Designing Mini Block Artwork from Colored Mesh

Man Zhang[1], Yuki Igarashi[2], Yoshihiro Kanamori[1], Jun Mitani[1]

[1] University of Tsukuba, Tsukuba, Japan
{eelzhang,kanamori,mitani}@npal.cs.tsukuba.ac.jp
[2] Meiji University, Tokyo, Japan
yukim@acm.org

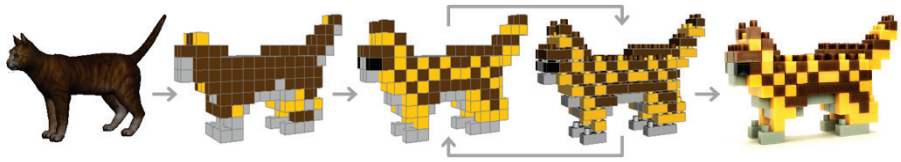

Fig. 1. Overview of our mini block artwork designing system

Abstract. Mini block artwork is a kind of well abstracted low-resolution block construction with aesthetically pleasing block layout. Similar with previous LEGO constructions, mini block artwork requires strong interconnection among blocks, i.e., a stable layout. However, what make mini block artwork different are the new requirements on highly abstracted shapes and colors and the regularity in block layout considering symmetry in the model itself. We focus on these requirements by first integrating quantization of colors into abstraction. We further explore layout generation method satisfying both stability and symmetry to support our prototype design system. Mini block artwork generated using different methods are evaluated on both stability and symmetry of the block layout. To facilitate a justified and discriminating layout comparison using stability, though we consider factors similar to classical heuristics, we experimentally optimize the weight of each factor for mini block artwork.
Keywords: LEGO®, mini block artwork, layout stability.

1 Introduction

A block is a convenient tool for fabricating physical objects. A well abstracted low-resolution design can save many block resources and much time for real block artwork. In this paper, we only focus on *brick pieces* considering the regular shape of a block. A Japanese product called Nanoblock [4], whose block length is almost half of the LEGO® block, has become popular in recent years. Examples of low-resolution block artwork can be found in the "Nanoblock mini collection [4]". Each design is assembled with approximately 200 blocks. The average voxel resolution we have observed is around 20 along the longest axis and around 4 along the depth axis. In

these Nanoblock products, regularity in layout is emphasized to make the design aesthetically pleasing as well as facilitating an easy building of real block artwork. A basic regularity is that blocks are placed symmetrically for symmetrical parts.

We face challenges with three aspects when designing mini block artwork. Firstly, block design aided by computers generally requires voxelizing an input mesh model. However, voxelization for generating a high-quality low-resolution color model is challenging. Moreover, state-of-the-art methods search for a constructible layout with higher stability, while new restrictions due to low-resolution, color, and symmetry require more than that, i.e., an ingenious balance between stability and regularity in the block layout. Finally, current LEGO design systems are not yet sufficiently convenient enough for the exploration of highly abstracted designs. Based on the challenges above, we briefly review the related work on these aspects below.

Low-Resolution Sampling from Colored Mesh. Most voxelization applications [9], [13], [14] are not optimized for low-resolution in terms of either shape or color. A recent approach called Binvox [6] improves the final low-resolution shape. However, color information is not used. Moreover, basically block colors are limited. Quantization [1], [5] has proved to be effective in reducing the number of colors. For our research, we use Binvox [6] to generate a relatively "qualified" low-resolution voxel model. We further color voxels by optimizing the nearest-neighbor sampling process, and apply a color quantization to satisfy the color restriction in the block set.

Layout Generation for LEGO Models. Previous research [3], [10], [11], [14], [15], [17] have mainly focused on automatic optimization of block placement. Heuristic-driven merging which originated in a group of mathematicians [3], is a classical layout optimization method. A more stable block layout encourages classical factors [3], [11], such as larger blocks, more connections, less collocated block edges, and more perpendicularly placed blocks in successive layers (long axes of two overlapped blocks toward differently). Moreover, graph theory is applied in recent studies [8], [10], [14] and promotes another layout optimization method based on the detection and the repairing of flaws in layout. State-of-the-art studies [8], [14] optimize layout considering both heuristics and graph theory. They use a random greedy merging algorithm to increase connections and larger blocks in the initialized layout. For constructability and more solidity, they further use an iterative local random re-layout to cope with disconnected block groups and weak articulation points, which are detected in a connectivity graph representing the initialized layout.

However, since classical factors for stable layout are not independent, it is difficult to maximize all these factors in a layout at the same time. In some cases, random greedy merging algorithm fails in generating layout encouraging perpendicularity (an indicator [3] describing how well each block covers the previous layer perpendicularly). To handle this, we explore another layout merging algorithm to increase perpendicularity, as well as encouraging larger blocks especially near the surface. For specific model, both of these two merging algorithms are tested in our system for an optimal choice. On the other hand, current re-layout based on random merging is too unpredictable to control the optimized layout. Considering the symmetry in mini

block artwork, we introduce a layout symmetrization algorithm and a mild reconnection algorithm for disconnected block groups. Combining these layout processing algorithms, we discuss a layout generation method satisfying both stability and symmetry.

Design System for LEGO Models. Previous designer systems for LEGO blocks can be roughly categorized into three types: mouse based (e.g., LEGO Digital Designer, BlockCAD, Mike's LEGO CAD, Leo CAD, LSketchIt [12], Build with Chrome, Blocklizer [16], and faBrickator [8]), multi-touch based [7], and immersive [2].

We propose a designing flow for mini block artwork and develop a prototype system as well. Our system automatically generates a colored low-resolution voxel model from an input mesh model, as shown in Fig. 1. The user can also recolor the model by mapping sampled colors to block colors supported in a block set. After that, repeated manual-editing and optimization are allowed to generate a constructible block model considering both stability and symmetry. We define an illegal voxel as a badly placed or colored voxel resulting in a non-constructible layout theoretically. Editing on the surface of a voxel model can be done to erase illegal voxels like those at the joint of the cat's left hind leg in Fig. 1 (b). Such manual editing is also used for surface decoration. After new decoration, layout is re-optimized. We also provide block layout editing, during which disconnected block groups are automatically detected and the user is notified by contrasting colors rendered for the model. Finally, it takes us 19 minutes to build a cat in Fig. 1 (e) using real blocks.

2 Voxelization and Coloring

To better preserve shape features in the original mesh model, we choose the voxelization [6] revised for low-resolution shapes. Because a voxelized low-resolution model is largely transformed from its original shape, as observed in the head, legs and tail in Fig. 2, finding an appropriate color for each surface voxel is a challenging task. We found that point sampling of a certain triangle color based on Euclidean distance causes many unexpected colors, depending on the quality of the mesh. To inhibit this dependence, we employ *color sampling* based on Manhattan distance. We first find triangles intersecting rays emitted from the center of a target voxel to the centers of 26 neighboring voxels. We calculate colors of not only voxels visible from outside but also their neighbors lying one voxel inwards; we experimentally found that regarding surface voxels as two-voxel thick works well in the subsequent process. For each surface voxel, we select the most common color among the triangles as a voxel color. If two or more colors have the same counts, we randomly choose one. Due to the variation in shape, some surface voxels can get one or more correlated triangles, but some may not get any. We refer to the former as an *occupied voxel* and the latter as an *empty voxel*. A step of *color propagation* finally assigns each *empty voxel* a color chosen from neighboring *occupied voxels* (inside a 3×3×3 cube centered at this *empty voxel*). To calculate the color of *empty voxel*, each neighboring voxel is initially assigned a weight $w=4-d$, where d is the Manhattan distance

from the voxel's center to the cube's center. Such a weight is then accumulated for each color. The color weighted most is chosen. If two colors have the same weight, we randomly choose one.

To reduce the number of colors in voxelized model, we use a *color quantization*. We implement *color sampling* earlier than *color quantization* in order to sample more original colors. Our *color quantization* clusters sampled colors into an *N*-color set, where *N* is the maximal number of colors allowed in the current design. The *N*-color set is calculated according to whether the voxel color is sampled from texture or surface color. For a textured model, *N* colors are extracted from the texture using the method by Gerstner et al. [1]. For a mesh model with surface color, we simply select *N* sampled colors processed by most voxels. Note that we select *N* colors as our clustering centers, but not necessarily use all these *N* colors for the block artwork. After clustering, these *N* colors can be remapped to any color in the block set. With our prototype system, *N* is set to six by default because the standard color set for Nanoblock contains six colors.

In summary, as shown in Fig. 2, our color-assigning for voxelized result is composed by: 1) coloring *occupied voxels* using *color sampling*; 2) reducing the number of colors among *occupied voxels* using *color quantization* (implemented in LAB color space); 3) coloring *empty voxels* using *color propagation*.

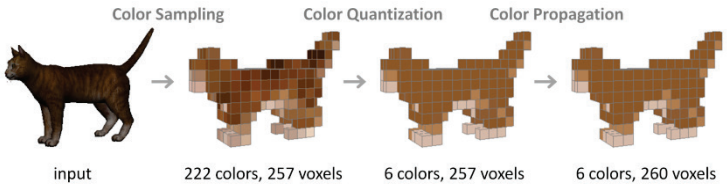

Fig. 2. Automatic color processing flow for the cat in Fig. 1. The numbers of valid colors and colored voxels in each voxel model are noted.

3 Block Layout Generation

In Section 3.1, we introduce newly explored layout processing algorithms. By combining them with state-of-the-art method [14], our proposed design system finally generates an optimized layout for mini block artwork.

3.1 Layout Processing Algorithms

We introduce three layout processing algorithms: perpendicularly ordered merging, subpart connection, and layout symmetrization.

Perpendicularly Ordered Merging. A merging is legal if it ensures the original voxel colors visible outside and generates a block in the block set. In a low-resolution model, the first seed voxel chosen to be merged (white rectangle in Fig. 3) will greatly affect the merging trend in the entire layer. Therefore, seed voxels need to be ordered.

We introduce two ordering principles both used in this algorithm, "surface-voxel preceding" and "layout alternating". The former merges surface voxels into larger blocks prior to invisible voxels. The latter encourages perpendicularity in successive layers. For each layer, we merge voxels into blocks as follows.

1. Store voxels in the model in an unmerged voxel list *L*.
2. Sort *L* using both principles, and choose, erase a seed voxel from top of *L*.
3. Find the legal set of neighbors with which the seed voxel can be merged.
4. Calculate the cost value developed by Testuz et al. [14], merge, and erase neighbors from *L*.
5. Goto Step 3 unless no neighbor can be legally merged with the seed voxel.
6. Goto Step 2 unless *L* is empty.

In Step 2, according to our two ordering principles, the unmerged voxel list *L* is sorted by considering two scores. One score is the number of neighbors able to be merged with this voxel. Since surface voxels have fewer neighbors, the first principle can be applied. The other score for the second principle involves a voxel's coordinate in a 3-dimensional array representing the voxel model. The score for voxel (*x, y, z*) equals *x* for layer *y=m* and equals *z* for layer *y=m+1*. An example of choosing a seed voxel using these two scores is illustrated in Fig. 3. Based on the two scores used for the sorting, the voxel with fewer neighbors will be assigned a higher priority. For two voxels with the same number of neighbors, we preferentially choose that with smaller *x* or *z*. A *naively ordered merging* considering only the first principle is compared with our *perpendicularly ordered merging* in Fig. 3.

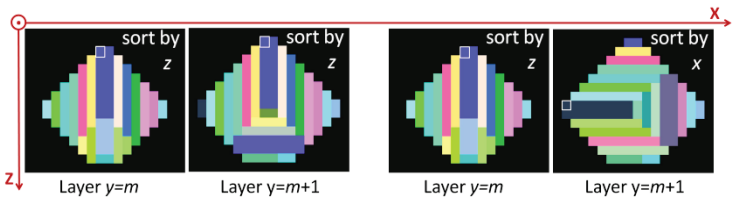

Fig. 3. A comparison of layouts in successive layers generated by *naively ordered merging* (two layers on the left) and *perpendicularly ordered merging* (two layers on the right). The first merging seed voxel is marked by a white square.

Subpart Reconnection. We rename disconnected block group as subpart for short. Our reconnection for subparts is achieved by manipulating the *separating section* in the layout. A *separating section* separating two blocks into disconnected subparts is a small section equivalent to the side face shared by both blocks. Unlike the state-of-the-art method involving locally-repeating random remerging [14], we first detect all the *separating sections* in the current block layout then create a new link across each *separating section* to connect the neighboring subparts. To change a *separating section* (orange line) into a link (orange arrow), as shown in the local layouts in Fig. 4, the following three steps are required.

1. Between the two separated blocks, divide the larger block along edges of the smaller block. Note that new edges (red lines) are created during this step.
2. Legally merge separated blocks to erase the *separating section*.
3. Erase new edges created in Step 1 by legally merging separated blocks.

Fig. 4 shows that our subpart reconnection algorithm has an advantage in changing little of the original layout. This elegant manipulation is suitable for subpart handling in a symmetric layout because fewer layout changes are preferred when maintaining symmetry in the layout.

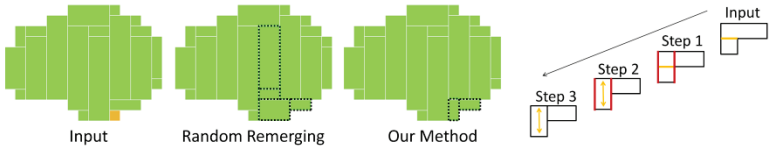

Fig. 4. Combining subparts using random remerging algorithm [14] and our subpart reconnection algorithm. To reconnect orange block, by using three steps, our method results in less change in layout (black rectangles with dotted border).

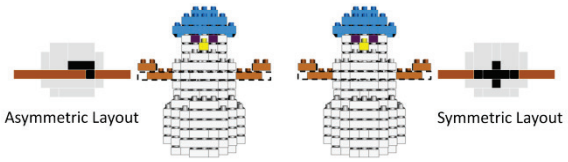

Fig. 5. An asymmetric layout (left) is modified to a symmetric layout (right)

Layout Symmetrization. With this algorithm, symmetry in shape is automatically detected considering an assumed axis of symmetry. Assumed axis of symmetry is simply calculated as the medial axis of a bounding box for voxels in the current layer. Our layout symmetrization algorithm for mirror symmetry can be summarized as follows.

1. Detect the axis of symmetry.
2. Scan and record block edges parallel to the x-axis (x-edge) or z-axis (z-edge).
3. If the axis of symmetry is parallel to the z-axis (or x-axis), for each recorded z-edge (or x-edge), add a symmetrical z-edge (or x-edge) to split blocks.
4. If the erasing of an edge and its symmetrical edge both cause legal merging, do it.

Our layout symmetrization algorithm can be used for beautifying layouts visible outside and those invisible inside, as shown in Fig. 5. It focuses mainly on mirror symmetry; however, it may not ensure rotational symmetry. In this case, the user must modify it manually to improve the layout.

3.2 Layout Generation Method

In this section, we introduce our layout generation method satisfying both stability and symmetry for mini block artwork. The first step is to optimize for a more stable

layout. What we need is a reliable stability measure sensitive to layout change to help make the optimal choice.

Stability. To facilitate an intuitive layout evaluation, we introduce a layout stability measure calculated as the average stability of all the blocks in the layout. The following stability for each block is normalized to the range of 0 to 1.

$$Stability = 1 - \frac{C_s / N_s + C_o /(1 + N_o) + C_e R_e + C_u R_u + C_p / R_p + C_n R_n}{C_s + C_o + C_e + C_u + C_p + C_n} \quad (1)$$

Our stability measure for each block is modified from that defined by Petrovic [11]. Petrovic defined a stability measure calculated for the block model, assuming the shape of the model remains unchanged; therefore he minimized an index of total number of blocks to encourage larger blocks in layout. However, since we prefer larger blocks, we favor an index more sensitive to the change in block size. Compared with Petrovic's definition, indices used in Eq. (1) for each block remain almost unchanged except for that of the total number of blocks. We define an index of block size N_s instead, calculated as the volume of a block in voxel units. The notations C_s, C_o, C_e, C_u, C_p, and C_n are the weight parameters for the six indices of N_s, connection with other blocks N_o, block edge R_e, uncovered block surface R_u, perpendicularity R_p, and alignment of neighboring blocks R_n, respectively.

Let C_i be index i's weight parameter ($i \in \{s, o, e, u, p, n\}$). Here we determine each C_i so that stability values become as discriminating as possible among different layouts. The Eq. (1) shows that stability value is decided by the index i's importance which is proportional to C_i. Therefore, we define index i's importance as the multiple of C_i and a corresponding coefficient R_i. Because preference on each index is not known, for fairness among indices, we simply assume that each index's importance is identical, i.e., $C_i R_i = 1$ for $\forall i$. We further define index i's discrimination as D_i, and then have $C_i R_i = D_i$, i.e., $C_i = D_i / R_i$. From a statistical point of view, R_i and D_i are better to be averaged among different layouts. During our experiment, we tested layouts generated for 11 low-resolution color models, considering different merging methods (random greedy merging [14], naively ordered merging and perpendicularly ordered merging in Section 3.1). To separate the effect of each index, we calculated the stability as Eq. (1) assuming one weight parameter as 1 and the other five as 0. For index i, R_i and D_i are selected separately as the average and standard deviation of stabilities calculated for all the layouts. The values calculated for C_s, C_o, C_e, C_u, C_p, C_n are 0.866, 0.266, 0.850, 0.163, 1.778, 0.275 respectively. For our tested layouts, the range of stability is widened from [0.569, 0.728] (using naive weight parameter equaling 1) to [0.396, 0.714] (using above weight parameters).

Layout Generation Method. We use the following steps to optimize the layout based on our modified stability measure. In the first step for layout initialization, we test for both random greedy merging [14] and perpendicularly ordered merging and choose the layout with larger stability. In the next step for layout constructability, we prefer a layout with fewer subparts. However, considering the illegal voxels in the model, it is

not guaranteed that during this step all the subparts can be connected. After manually erasing the illegality, random remerging [14] can well achieve a constructible layout. However, due to the randomness, sometimes a great effort is needed for random remerging (e.g., 67 loops tested for our flower model) but not for *subpart reconnection*. Therefore, our optimization first iteratively performs *subpart reconnection* L_1 times ($L_1 \leq 5$). If it fails in constructability, the smaller block beside each *separating section* is split smaller for extra L_1 times of *subpart reconnection*. To further explore a constructible layout with larger stability, we segment and remerge around weak articulation points, the same as done by Testuz et al. [14].

Especially for a symmetric layout, we aim at a final symmetrization resulting in less reduction of stability and fewer subparts. Subparts created in this step are further connected using the *subpart reconnection*. To maintain symmetry as well, layout change due to *subpart reconnection* is also handled symmetrically.

4 Results

We developed a prototype interactive system to facilitate the design of a mini block artwork. The prototype system was implemented using C++ and tested on a laptop with a 2.40-GHz, Intel Core (TM) i5-2430M processor, 8 GB RAM, and NVIDIA NVS 4200M GPU. We evaluated our system in different steps of the processing flow. Test mesh models, including those with texture (e.g., cat, flower and camera) and surface color (e.g., Legoman, headphone, and sunglass), were taken from free sources available online. Binvox [6] was used for low-resolution voxelization. For coloring, we compared our algorithm with naive alternatives. For layout generation, we compared our method with the state-of-the-art method [14].

Table 1. Layouts initialized for voxel models with (w/) or without (w/o) color quantization

w/ quantization	Color	Stability	Subpart	w/o quantization	Color	Stability	Subpart
cat	6	0.583	5	cat	222	0.564	21
flower	5	0.643	1	flower	37	0.576	18
camera	6	0.621	2	camera	15	0.588	3

Coloring. To evaluate our system based on quantization and sampling, we tested meshes with texture and surface color. Table 1 shows that our quantization can efficiently decrease the number of colors in a model; therefore, contributing to a more stable layout. When implementing the nearest-neighbor sampling, color results can vary when considering different searching areas and different distance measures. Fig. 6 shows the comparison of three strategies: (a) Manhattan distance/6 neighbors, (b) Manhattan distance/26 neighbors, (c) Euclidean distance/26 neighbors. We can find that, results of (c) contain many artifacts, which might be caused by the mesh quality in the input model, such as the mesh difference between the left and right eyeglasses. However, this artifact can be removed using Manhattan distance instead, as shown in the results of (a) and (b). Compared with (a), by searching more neighbors, (b) avoids obviously wrong samplings. Therefore, (b) is finally adopted for our system.

Layout Merging. We used nine colored mesh models as our test models. To show the influence of color and low-resolution, we processed test models in two ways separately for "Color/Low" (Model ID "1-11" in Fig. 7 for the 9 test models, voxelized in a resolution of no larger than 16 and well corrected, with 2 test models corrected both symmetrically and asymmetrically for comparison) and "Black/High" (Model ID "12-26" in Fig. 7 for 5 test models, colored in black and voxelized in resolutions of 16, 24, and 32). With these processed models, we then applied the following three merging algorithms: the state-of-the-art algorithm of random greedy merging [14], and two of our layout merging algorithms (naively ordered and perpendicularly ordered introduced in Section 3.1). Fig. 7 shows the stability and number of subparts for each voxel model. We can find that naively ordered merging greatly reduces the subparts and layout alternating further increases stability, especially for "Black/High". For "Black/High", our perpendicularly ordered merging performs better than the random greedy merging method. However, for "Color/Low", it is difficult to judge which algorithm performs better. Therefore, in our optimization for mini block artwork, both of these merging algorithms are tested for choosing a more stable layout.

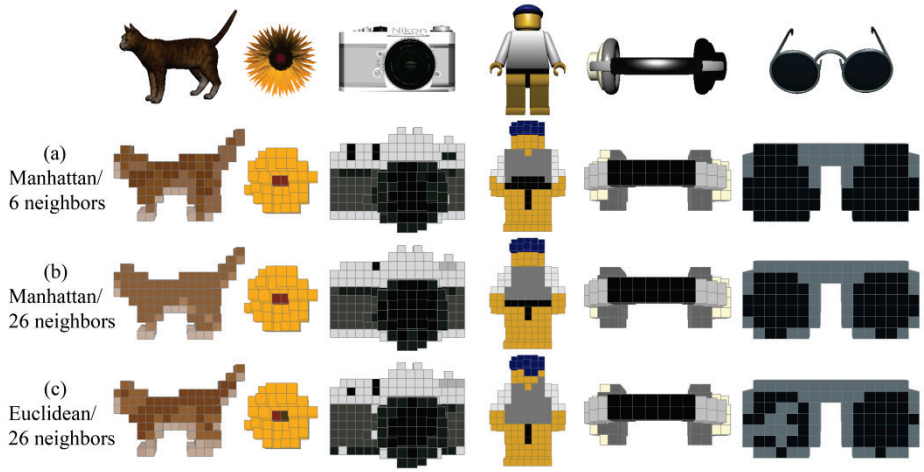

Fig. 6. Automatically abstracted voxel models without manual editing, considering three naive alternatives for color sampling strategy.

Theoretically, the coloring of voxels should only matter for those that are visible on the exterior. By keeping the interior color variable, there should be the greatest number of possible block layouts. However, our experimental results (Table 2) show that among layouts created for different thicknesses, layouts generated considering the surface color (thickness 1) are not always the most stable ones. This means that if we can find a heuristic coloring of inner voxels, there will be still room for improvement on stability of the initialized layout.

Layout Optimization. In regards to a comprehensive optimization for constructability and symmetry, we compared our layout generation method discussed in Section 3.2

with the state-of-the-art method [14]. In our experiment for the models in Table 3, we found that re-layout of 50 times did not ensure the removal of all the weak articulation points. However, the final constructability was guaranteed by ensuring one subpart in a model. Besides stability, we calculated another index involving all the edges in a layout, called layout symmetry, to show the percentage of edges having a paired edge in layout symmetrical to the assumed axis of symmetry parallel to the z-axis or x-axis. We can find that symmetry of the original input model is better maintained in the layout optimized with our method than that with the state-of-the-art method [14]. Though symmetrization is normally at the cost of stability, we can also find that almost half of our optimization results (rows in Table 3 from "dolphin_asym" to "dolphin_sym") exhibit larger stability than those of the state-of-the-art method [14]. Some intuitive layout comparison can be viewed in Fig. 8.

5 Conclusions and Future Work

We have proposed and developed a block design system considering various features such as stability, symmetry, and color separation, to assist in the generation of mini block artwork based on an input of 3D mesh. To facilitate our layout optimization and the additional manual decoration, we have quantitatively calculated an intuitive stability measure, as well as having experimentally tuned its weight parameters to make this measure more justified and discriminating. Some of the experimental conclusions about layout merging are thought provoking in exploring more effective layout generation method. Finally, we have discussed the feasibility and effectiveness of our method by comparing it with naive alternatives and state-of-the-art method.

In the future, to automatically create a better abstracted sampling, shape and color can perhaps be sampled in a way that adds to certain measures based on perception. The manual editing can be improved by a skilled surface decoration (e.g., character sculpture), an intelligent fixing for illegal voxels, and a heuristic coloring of inner voxels for a more stable layout. Moreover, perpendicularity in a layout can be further strengthened. Layout generation might also benefit from a proper classification considering features in model. To improve color selection, colors should probably be limited to real Nanoblock colors that are available to the builder. Constraints on the number and/or types of blocks can be incorporated into our system as well.

Table 2. Statistics (stability, number of subparts) for layouts merged considering different color restrictions (thickness of colored surface). Layout with highest stability for each model is marked in red. Note that blank cells indicating large thicknesses can not be set for thin models.

	Thick. = 1	Thick. = 2	Thick.= 3	Thick. = 4	Thick. = 5	Avg.
nightstand	0.600, 5	0.600, 5	0.600, 5	0.600, 5	0.600, 5	0.600, 5
soccer ball	0.591, 1	0.618, 1	0.622, 1	0.616, 1	0.622, 1	0.614, 1
camera	0.555, 1	0.557, 1	0.531, 1	0.531, 1	0.531, 1	0.541, 1
Legoman_sym	0.584, 5	0.535, 2	0.535, 2			0.551, 3
Legoman_asym	0.594, 6	0.541, 2	0.541, 2			0.559, 3
dolphin_sym	0.714, 3	0.714, 3				0.714, 3
dolphin_asym	0.658, 1	0.658, 1				0.658, 1
flower	0.606, 3	0.646, 3				0.626, 3

headphone	0.550, 4	0.526, 1				0.538, 3
cat	0.604, 1	0.595, 1				0.600, 1
sunglass	0.439, 2					0.439, 2

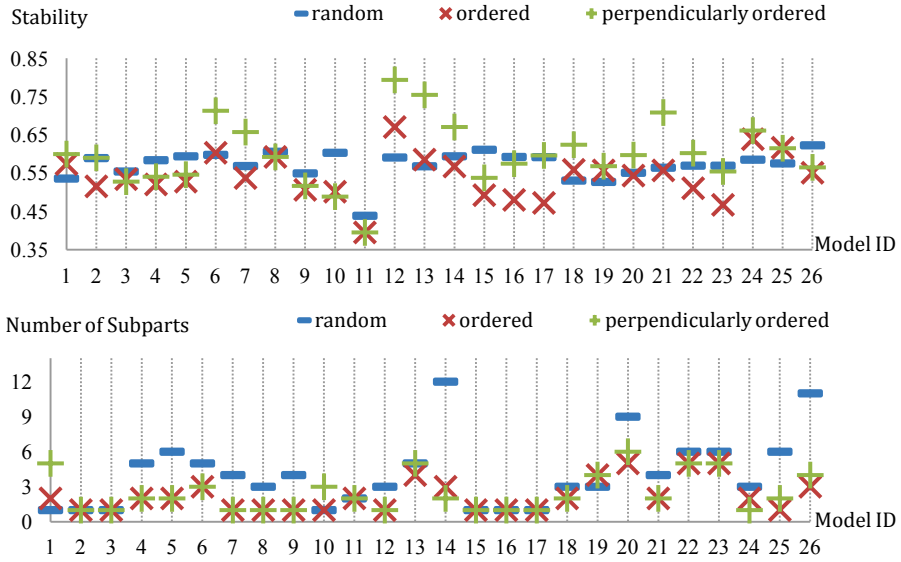

Fig. 7. Statistics of stability and number of subparts for layouts of different models created using different merging methods. Each model ID is shown along the horizontal axis.

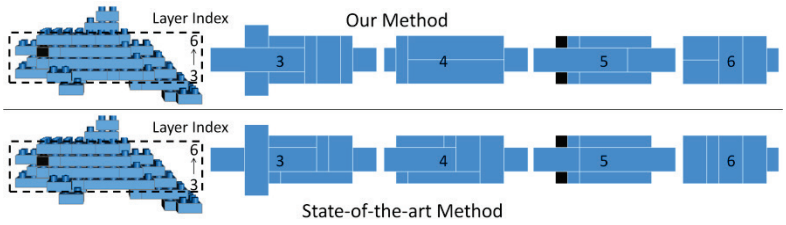

Fig. 8. Layouts layer by layer optimized using our method and state-of-the-art method [14]

Table 3. Comparison of stability and symmetry between two layout optimization methods. A 2-tuple for layout symmetry is shown considering that different models may have different degrees of symmetry along z-axis and x-axis.

Model	Stability			Symmetry (z-axis, x-axis)		
	Our method	Testuz et al.	Avg.	Input	Our method	Testuz et al.
dolphin_asym	0.684	0.619	0.652	0.911, 0.862	0.739, 0.333	0.582, 0.448
nightstand	0.580	0.529	0.555	1.000, 1.000	1.000, 0.481	0.726, 0.653
cat	0.624	0.593	0.609	1.000, 0.904	1.000, 0.542	0.952, 0.649
sunglass	0.454	0.433	0.444	1.000, 0.636	1.000, 0.357	0.581, 0.387
dolphin_sym	0.700	0.683	0.692	1.000, 0.910	1.000, 0.568	0.876, 0.528
camera	0.555	0.555	0.555	0.845, 0.668	0.471, 0.466	0.471, 0.466
flower	0.643	0.647	0.645	0.855, 0.773	0.643, 0.690	0.595, 0.690
headphone	0.536	0.544	0.540	1.000, 1.000	1.000, 1.000	0.632, 1.000
Legoman_asym	0.549	0.560	0.555	1.000, 0.889	0.997, 0.676	0.668, 0.732

soccer ball	0.571	0.590	0.581	1.000, 0.990	1.000, 0.782	0.746, 0.642
Legoman_sym	0.550	0.583	0.567	1.000, 0.891	1.000, 0.647	0.714, 0.684

References

1. T. Gerstner, D. DeCarlo, M. Alexa, A. Finkelstein, Y. Gingold, and A. Nealen. Pixelated image abstraction. In Proc. of the Symposium on Non-Photorealistic Animation and Rendering, 29–36 (2012).
2. A. Gupta, D. Fox, B. Curless, and M. Cohen. DuploTrack: A Realtime System for Authoring and Guiding Duplo Block Assembly. In Proc. of the 25th Annual ACM Symposium on User Interface Software and Technology, 389–402 (2012).
3. R. Gower, A. Heydtmann, and H. Petersen. LEGO: Automated Model Construction. Jens Gravesen and Poul Hjorth, 1998.
4. Kawada Co. Ltd. Nanoblock. http://www.diablock.co.jp/nanoblock/catalog/minicollection.
5. J. Kopf, A. Shamir, and P. Peers. Content-adaptive image downscaling. ACM Trans. Graph. (Proc. of SIGGRAPH 2013), 32(6):173:1–173:8 (2013).
6. P. Min. Binvox. http://www.cs.princeton.edu/~min/binvox/.
7. D. Mendes, P. Lopes, and A. Ferreira. Hands-on Interactive Tabletop LEGO Application. In Proc. of the 8th International Conference on Advances in Computer Entertainment Technology, 19:1–19:8 (2011).
8. S. Mueller, T. Mohr, K. Guenther, J. Frohnhofen, and P. Baudisch. faBrickation: Fast 3D Printing of Functional Objects by Integrating Construction Kit Building Blocks. In CHI ’14 Extended Abstracts on Human Factors in Computing Systems, 187–188 (2014).
9. F. S. Nooruddin and G. Turk. Simplification and repair of polygonal models using volumetric techniques. IEEE Trans. on Visualization and Computer Graphics, 9(2):191–205 (2003).
10. S. Ono, A. Andre, Y. Chang, and M. Nakajima. LEGO Builder: Automatic Generation of LEGO Assembly Manual from 3D Polygon Model. ITE Trans. on Media Technology and Applications, 1(4):354–360, 2013.
11. P. Petrovic. Solving lego brick layout problem using evolutionary algorithms. In Proc. of Norsk Informatik Konferanse, 87–97 (2001).
12. T. Santos, A. Ferreira, F. Dias, and M. J. Fonseca. Using Sketches and Retrieval to Create LEGO Models. In Proc. of the Fifth Eurographics Conference on Sketch-Based Interfaces and Modeling, 89–96 (2008).
13. L. Silva, V. Pamplona, J. Comba. Legolizer: A Real-Time System for Modeling and Rendering LEGO Representations of Boundary Models. In Proc. of the 2009 XXII Brazilian Symposium on Computer Graphics and Image Processing, 17-23 (2009).
14. R. Testuz, Y. Schwartzburg, and M. Pauly. Automatic generation of constructable brick sculptures. In Proc. of Eurographics 2013 (short paper), 81-84 (2013).
15. D. V. Winkler. Automated brick layout, 2005. BrickFest 2005.
16. M. Zhang, J. Mitani, Y. Kanamori, and Y. Fukui. Blocklizer: Interactive design of stable mini block artwork. In Proc. of SIGGRAPH 2014 Posters, 18:1–18:1 (2014).
17. L. V. Zijl and E. Smal. Cellular automata with cell clustering. In Proc. of Automata ’08, 425–441 (2008).