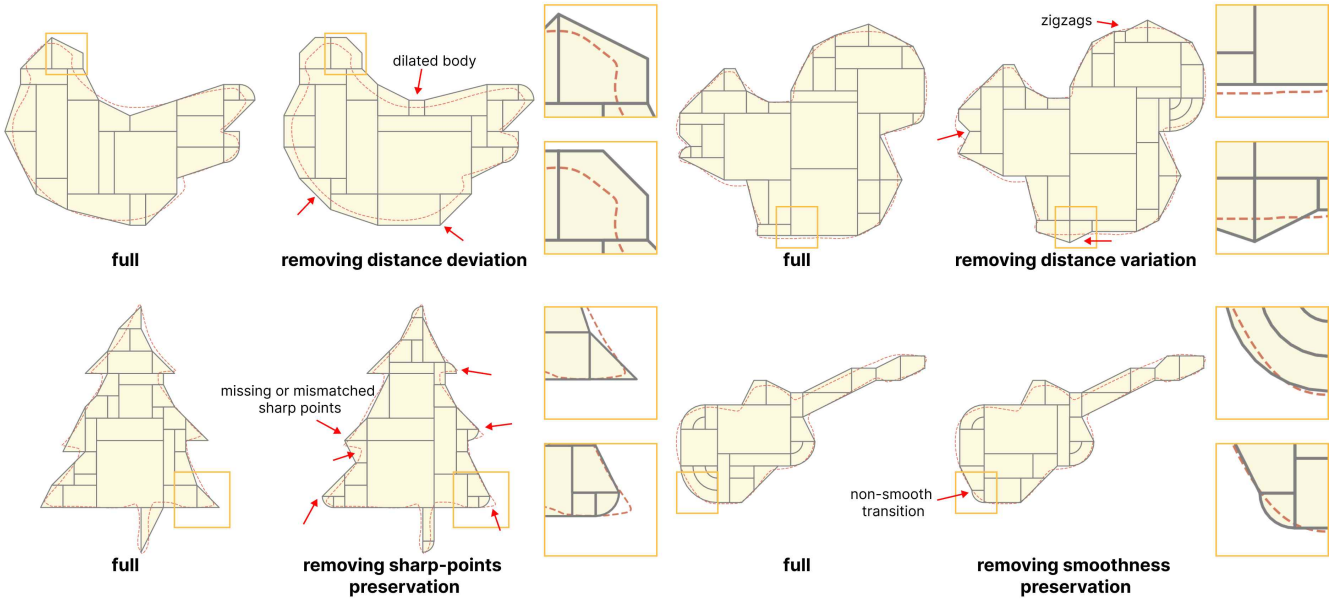


Fig. 24. Visual ablation results. The red dashed line in each figure reveal the input contours to aid the comparison. For each case of removing a term, the generated model degrades, compared with the result produced by the full objective. In particular, the degradation includes dilation, zigzag, missing sharp corners, and unwanted non-smooth transitions. From left to right then top to bottom, the models employed are Banana, Squirrel, Tree, and Guitar.

for 18 increasing resolutions, ranging from 20×20 to 280×280. The plots shown in Figure 23 revealed that both the runtime and peak memory consumption increase roughly exponentially with the number of bricks. This can be attributed to our choice of an IP solver that predominantly employs a branch-and-bound algorithm. Such an algorithm typically exhibits exponential growth, concerning the number of variables and constraints.

To mitigate the computational complexity, our method focuses more computational resources on processing the exterior bricks (elaborated in Section 5). Consequently, the growth in the number of variables is approximately linear in relation to the length of the input contours, rather than the areas of the color regions. As presented in Figure 23, our method can generate results within four hundred bricks in a few minutes, which corresponds to a resolution of around 64×64. However, the runtime significantly increases when processing a LEGO® sketch containing six hundred bricks. Yet, as shown in the plot, our method is able to produce LEGO® sketches of up to 280×280 resolution within two hours.

Ablation Study. We further explore the influence of the four objective terms in our optimization by conducting an ablation study. This study comprised 120 different inputs: 20 simply-connected shapes, each at 6 resolutions (from 16×16 to 26×26). For each input, we generated five LEGO® sketch models: one using our objective with all terms and four with each of the terms is removed from the full objective function. We adopted the same four metrics from the *comparison with baseline methods* experiment to evaluate every model. Table 3 shows the average metric values of all models under different experimental settings, where we can see that removing a specific term typically degrades its corresponding metric without substantially benefiting the other metrics.

Table 3. Ablating each objective term. Each row shows an average metric score under five configurations: (i) all objective terms included, (ii) without distance deviation, (iii) without distance variation, (iv) without sharp-point preservation, and (v) without smoothness preservation. In each row, we highlight the score with the largest rise (*i.e.*, degradation of the associated metric) in red to reveal the impact of excluding the associated term.

Metrics	full	Without distance deviation	Without distance variation	Without sharp-points preservation	Without smoothness preservation
distance deviation ↓	0.052	0.084	0.049	0.049	0.050
distance variation ↓	0.396	0.361	0.513	0.369	0.397
sharpness matching ↓	0.037	0.033	0.035	0.116	0.040
gradient ↓	0.187	0.172	0.211	0.180	0.189

Figure 24 provide a visual comparison on some of the results, showing that removing the term *distance deviation*, *distance variation*, *sharp-points preservation*, *smoothness preservation* leads to the issue of over-dilation, zigzags, missing sharp corners, and abrupt non-smooth boundary transitions, respectively. See supplementary material Part J for more visual comparison.

9 CONCLUSION, LIMITATIONS, AND FUTURE WORKS

This paper presents the first computational system to aid the design of LEGO® sketch models. Altogether, there are three contributions

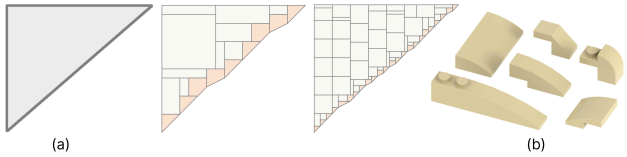


Fig. 25. Limitations. (a) A long and straight line segment whose orientation does not match the sloping orientations in existing LEGO® bricks. (b) Example cross-layer LEGO® bricks.

in this work. First is a computational framework to aid the design and creation of LEGO® sketch models. Particularly, this framework is able to automatically select and arrange LEGO® pieces to produce coherent results that resemble the simple input images. Second is our meticulous and comprehensive modeling of the representational capability of LEGO® plates and tiles, the formulation of the LEGO® sketch construction into a graph optimization, and the design of the objective function to quantify various construction requirements. Third, we extend our optimization formulation to handle regions of different colors and regions across multiple layers and propose to enhance our results by a grid aligning process and by various perceptual constraints to refine the results. In the end, we employed our method to construct LEGO® sketch models of various resolutions and complexities, physically built some of them, and compared our method with three baseline methods, general users, and experienced designers on the quality of the results and construction speed.

Limitations. First, small and thin structures, around or smaller than the size of a LEGO® unit, may not be well approximated by LEGO®. Second, though our method strives to best approximate the input, due to the limited representation capabilities of LEGO® bricks (*i.e.*, limited sloping orientations), it is hard to perfectly reproduce any contour, especially long and straight lines that do not match the sloping-edge orientations; see Figure 25 (a). Third, while stacking LEGO® plates over multiple layers exhibits genuine depth perception, our current approach does not consider cross-layer pieces (see, *e.g.*, Figure 25(b)), which in fact can help produce intriguing 3D sloping effects, as in the official LEGO® designs.

Future works. Currently, our method relies on users to provide layering information and perceptual constraints. We are interested in exploring machine learning methods to help generate the layering information and recognize perceptual constraints such as partial symmetry and small but salient features. Second, our current approach creates the LEGO® constructions layer by layer. Given that there are LEGO® bricks that go across multiple layers, we would like to generalize our approach to consider these bricks to further improve the perceptual quality of our results. Last, we would like to provide user interactions for one to interactively modify the input and get real-time feedback by online optimization.

ACKNOWLEDGMENTS

We thank all the anonymous reviewers for their comments and feedback, LEGO® designers for designing sketch models to be compared with our results, and participants for evaluating both our results and the human designed models. We thank Ms. Ding, Zihan for helping with the physical assembly of the LEGO® sketch model in Figure 1. The Tongue and Lips logo in Figure 1 is a trademark owned by Musidor B.V. Permission granted by Musidor B.V. The copyrights of the LEGO® products in Figure 2 are held by LEGO® Group. Permission granted by LEGO®. The input image of Android logo is built upon the Android Robot icon. The Android robot is reproduced or modified from work created and shared by Google and used according to terms described in the Creative Commons 3.0 Attribution License. The input image of Brickhead is built upon "lego"_(c) by Musket from Noun Project, which is licensed under CC BY 3.0. The input image of Australia terrain is built upon "Climate features of Australia - just roughly.svg"_(c) by Christoph Friedrich, which is licensed under CC BY-SA 3.0. The input image of Smiley face is built upon "718smiley.png"_(c) by East718, which is licensed under CC BY-SA 3.0. The input image of Sydney opera is hand-drawn by authors with the reference from https://dribbble.com/shots/2102533-Sydney-Opera-House. This work is supported by the Research Grants Council of the Hong Kong Special Administrative Region (Project no. CUHK 14201921).

REFERENCES

2023. BrickLink - Studio. https://www.bricklink.com/v2/build/studio.page
Nafiz Arica and Fatos T. Yarman Vural. 2003. BAS: A Perceptual Shape Descriptor Based on the Beam Angle Statistics. *Pattern Recognition Letters* 24, 9 (2003), 1627–1639.
Ilaruo Asada and Michael Brady. 1986. The Curvature Primal Sketch. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 8 (1986), 2–14.
Xiang Bai, Cong Rao, and Xinggang Wang. 2014. Shape Vocabulary: A Robust and Efficient Shape Representation for Shape Matching. *IEEE Transactions on Image Processing* 23, 9 (2014), 3935–3949.
Serge Belongie, Jitendra Malik, and Jan Puzicha. 2002. Shape Matching and Object Recognition Using Shape Contexts. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 24, 24 (2002), 14.
Gary Bradski. 2000. The OpenCV Library. *Dr. Dobb's Journal: Software Tools for the Professional Programmer* 25, 11 (2000).
Weikai Chen, Yuexin Ma, Sylvain Lefebvre, Shiqing Xin, Jonàs Martínez, and wenping wang. 2017. Fabricable Tile Decors. *ACM Transactions on Graphics (SIGGRAPH Asia)* 36, 6 (2017), 175:1–175:15.
Xuelin Chen, Honghua Li, Chi-Wing Fu, Hao Zhang, Daniel Cohen-Or, and Baoquan Chen. 2018. 3D Fabrication with Universal Building Blocks and Pyramidal Shells. *ACM Transactions on Graphics (SIGGRAPH Asia)* 37, 6 (2018), 189:1–189:15.
Gene C.-H. Chuang and C.-C. Jay Kuo. 1996. Wavelet Descriptor of Planar Curves: Theory and Applications. *IEEE Transactions on Image Processing* 5, 1 (1996), 56–70.
Mario Deuss, Daniele Panozzo, Emily Whiting, Yang Liu, Philippe Block, Olga Sorkine-Hornung, and Mark Pauly. 2014. Assembling Self-Supporting Structures. *ACM Transactions on Graphics (SIGGRAPH Asia)* 33, 6 (2014), 214:1–214:10.
Michael Eigensatz, Martin Kilian, Alexander Schiftner, Niloy J. Mitra, Helmut Pottmann, and Mark Pauly. 2010. Paneling Architectural Freeform Surfaces. *ACM Transactions on Graphics (SIGGRAPH)* 29, 4 (2010), 45:1–45:10.
Chi-Wing Fu, Chi-Fu Lai, Ying He, and Daniel Cohen-Or. 2010. K-Set Tilable Surfaces. *ACM Transactions on Graphics (SIGGRAPH)* 29, 4 (2010), 44:1–44:6.
Jiasi Gao, Jiangtao Gong, Guyue Zhou, Haole Guo, and Tong Qi. 2022. Learning with Yourself: A Tangible Twin Robot System to Promote STEM Education. In *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 4981–4988.
Konstantinos Gavriil, Ruslan Guseinov, Jesús Pérez, Davide Pellis, Paul Henderson, Florian Rist, Helmut Pottmann, and Bernd Bickel. 2020. Computational Design of Cold Bent Glass Facades. *ACM Transactions on Graphics (SIGGRAPH Asia)* 39, 6 (2020), 208:1–208:16.
Sean Gillies, Casper van der Wel, Joris Van den Bossche, Mike W. Taves, Joshua Arnott, Brendan C. Ward, et al. 2023. Shapely. https://github.com/shapely/shapely
Rebecca A. H. Gower, Agnes E. Heydtmann, and Henrik G. Petersen. 1998. LEGO: Automated Model Construction. *32nd European Study Group with Industry - Final Report* (1998), 81–94.
Branko Grünbaum and G. C. (Geoffrey Colin) Shephard. 1987. *Tilings and Patterns*. W.H.Freeman and Company.
Gurobi Optimization, LLC. 2023. Gurobi Optimizer Reference Manual. https://www.gurobi.com
Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J.

Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array programming with NumPy. *Nature* 585, 7825 (2020), 357–362.

Alejo Hausner. 2001. Simulating Decorative Mosaics. In *Proceedings of SIGGRAPH*. 573–580.

Jhen-Yao Hong, Der-Lor Way, Zen-Chung Shih, Wen-Kai Tai, and Chin-Chen Chang. 2016. Inner Engraving for the Creation of a Balanced LEGO Sculpture. *The Visual Computer* 32, 5 (2016), 569–578.

Julian Iseringhausen, Michael Weinmann, Weizhen Huang, and Matthias B. Hullin. 2020. Computational Parquetry: Fabricated Style Transfer with Wood Pixels. *ACM Transactions on Graphics (SIGGRAPH)* 39, 2 (2020), 12:1–12:14.

Caigui Jiang, Hui Wang, Victor Ceballos Inza, Felix Dellinger, Florian Rist, Johannes Wallner, and Helmut Pottmann. 2021. Using Isometries for Computational Design and Fabrication. *ACM Transactions on Graphics (SIGGRAPH)* 40, 4 (2021), 42:1–42:12.

Craig S. Kaplan and David H. Salesin. 2000. Escherization. In *Proceedings of SIGGRAPH*. 499–510.

Jae Woo Kim, Kyung Kyu Kang, and Ji Hyoung Lee. 2014. Survey on Automated LEGO Assembly Construction. In *Proc. WSCG*. 89–96.

Mikiya Kohama, Chiharu Sugimoto, Ojiro Nakano, and Yusuke Maeda. 2021. Robotic Additive Manufacturing with Toy Blocks. *IISE Transactions* 53, 3 (2021), 273–284.

Torkil Kollsker. 2020. *Mathematical Models and Algorithms for Optimisation of the LEGO Construction Problem*. Ph.D. Dissertation. Technical University of Denmark.

Torkil Kollsker and Enrico Malaguti. 2021. Models and Algorithms for Optimising Two-Dimensional LEGO Constructions. *European Journal of Operational Research* 289, 1 (2021), 270–284.

Torkil Kollsker and Thomas J. R. Stidsen. 2021. Optimisation and Static Equilibrium of Three-Dimensional LEGO Constructions. *Operations Research Forum* 2, 21 (2021).

Ming-Hsun Kuo, You-En Lin, Hung-Kuo Chu, Ruen-Rone Lee, and Yong-Liang Yang. 2015. Pixel2Brick: Constructing Brick Sculptures from Pixel Art. *Computer Graphics Forum* 34, 7 (2015), 339–348.

Kin Chung Kwan, Lok Tsun Sinn, Chu Han, Tien-Tsin Wong, and Chi-Wing Fu. 2016. Pyramid of Arclength Descriptor for Generating Collage of Shapes. *ACM Transactions on Graphics (SIGGRAPH Asia)* 35, 6 (2016), 229:1–229:12.

Sangyeop Lee, Jinhyun Kim, Jae Woo Kim, and Byung-Ro Moon. 2015. Finding an Optimal LEGO Brick Layout of Voxelized 3D Object Using a Genetic Algorithm. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation (GECCO '15)*. 1215–1222.

Zhenyuan Liu, Jingyu Hu, Hao Xu, Peng Song, Ran Zhang, Bernd Bickel, and Chi-Wing Fu. 2022. Worst-Case Rigidity Analysis and Optimization for Assemblies with Mechanical Joints. *Computer Graphics Forum* 41, 2 (2022), 507–519.

Zhong-Yuan Liu, Zhan Zhang, Di Zhang, Chunyang Ye, Ligang Liu, and Xiao-Ming Fu. 2021. Modeling and Fabrication with Specified Discrete Equivalence Classes. *ACM Transactions on Graphics (SIGGRAPH)* 40, 4 (2021), 41:1–41:12.

Sheng-Jie Luo, Yonghao Yue, Chun-Kai Huang, Yu-Huan Chung, Sei Imai, Tomoyuki Nishita, and Bing-Yu Chen. 2015. Legolization: Optimizing LEGO Designs. *ACM Transactions on Graphics (SIGGRAPH Asia)* 34, 6 (2015), 222:1–222:12.

Brick me. 2022. Brick.me. https://brick.me/ [Online; accessed 18-May-2023].

Yuichi Nagata and Shinji Imahori. 2021. Escherization with Large Deformations Based on As-Rigid-As-Possible Shape Modeling. *ACM Transactions on Graphics (SIGGRAPH)* 41, 2 (2021), 11:1–11:16.

Chi-Han Peng, Caigui Jiang, Peter Wonka, and Helmut Pottmann. 2019. Checkerboard Patterns with Black Rectangles. *ACM Transactions on Graphics (SIGGRAPH Asia)* 38, 6 (2019), 171:1–171:13.

Eric Persoon and King-Sun Fu. 1977. Shape Discrimination Using Fourier Descriptors. *IEEE Transactions on Systems, Man, and Cybernetics* 7, 3 (1977), 170–179.

Maxim Peysakhov, Vlada Galinskaya, and William C. Regli. 2000. Representation and Evolution of Lego-Based Assemblies. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*. 1089.

Eugene Smal. 2008. *Automated Brick Sculpture Construction*. Thesis. Stellenbosch : Stellenbosch University.

Kaleigh Smith, Yunjun Liu, and Allison Klein. 2005. Animosaics. In *Proceedings of the 2005 ACM SIGGRAPH/Eurographics Symposium on Computer Animation (SCA '05)*. 201–208.

Alexander Toshev, Ben Taskar, and Kostas Daniilidis. 2012. Shape-Based Object Detection via Boundary Structure. *International Journal of Computer Vision* 99 (2012), 123–146.

Ruocheng Wang, Yunzhi Zhang, Jiayuan Mao, Chin-Yi Cheng, and Jiajun Wu. 2022. Translating a Visual LEGO Manual to a Machine-Executable Plan. In *Computer Vision – ECCV 2022 (Lecture Notes in Computer Science)*. 677–694.

Xplicator. 2020. Mosaic Art. https://brickmosaicdesigner.com/ [Online; accessed 18-May-2023].

Hao Xu, Ka-Hei Hui, Chi-Wing Fu, and Hao Zhang. 2019. Computational LEGO Technic Design. *ACM Transactions on Graphics (SIGGRAPH Asia)* 38, 6 (2019), 196.

Hao Xu, Ka-Hei Hui, Chi-Wing Fu, and Hao Zhang. 2020. TilinGNN: Learning to Tile with Self-Supervised Graph Neural Network. *ACM Transactions on Graphics (SIGGRAPH)* 39, 4 (2020), 129:1–129:16.

Xuemiao Xu, Linling Zhang, and Tien-Tsin Wong. 2010. Structure-Based ASCII Art. In *Proceedings of SIGGRAPH*. 52:1–52:10.

Grim Yun, Cheolseong Park, Heekyung Yang, and Kyungha Min. 2017. Legorization with Multi-Height Bricks from Silhouette-Fitted Voxelization. In *Proceedings of the Computer Graphics International Conference (CGI '17)*. 40:1–40:6.

Jie Zhou, Xuejin Chen, and Ying-qing Xu. 2019. Automatic Generation of Vivid LEGO Architectural Sculptures. *Computer Graphics Forum* 38, 6 (2019), 31–42.