Computer-Aided Creation of Mini Block Constructions

September 2016

Man Zhang

Computer-Aided Creation of Mini Block Constructions

Graduate School of Systems and Information Engineering

University of Tsukuba

September 2016

Man Zhang

Abstract

Mini block construction is a kind of block construction representing a 3D model in a highly abstracted way, i.e., in the way of using only a small number of blocks. Due to this abstract representation, there are two distinctive characteristics that not only differ from previous LEGO constructions, but also make the creation (design and assembly) of mini block construction challenging.

Firstly, mini block construction has a nature and challenge of being designed in low-resolution space. Basically, LEGO construction is resolution-free, i.e., being able to be designed in any satisfying resolution. However, if allowing voxel-like blocks only to be used, previous LEGO constructions are prone to be designed in relatively high-resolution space in order to achieve an accurate representation. For mini block constructions, rather than an increase in resolution, to make up for the inaccuracy in low-resolution space, a challenge of finding a reasonable abstract representation in low-resolution space should be considered.

Secondly, mini block construction has a relatively high probability of suffering from fragile assembly. Compared with previous LEGO constructions, parts in a mini block construction are much more fragile because these parts are composed of much fewer blocks which cause much fewer interconnections as well. To enjoy the assembly of mini block construction, a well-designed set of instructions is crucial because fragile constructions of blocks might easily fall to pieces.

For the challenge of being designed in low-resolution space, we propose a method to automatically generate low-resolution voxel model by downsampling from a high-resolution voxel model. To optimize downsampled model in shape, a voxel attribute of *density* is calculated for each voxel in low-resolution voxel space. By employing voxel *density* in our method, we achieve our goal in two steps. Firstly, we have observed some objective laws in efficiently generating accurate shape features. Secondly, we move a step forward to further integrate reasonably abstracted shape features into the accurate results. We also conducted a user study and some experiments to evaluate low-resolution voxel models generated in different methods. Statistics showed that, models generated by our method were comparatively highly evaluated and won a common sense among different users.

During the design of mini block construction, we not only automatically abstract shape details from an original input (e.g., 3D mesh model, 2D sketch), but also integrate quantization of colors into abstracted shape if original color details are given. After generating low-resolution voxel model, we further explore generation and beautification of block layout to support our prototype design system for aesthetically pleasing mini block artwork.

For the assembly of mini block construction, we prefer a building guide which not merely solves the problem of fragmentation, but also is user-friendly by well considering users' assembly habits. A simple layer-by-layer, bottom-up assembly does not work well, especially when over-hanging regions exist. We propose a method for generating component-based building instructions aiming at supporting users to assemble block models efficiently. Our method contains two independent segmentations: segmentation at weakly-connected blocks and segmentation for avoiding *floating blocks*. Based on these segmentations, the whole model is divided into components. A set of building instructions is generated by deciding the assembly order of components. The effectiveness of our method is demonstrated through a user study.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Playing with LEGO$^{®}$, in most cases, like young children's block game, is enjoyed freely and requires no cautious design in advance. Another popular way of enjoying LEGO is assembling blocks following the guidebook of a commercial design.

By contrast, some LEGO players, not a few as well, are keen on assembling their original LEGO designs. Such a LEGO player initially decides a prototype (e.g., a type of animal or architecture) as what to be designed, and then starts to consider how to present the prototype with LEGO blocks. When considering the representation, one can choose to use many blocks to simulate the prototype as exactly as possible, or choose to use as few blocks as possible to abstract the prototype. The second choice, i.e., representing a prototype using as few blocks as possible, requires a skillful capture of features in the prototype. Actually, it is extremely challenging to achieve a well-represented design using only a small number of blocks for abstraction.

Back to the root problem of abstraction, predecessors' way of exploring widely-accepted abstraction approach was not smooth at all. The impressionist movement in the history of western art is full of rebellion and courage. When reacting against neoclassicism and romanticism, despite strong oppositions from critics, impressionist painters (e.g., Claude Monet, Camille Pissarro, Pierre-Auguste Renoir) finally put into practice their innovative ideas: recording an instantaneous impression of what they saw in front of them, often working quickly in a sketch-like manner which blurred the visual field. However, with the increasing acceptance by the public and by many of the critics, the unified momentum of impressionist movement was becoming to halfway, e.g., the impressionist's lack of emphasis on drawing became to be doubted. Many artists began to seek answers to their own questions about art. For some of these artists (e.g., Georges Seurat, Paul Czanne, Paul Gauguin, Vincent van Gogh), their subtly different approach such as the regularity in dots and strokes, the pure colors, eventually lead to their art being described as Post-Impressionist.

Our abstract block representation shares some similarities with the abstraction approach of Post-Impressionist: a regularly-shaped block corresponds to a dot or a stroke; a colorful block has a pure color. However, to automatically design an abstract block representation is still challenging, because even using the same abstraction approach, different paintings created by different painters are still different. In this paper, we will present our way of exploring an abstract block representation considering a common sense among different users.

1.1 Building Blocks of LEGO

As we all know, LEGO, is the world's most famous block brand. The LEGO group reported the following data in 2014 [40] that:

1. Globally over 15 million people visit the site of LEGO.com every month, the majority returning every day.
2. The LEGO Club (for children aged four to 11 years) has a global membership of nearly 5 million.
3. LEGO User Groups set up by "AFOLs" ("Adult Fans of LEGO") have a total of more than 200,000 registered members, and have their own websites, blogs and discussion forums. The most popular LEGO fan blogs have more than 300,000 unique visitors each month.
4. The LEGO Group has actively developed relations with approximately 200 LEGO User Groups.
5. 12 LEGO Certified Professionals have been officially recognized by the LEGO Group as trusted business partners.

LEGO is able to be as popular as mentioned above, because the LEGO Group has done an amazing job creating interchangeable pieces simply snap together and apart with no need for glue. A global LEGO subculture has developed since 1949, a year when the LEGO Group began manufacturing the initial interlocking LEGO bricks. Since the 1950s, to make LEGO more versatile, the LEGO Group has released thousands of sets allowing pieces with more diversified usages, from telling story of LEGO minifigures [38], to programming a multifunctional robot [37]. Over the years, approximately 700 billion LEGO elements [40] have been manufactured, which is enough for every person on earth to have an average of 94 LEGO elements [40]. Of course, some person might not need any; However, millions of LEGO fans hope to get "one more piece". Website like BrickRecycler.com [4] can accept new or used LEGO pieces, sometimes might either donate or sell pieces to help pay for the recycling costs.

1.2 Mini Block Constructions

Building blocks of LEGO have a wide range of applications, e.g., professional LEGO sculptures, entertainment activities, and rapid prototyping [57] that combines LEGO pieces with customized 3D printing.

Building a professional LEGO sculpture might be a tough work for general people. In most cases, professional LEGO sculptures are built by enthusiastic LEGO funs, or by members of LEGO Certified Professional which is a community-based program made up of adult LEGO hobbyists who have turned their passion for building and creating with LEGO bricks into a full-time or part-time profession. Professional LEGO sculptures can be very large and detailed. In May 2013, the largest model ever created [11] was a 1:1 scale model of an X-wing fighter, displayed in New York, made of over 5 million bricks.

Aiming at an entertainment for non-professional block players, instead of the professional LEGO sculpture, we focus on the creation of a customized, small-scale LEGO sculpture, which we called *mini block construction*. During the creation, we prefer the use of block pieces exampled by LEGO bricks, i.e., pieces assumed to be cuboid-like solids having the same height, as used in the first row of Fig.1.1. When the scale of the LEGO sculpture becomes smaller, reducing the number of bricks helps to save on cost and workload. Typical examples of mini block construction can be found in a series of designs called "Nanoblock mini collection" [48], which appears in a Japanese block brand called Nanoblock. Each design in this collection is assembled with approximately 200 pieces. The average voxel resolution we have observed is around 20 along the longest axis and around 4 along the depth axis. Brick length of Nanoblock in each dimension is almost half of LEGO brick, making a design more compact, portable and space-saving. Moreover, designs in this collection emphasize regularity in layout, making products aesthetically pleasing as well as facilitating an easy building using real block pieces. A basic regularity is that block pieces are placed symmetrically for symmetrical parts.

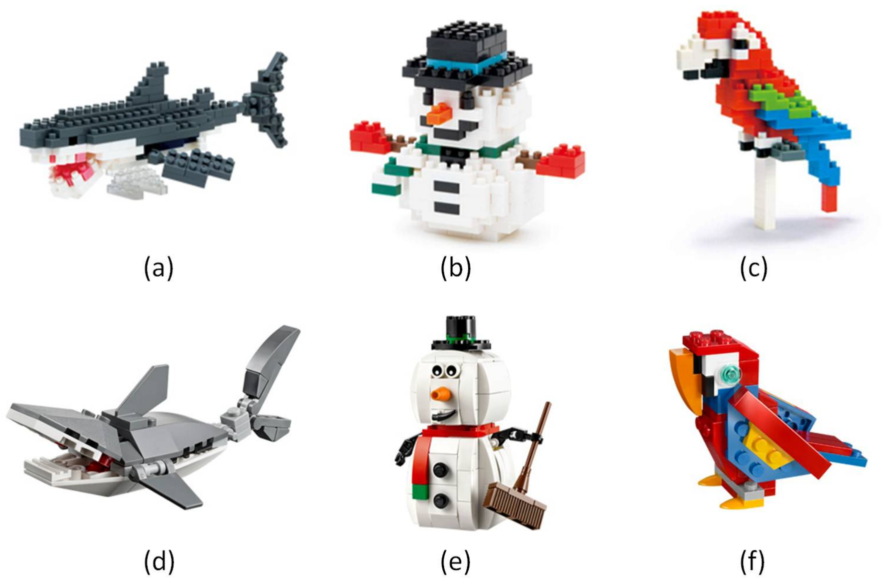


Figure 1.1: Mini block constructions commercially designed using cuboid-like bricks (1st row, object of our study) or using free shape pieces (2nd row). (a) Nanoblock NBC_082 [48]; (b) Nanoblock NBC_064 [48]; (c) Nanoblock NBC_034 [48]; (d) LEGO 40136 [39]; (e) LEGO 40093 [39]; (f) LEGO 30021 [39].

1.3 Motivations of Our Research on Mini Block Constructions

The traditional way for a user to manually explore a well-designed block construction is, try-and-error, by repeatedly assembling and disassembling tangible blocks, until finally realizing an expected design. However, the problem is that, such a repeated assembly experiment should be a time-consuming labor work, as well as making it difficult to predict required block resources in advance.

In academic area, creation of block constructions using a computer-aided tool solves above problems. For the creation in this way, available blocks are generally LEGO bricks with their originally defined sizes. Typical research topics include, 1) the automatic/semi-automatic design of virtual block constructions, and 2) the automatic generation of building instructions to guide the assembly using tangible blocks.

Research on creating an ideal block construction is much more complicate than child's play. For an input object, a creation tool initially generates a *voxel model* composed of cubic voxels monotone or colorful, then calculates a *block layout* (i.e. block arrangement) telling how to replace voxel/voxels into LEGO bricks, and finally shows building instructions step-by-step. For a long time, for professional LEGO players, designing a constructable, and strong, block model has been the mainstream. *Constructability* tells whether all blocks are interconnected as one. *Stability* measures whether a block model is weak, throughout the whole model, or at certain spots. Up to now, as far as we know, optimization of *block layout* optionally accounts for objects shape, colors, symmetry, pose, and external load sometimes.

In recent years, like the fast fashion, the creation of customized block constructions seems popular among amateur LEGO players. *Mini block construction* is one of these kinds. Because a *mini block construction* is basically designed in low-resolution space, a main challenge is to reasonably abstract a *voxel model* from a target 3D object. Moreover, because a *mini block construction* is generally weak in its thin parts, to avoid a suffering from fragile assembly, a reasonable segmentation of parts based on *block layout* is crucial.

These new research topics driven by *mini block construction* are potentially important and expected to grow. If the low-resolution abstraction from a target 3D object can be solved, a 2D abstraction for image processing can also benefit. If an assembly free from fragile block structure is available, more applications can be expected in the automatic assembly controlled by robots with cube-like modules, e.g., self-assembling robot of Roombots [71].

Currently, we aim at building a user-friendly system for the computer-aided creation of *mini block constructions*. Our prototype system at present is developed from our previously built demo system [84]. The effectiveness of our method is demonstrated through quantitative comparisons with other tools as well as several user studies. Our main research contributions are as follows:

1. For the convenient design of mini block construction, fill a 3D low-resolution voxelization gap by considering an automatic multi-degree abstraction.

2. For the aesthetically pleasing design of mini block construction, optimize the block layout while additionally accounting for symmetry in shape.

3. For the efficient assembly of mini block construction, provide a novel method for automatic generation of component-based building instructions.

4. Newly build a prototype system to facilitate the user-friendly creation (design and assembly) of mini block construction, especially for attracting people with few or no experience of LEGO.

In the future, we hope to share our developed system among more users. Because our system supports both original design and time-efficient assembly of *mini block construction*, users can be increased in a way much faster. For example, users of the first generation can explore their original designs, and rapidly assemble them as presents for friends who might probably be users of the second generation. In the final stage, we can also collect users' virtual designs, to build a benchmark composed of low-resolution block models, and to learn more data for a further improvement on abstraction algorithm or for a more general academic usage.

1.4 Overview of Our Prototype System and Our Approach

Previous designer systems for LEGO blocks can be roughly categorized into three types: mouse based (e.g., LEGO Digital Designer [36], BlockCAD [3], Mike's LEGO CAD [34], Leo CAD [41], LSketchIt [67], Build with Chrome [20], Blocklizer [87], and faBrickator [57]), multi-touch based [52], and immersive [23].

We have proposed a designing flow for mini block artwork [84] and have developed a mouse-based prototype system as well. Our system automatically generates a colored low-resolution voxel model from an input mesh model, as shown in Fig.1.2. 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 constructable block model considering both stability and symmetry. During this repeated process, an interactive manual editing on the surface of a voxel model is available 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, guided by our automatically generated building instructions, designed mini block construction can be efficiently assembled using real blocks. For example, it took us 19 minutes to build a cat in Fig.1.2(e).

There are two distinctive characteristics of mini block constructions: *designed in low-resolution space*, *assembled with few blocks*. These characteristics make the creation of mini block construction challenging, as well as making the computer-aided design expected.

Designing in low-resolution space. This causes the challenge of finding a discrete representation that is reasonably-abstracted from the prototype. In 3D space,

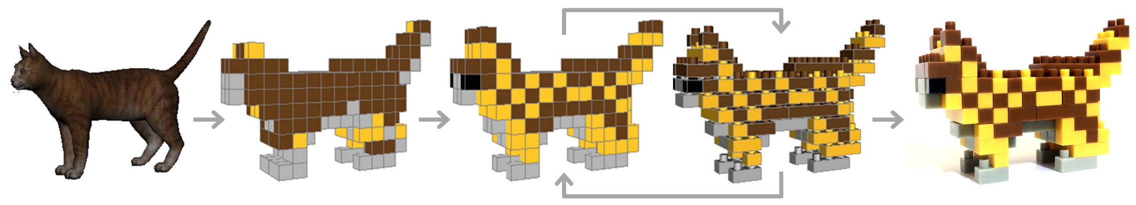


Figure 1.2: Overview of our prototype system for mini block construction.

generating a discrete representation from a continuous input has always been the basic task of voxelization. However, it is a pity that previous voxelization methods [9] [8] [13] [27] [29] seem good at precisely and efficiently "copy" (rather than "simplify" or "abstract") the shape from a continuous model to a voxelized model (usually in high resolution). The aim of abstraction [51] is similar to that of object description [58]: to only provide enough information (i.e., main features of the shape) for identifying an object as a member of some object class. At this point, abstraction is different from two other similar tasks: simplification and non-photorealistic rendering (NPR). Simplification aims at minimizing the deviation of the simplified object from the original one; NPR aims at highlighting or amplifying main features. Tasks of abstraction, simplification, and NPR have different advantages, therefore making a masterly abstracted low-resolution voxel model tricky to generate.

We propose a low-resolution (hereinafter, low-res) voxel model generation method, which aims at balancing *accuracy* and *abstraction* when downsampling a low-res voxel model from a high-res one. This task is not easy for conventional downscaling methods, as shown in Fig.1.3. In our method, we assume that a high-res voxel model is composed of features abstracted in three degrees: 1) features do not need to be abstracted, i.e., they need to maintain *accuracy* in a low-res voxel model; 2) features need to be moderately abstracted, i.e., they need to be balanced with *accuracy*; 3) features need to be deeply abstracted, i.e., they are not suitable for maintaining *accuracy*. To distinguish and handle these features in a model's shape, a voxel attribute of *density* is calculated for each voxel in low-res voxel space at first, and then used for deciding and generating our low-res results. To compare low-res voxel models generated in different methods, we conducted a user study to measure qualities of these models and to further observe different users' evaluations for the same model. Our user study showed that although people with and without art backgrounds evaluated abstract low-res voxel models differently, those models generated by our proposed method received a relatively high evaluation from most users.

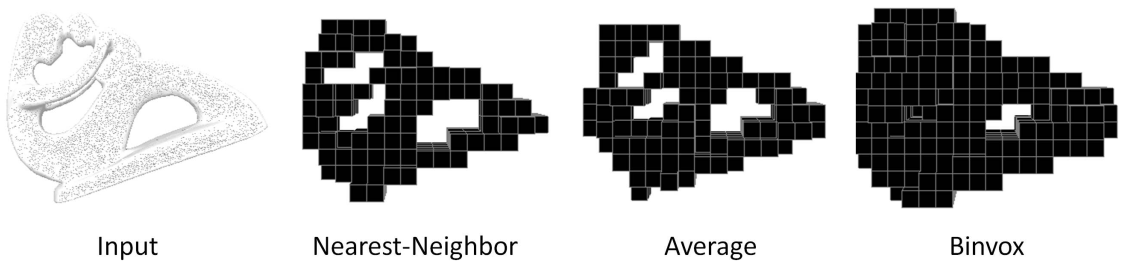


Figure 1.3: Low-res voxel models generated using different downscaling methods.

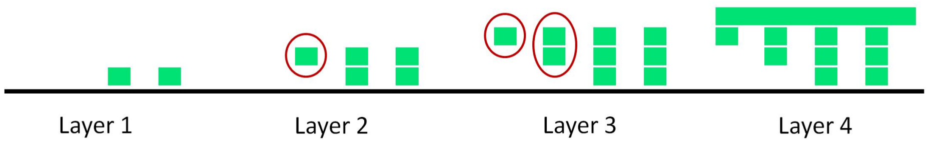


Figure 1.4: Blocks floating in the air (red circles) in naive layer-by-layer building.

Assembling with few blocks. This might cause a suffering when assembling a mini block construction fragile at extremely thin parts. Rather than a free assembly, it is crucial to create a well-designed set of building instructions. To avoid fragmentation during assembly, a smart strategy originating from assembling articulated objects [1] [2] [24] is to segment a model into solid components, assemble each of them separately, and finally combine them together. However, most block models do not have apparent articulations. For user-friendly assembly, a block model should be divided at weakly-connected blocks, and segmented into as few and as large components as possible to avoid over-segmentation. Also, the preferred assembly orders among LEGO fans seem to be "layer-by-layer and from bottom to top" [16], as these are natural orders for building architecture. However, if building instructions are not carefully designed, as shown in Fig.1.4, some blocks might have neither upward nor downward connections during assembly. Such physically-impossible blocks, defined as *floating blocks*, are not rare in instructions generated by existing LEGO design systems [16] [50] [73].

In line with the principles stated above, we propose a method for automatically generating building instructions for mini block constructions. Our method initially generates components, by masterly segmenting a model at the weakly-connected blocks and at the incoherent spots identified by *floating blocks*. During this step, it is ensured that no *floating block* exists in each component. For obtained components, our method further generates a set of building instructions by deciding the assembly order of the components based on our criteria for easy assembly. The effectiveness of our method is demonstrated through a quantitative comparison with other tools as well as a user study that proves users can assemble block models more efficiently using our instructions.

1.5 Structure of Our Paper

The rest of the paper is organized as follows. In Chapter 2, we briefly introduce our proposed method for low-res voxel model generation considering the balance of *accuracy* and *abstraction*. In Chapter 3, focus on the block model design of mini block construction, we summarize some state-of-the-art methods and our improvement [84] on the automatic generation of artwork-like block model considering shape, color, and block layout. In Chapter 4, focus on the assembly of mini block construction, we introduce our generation of component-based building instructions [85] [86]. Finally, in Chapter 5, we summarize the whole paper and discuss the future work.

Chapter 2

Balance of Accuracy and Abstraction in Generation of Low-res Voxel Model

The low-res voxel model is a 3D model represented in a 3D raster space containing a small number of voxels, as shown in Fig.2.1. A low-res voxel model includes only a small number of voxels, making it beneficial for its wide use. For example, a low-res voxel model satisfying certain physical constraints can be applied in LEGO design that uses few blocks and takes little assembly time, therefore making playing LEGO less expensive in terms of resources and time. Besides use in 3D, once projected to a 2D plane, a low-res voxel model generates 2D digital art that only handles a few pixels on 2D canvas. Such art is less expensive in terms of display, making it similar with a pixel art [31], which has been used in various graphics like those of characters in the first of the Super Mario series of games, or those icons in older desktop environments and in small-screen devices like mobile phones. Furthermore, volumetric graphics like 3D low-res voxel model and its 2D versions are also quite beneficial to give full play to the spatiotemporal resolution of volumetric displays using some emerging techniques. For example, we can expect many of these volumetric graphics to be displayed simultaneously in the air, when using emerging techniques like 3D midair laser plasma display (using nanosecond laser [5] or the safer femtosecond laser [61] to induce plasma in air for rendering), or pixie dust [60] (using acoustic beams of standing waves to levitate small particles for graphics).

Driven by the requirement of designing a mini block construction, in this chapter, we will introduce our automatic generation of low-res voxel model. Given a 3D mesh model as the input prototype, we aim at outputting a low-res voxel model, which is on the one hand similar to the input due to the expression of the same subject, and

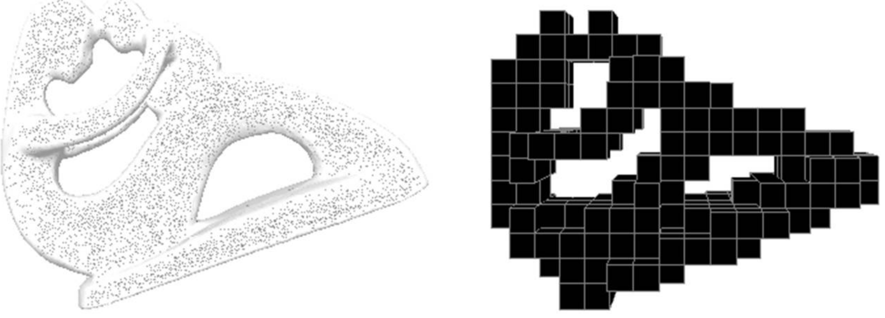


Figure 2.1: Example of manually edited low-res voxel model.

on the other hand different to the input due to the limited expressiveness in low-res space. The requirement of similarity hopes for *accuracy*, while the requirement of difference has necessity for *abstraction*. Therefore, we need a good balance between *accuracy* and *abstraction*.

The rest of this chapter is organized as follows. In Section 2.1, we will introduce the background of the research on voxel model generation, by reviewing some methods and some previous theories related to the *accuracy* and *abstraction* of objects. In Section 2.2, we summarize our main idea of optimizing *accuracy*. In Section 2.3, we further discuss how to improve optimized shapes for *abstraction*. In Section 2.4 and 2.5, we introduce our user study and compare low-res voxel models generated by different methods. In Section 2.6, we discuss some possible applications for using the low-res voxel models.

2.1 Background Knowledge

Voxelization algorithms have long been researched for generating a set of voxels that approximates a continuous geometric model. Previously published algorithms [9] [8] [13] [27] [29] [42] [12] have mainly addressed various conventional issues: geometrical accuracy, topological preciseness (e.g., separability between inside and outside of the model), and fast calculation.

The goal of previous voxelization methods can be summarized as to precisely and efficiently "copy" the shape from a continuous model to a voxelized model (usually in high resolution, rather than in low resolution). Note that "copy" is used here instead of "simplify", and also instead of "abstract" which aims at a representation of the shape to highly prioritize feature preservation. In fact, there is a profound reason for this. Voxelization was born in the research area of volume graphics [30].

In this area, voxelization is mainly used as a “pre-processing step” [13] that aims at pre-converting a 3D geometry into a prototype volume representation to store data in a 3D raster space. Prototype volume representation can be further handled by volume visualization in “post-processing steps”, such as abstraction, interpretation, and re-rendering. For this reason, abstraction for low-res representation has barely been addressed during voxelization.

Compared with applying voxelization in an unusual way to the generation of low-res voxel model, resolution reduction from high-res voxel model seems to be paid more attention. To generate a low-res voxel model by resolution reduction, a general solution mentioned in recent research [88] is to find a high-res discrete representation of input mesh to facilitate the calculation of error metrics, unlike the classical nearest-neighbor method [8] comparing directly with the input mesh. However, these two types of methods are similar at maintaining accuracy (i.e., low error in shape) to a target shape as much as possible. We call this rule *accuracy* first. On the other hand, to make a low-res voxel model seem like a pixel art, methods must not mechanically downsample for *accuracy* but move forward a step for representing the main shape features in input. We call this feature-considering rule *abstraction* first.

Accuracy is generally maintained by minimizing a well-designed error metric in shape, when shape details are simplified from original input. For example, a quadric error metric is widely used in surface simplification [17] [26] to produce high quality approximations of polygonal models. In a similar way, voxelized results are generally evaluated by a distance metric describing shape variation in Euclidean space [8], or by an objective function describing shape difference in voxel space [88].

Abstraction has been studied in many fields and disciplines. A thorough review of abstraction has been given by Mehra et al. [51]. On the one hand, capturing the main features is doubtlessly important for abstraction. Important shape features in the input can be the large protrusions [46], and the characteristic curves or contours [51]. However, as illustrated by Fig.1.3 in Chapter 1, important shape features are not guaranteed when using conventional downscaling methods. On the other hand, over abstraction should be avoided. For example, although some abstractive voxel models (e.g., manually designed PolyCube [72] and automatically simplified PolyCube [46]) guarantee the existence of voxels for main features, these models are too abstractive to be recognized as results resembling those original designs because even those basic poses have been changed, not to mention those important curves.

To abstract masterly is no less tricky than a pixel art. Pixel art is proposed as a metaphor, but pixel art was clearly, and strongly, justified by a concrete scenario (low resolution screens), which sets the rules and determined the objectives. In classical pixel art, which was once manually designed by artists rather than by mechanically downscaling higher resolution artwork, every pixel is carefully arranged, so such art is marked by its *minimalism and inherent modesty* [31], i.e., ultimate simplicity and strictness. Therefore, pixel art is not equivalent to a simple task of reducing resolution for 2D image.

Our research aims at balancing *accuracy* and *abstraction* when downsampling a low-res voxel model from a high-res one. To facilitate the design of mini block construction, a low-res voxel space of $16 \times 16 \times 16$ is considered. A high-res voxel model of $128 \times 128 \times 128$ is used because it is an empirical resolution [53] allowing adequate accuracy to a mesh input.

2.2 Main Idea of Optimizing Accuracy

Given a surface model as the input mesh, generation of a low-res voxel model corresponds to a task of selectively marking voxels as 1 in an all-zero low-res 3D raster space. In this paper, we assume the low-res 3D raster space to be $16 \times 16 \times 16$. Voxels in this 3D raster space can be divided into three types (*intersecting-voxel* exampled by the red rectangle in Fig.2.2, *inside-voxel* exampled by the green rectangle in Fig.2.2, and *outside-voxel* exampled by the blue rectangle in Fig.2.2) separately, with each intersecting the surface model, or being completely inside or outside the surface model. We make it a rule that all *outside-voxels* should be unmarked, i.e., being 0, and all *inside-voxels* should be marked, i.e., being 1. The problem is whether each *intersecting-voxel* should be 0 or 1. Each marking selection for all the *intersecting-voxels* in 3D raster space corresponds to a possible output, i.e., a low-res voxel model that allows a binary value assigned to each voxel. When choosing from all the possibilities, accuracy is a basic requirement because the output voxel model should roughly resemble the input mesh model in shape.

To summarize, our task is for an input mesh I, choosing an optimal (most accurate in shape) output O among all possible low-res voxel models VLs generated in a prescribed low-res 3D raster space of $16 \times 16 \times 16$. The number of VLs, noted as N, is decided by the number of *intersecting-voxels*. In low-res 3D raster space of $16 \times 16 \times 16$, the number of *intersecting-voxels* for general inputs is not small and is thought to be 100 here. It will make N larger than 2^{100}, which is too large to find O efficiently. To simplify the problem, we suggest a parameterization of VLs based on a concept of *density*.

Density of a voxel, as an important voxel attribute, was originally proposed for reducing the aliasing when rendering a discrete model. A discrete model can be generated from a continuous model using different methods [59] [69] [73], and will then be used for different volume graphics applications (e.g., the collision detection in flight simulations by discrete ray tracing, and the discrete normal estimation during shading). However, between the discrete model and the continuous model, there is always an aliasing which might cause maladies of voxel-based graphics. To solve this problem, Wang and Kaufman [78] proposed a technique of "volume sampling": sampling and filtering the voxels in 3D space to calculate a continuous scale attribute (e.g. a value ranging from 0 to 1) instead of a binary attribute (e.g. a value of 0 or 1) for each voxel. A voxel model generated with this grey scale voxel attribute is named an alias-free volume model. This voxel attribute in an alias-free volume model is able to reflect important shape characteristics in the original model and therefore can be directly used in many volume graphics applications. Zhou et al. [88] recently calculated a voxel attribute of "fullness" to satisfy low-res voxelization constraints arising from the fabricability and geometric-fit objectives in an application of folding 3D objects into boxes. In this paper, we further discuss the possibility of using a voxel attribute for obtaining a low-res voxel model satisfying not only geometric-fit objectives for accuracy but also subjective requirements for abstraction.

Density of a voxel v in low-res 3D raster space tells us the proportion of v that is contained by the input mesh I. It is defined as:

$$density(v) = \frac{volume(VH \cap v)}{volume(v)}, \tag{1}$$

where $0 \leq density(v) \leq 1$, and *VH* is a high-res voxel model that can be treated as a good representation of I. This concept is introduced as *fullness* in the research of Zhou et al. [88], but it is calculated slightly differently here. Their resolution of *VH* is not fixed, ranging from $60 \times 80 \times 80$ to $100 \times 100 \times 100$. However, due to the fixed size ($16 \times 16 \times 16$) of our low-res 3D raster space, we generate *VH* fixed as $128 \times 128 \times 128$, with a magnification equaling 8 as implemented in another widely used voxelization tool [53]. To generate *VH*, a general voxelization method supporting a high resolution will work. In this paper, we choose the solution of Binvox [53], which combines a parity count method and the ray stabbing method described by Nooruddin and Turk [59]. Generated *VH* is used to replace the I to efficiently calculate *density* as shown in Fig.2.2.

When *density* is calculated for each v in low-res 3D raster space, we mark voxels by setting a threshold t to filter *intersecting-voxels* with the same *density*, implemented

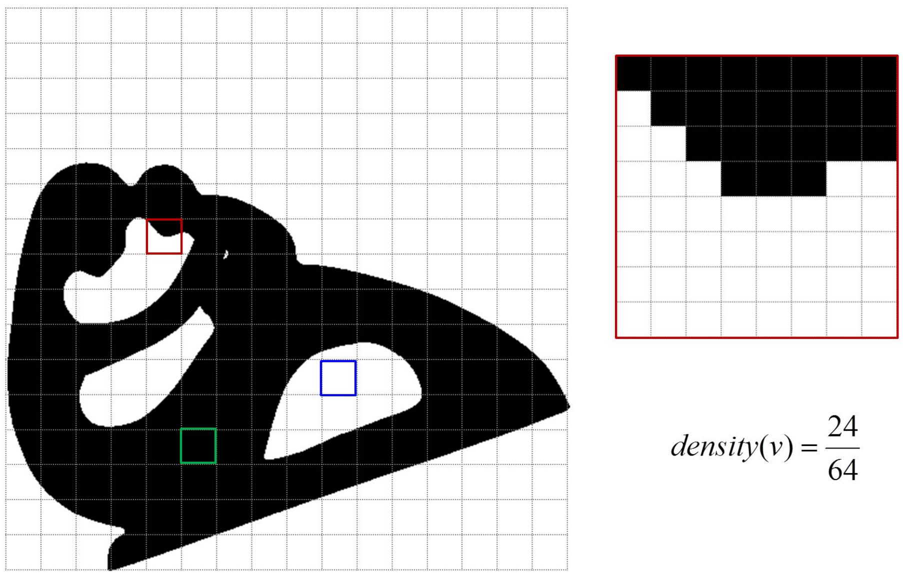


Figure 2.2: Calculation of *density* illustrated in 2D.

as:

$$vox(v,t) = \begin{cases} 0 & \text{if } density(v) < t, \\ 1 & \text{otherwise,} \end{cases} \tag{2}$$

where $0 \leq t \leq 1$. In this case, one t generates one VL, i.e., VLs are parameterized. Note that, similar to Fig.2.2, each $density(v)$ calculated in 3D is a multiple of $1/(8 \times 8 \times 8)$, i.e., 0/512, 1/512, 2/512, ..., 512/512. Therefore, t can be chosen from these discrete values. In this paper, we discretize t as a multiple of 1/128 to encourage a slight clustering of neighboring voxels with similar but probably different values of $density(v)$. Therefore, the total number of parameterized VLs is limited as $(128+1)$.

Due to the parameterization of VLs, it becomes much more efficient to find the output voxel model O optimized for *accuracy* (i.e., low error in shape). To find O, each $VL(t)$ should be compared with VH for error calculation. We treat difference in shape as error. Difference obtained at each low-res voxel is called E_{voxel}, while difference obtained throughout the whole low-res model is called E_{model}. We define E_{voxel} and E_{model} as:

$$E_{voxel}(v,t) = \begin{cases} density(v) & \text{if } vox(v,t) = 0, \\ 1 - density(v) & \text{otherwise,} \end{cases} \tag{3}$$

$$E_{model}(t) = \frac{8^3 \sum_{v \in VL(t)} E_{voxel}(v,t)}{volume(VH)}, \tag{4}$$

where 8^3 is the ratio of low-res voxel size to high-res voxel size, $volume(VH)$ equals the number of marked voxels in VH. For each $VL(t)$ generated by filtering using a t

as Eq.(2), we can calculate the E_{model} as Eq.(4). Finally, the $VL(t)$ with the minimal E_{model} is treated as our *accuracy* optimized output O.

2.3 Combination of Accuracy and Abstraction

In the section above, we have introduced our main idea of optimizing *accuracy* in low-res voxel model. However, we still have no intuitive feeling of what the most accurate result looks like. Moreover, generating a high-quality low-res voxel model should take into account abstraction as well. Currently, it is still ambiguous for which part and to what degree an abstraction is required. To make these both clear, in this section, we raise three assumptions (including one lemma and one theorem) step by step and further explain and develop them one by one.

2.3.1 Assumption for Accuracy

Eqs. 2-4 show this logic: t decides the value of a voxel v, further decides E_{voxel} of v, and finally decides E_{model} of the whole voxel model. Here a question comes up: how does E_{model} change with t? We give our answer by raising Lemma 1.

Lemma 1 When $t = 0.5$, E_{model} obtains a global minimum. A concept map is shown in Fig.2.3.

Proof. Let $density(v_i) = d_i$ where $v_i \in VL$, and let $f(t) = \sum_{v_i} E_{voxel}(v_i, t)$.
When $t_1, t_2 \in [0, 0.5)$, $t_1 < t_2$,

$$\begin{aligned}
& f(t_1) - f(t_2) \\
= \ & \big[\sum_{d_i<t_1} d_i + \sum_{d_i \geq t_1} (1-d_i)\big] - \big[\sum_{d_i<t_2} d_i + \sum_{d_i \geq t_2} (1-d_i)\big] \\
= \ & \big[\sum_{d_i<t_1} d_i - \sum_{d_i<t_2} d_i\big] + \big[\sum_{d_i \geq t_1} (1-d_i) - \sum_{d_i \geq t_2} (1-d_i)\big] \\
= \ & -\sum_{t_1 \leq d_i < t_2} d_i + \sum_{t_1 \leq d_i < t_2} (1-d_i) \\
= \ & \sum_{t_1 \leq d_i < t_2 < 0.5} (1-2d_i) \\
\geq \ & 0
\end{aligned}$$

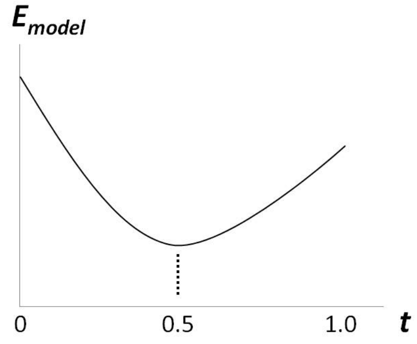


Figure 2.3: Concept map for global minimum of E_{model}.

$\Longrightarrow$ When t increases from 0 to 0.5, $f(t)$ will never increase.
When $t_1, t_2 \in (0.5, 1]$, $t_1 < t_2$,

$$\begin{aligned}
& f(t_1) - f(t_2) \\
= \; & \Big[\sum_{d_i<t_1} d_i + \sum_{d_i \ge t_1} (1-d_i)\Big] - \Big[\sum_{d_i<t_2} d_i + \sum_{d_i \ge t_2} (1-d_i)\Big] \\
= \; & \Big[\sum_{d_i<t_1} d_i - \sum_{d_i<t_2} d_i\Big] + \Big[\sum_{d_i \ge t_1} (1-d_i) - \sum_{d_i \ge t_2} (1-d_i)\Big] \\
= \; & -\sum_{t_1 \le d_i < t_2} d_i + \sum_{t_1 \le d_i < t_2} (1-d_i) \\
= \; & \sum_{0.5 < t_1 \le d_i < t_2} (1-2d_i) \\
\le \; & 0
\end{aligned}$$

$\Longrightarrow$ When t increases from 0.5 to 1, $f(t)$ will never decrease.

In summary, when $t = 0.5$, $f(t)$ obtains its global minimum. Because $E_{model}(t) \propto f(t)$, when $t = 0.5$, $E_{model}(t)$ also obtains its global minimum. □

Lemma 1 tells that, theoretically, the optimally accurate output O can be obtained by Eq.(2) when assigning t a value of 0.5. We call this threshold value $T_{density}$. By knowing such a threshold value $T_{density}$, i.e., 0.5, optimization for *accuracy* becomes extremely simple. Some examples generated in this way are shown in Fig.2.4.

However, as shown in Fig.2.4, we also found some unsatisfactory examples generated for models that have very thin parts. These examples explain that optimizing merely *accuracy* is not all-powerful. We should also take into account abstraction.

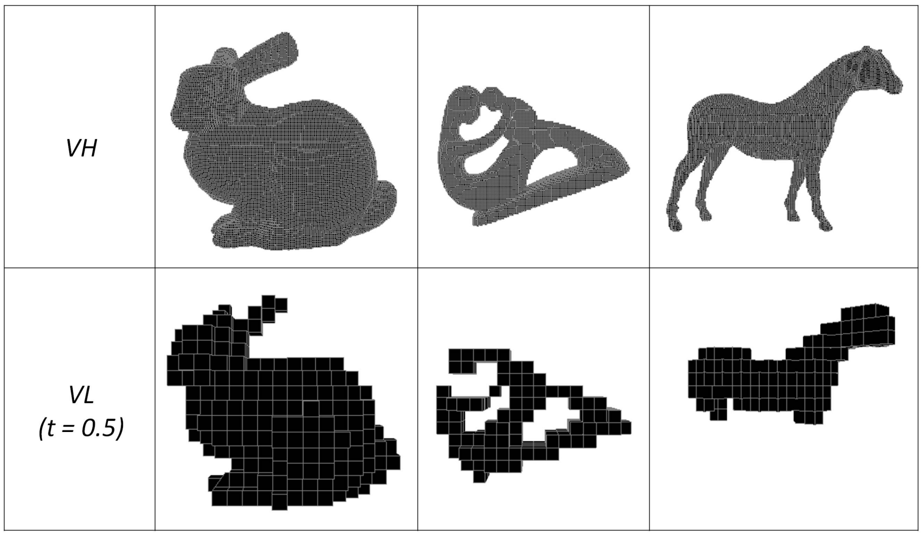


Figure 2.4: Low-res voxel models generated considering filtering voxels by a voxel *density* threshold equaling 0.5.

2.3.2 Assumption for Moderate-Abstraction

Lemma 1 concludes that a threshold $T_{density}$ exists and equals 0.5 when the optimization is for *accuracy*. However, optimization of low-res output should take into account abstraction as well. We want to know if there is a similar threshold $T'_{density}$ that is useful for abstraction. To find it out, we raise Assumption 2. For Assumption 2, we pre-generate a "voxel hull" for guiding an ingenious tradeoff between accuracy and abstraction in shape. This idea is inspired by a previous study related to abstraction. To preserve features at a desired level of abstraction, Mehra et al. [51] approximated the input geometric model by embedding the input model in a user defined regular grid and then extracting a "voxel hull", which is the set of all grid voxels that intersect any of the model triangles. This "voxel hull" is further used for extracting their main contribution of a well abstracted vector-based representation of 3D geometry for a man-made shape.

Assumption 2 A "voxel hull", defined as a set of *surface-voxels* (i.e., voxels that intersect any of the model triangles) in *VH*, includes enough shape details that are important for recognition. Similar to using *VH* to find $T_{density}$ for *accuracy*, we can use "voxel hull" to find $T'_{density}$ for *moderate-abstraction*, which is an abstraction balanced with accuracy. To facilitate the calculation, we extract outer visible voxels of *VH* to approximate "voxel hull" of *VH*.

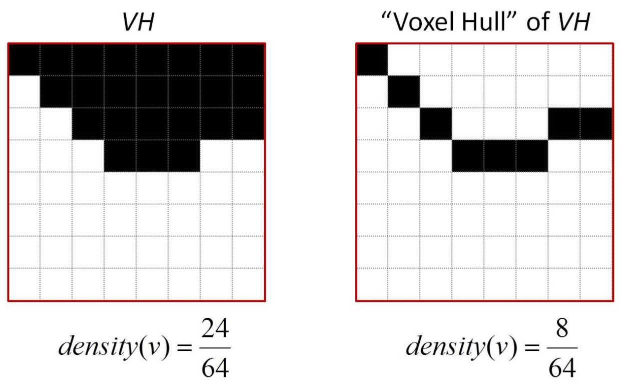


Figure 2.5: 2D illustrations for *density(v)* in *VH* and "voxel hull".

A simple approach to preserve *features* of models in Fig.2.4 is to use a t smaller than the $T_{density}$ and then mark voxels as Eq.(2). A proper decrease of t can increase marked voxels moderately. At the same time, a thicker part normally gains a lower growth rate than a thinner part, which can be explained as a feature-dependent abstraction. However, to avoid inaccuracy, this t should not be too small because when t is decreased, *intersecting-voxels* with relatively low *density* would be added. Marking voxels with low *density*, rather than unmarking them, causes higher E_{voxel} based on Eq.(3), therefore making an abstraction less accurate.

Now we know that, to balance accuracy and abstraction, the key point lies in a proper threshold $T'_{density}$. To further decide the value of $T'_{density}$, an analysis of $density(v)$ in different models is required. Generally, a *VH* has at least one thick area (see *inside-voxel* in Fig.2.2) causing the maximum of $density(v)$ to equal the highest value of 1. Besides, similar to the image of enlarged *intersecting-voxel* in Fig.2.2 (i.e., the left of Fig.2.5), a *VH* normally includes *surface-voxels* with many voxelized neighbors, which cause a relatively high average of $density(v)$ calculated in low-res voxel space. For these two reasons, we call this property of *VH* "high-density property". In contrast, $density(v)$ calculated using "voxel hull" of a *VH*, i.e., $density(v)$ calculated by changing *VH* in Eq.(1) into "voxel hull" of this *VH*, is much lower than that directly calculated using this *VH*, as shown in Fig.2.5. For "voxel hull", after testing different mesh models, we observed that calculated $density(v)$ is always less than 0.5 and has a maximum that is not fixed for all models but is normally around 0.2 or 0.3. We call this property of "voxel hull" "low-density property". On the basis of "low-density property", we developed the following theorem.

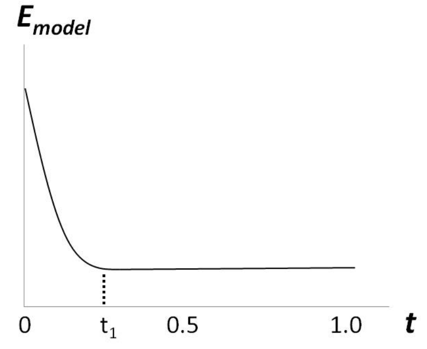


Figure 2.6: Concept map of E_{model} curve for an input voxel model with "low-density property".

Theorem 1 If an input model has a "low-density property" (i.e., $max\{density(v), v \in VL\} < 0.5$), then in the curve of E_{model} calculated for this input model, the global minimal of E_{model} can be obtained when $t = 0.5$ (proved in Lemma 1) but not uniquely by $t = 0.5$. More specifically, there is always a t_1 that satisfies both $t_1 < 0.5$ and $E_{model}(t_1) = E_{model}(0.5) = min\{E_{model}(t), 0 \leq t \leq 1\}$. A concept map is shown in Fig.2.6.

Proof. Let VH in Eq.(1) and Eq.(4) be a voxel model with "low-density property", so we have $density(v) \leq max\{density(v), v \in VL\} < 0.5$. We can without doubt find a d_1 satisfying $max\{density(v), v \in VL\} < d_1 < 0.5$.
$\because density(v) < d_1 < 0.5$ ("low-density property")
$\therefore vox(d_1) \equiv vox(0.5) \equiv 0$ (Eq.(2))
$\therefore E_{model}(d_1) \equiv E_{model}(0.5)$ (Eq.(3-4))
$\because E_{model}(0.5) \equiv min\{E_{model}(t), 0 \leq t \leq 1\}$ (Lemma 1)
$\therefore E_{model}(d_1) \equiv E_{model}(0.5) \equiv min\{E_{model}(t), 0 \leq t \leq 1\}$
In summary, d_1 is a qualified $t_1 < 0.5$ that satisfies the theorem. □

With Theorem 1, we can further decide the value of $T'_{density}$ for *moderate-abstraction*. We have observed that "voxel hull" has a "low-density property", therefore a threshold equaling d_1 ($max\{density(v), v \in VL\} < d_1 < 0.5$) satisfies a "voxel hull" guided *accuracy* (i.e., $E_{model}(d_1) = min\{E_{model}(t), 0 \leq t \leq 1\}$). At the same time, a smaller d_1 (i.e., $d_1 \to max\{density(v), v \in VL\}$) contributes to an abstraction able to preserve more *features*. Therefore, in our algorithm, we calculate $T'_{density}$ as a discrete value of t approaching from the positive direction to the $max\{density(v), v \in VL\}$, which is the highest $density(v)$ calculated considering an input of "voxel hull". After

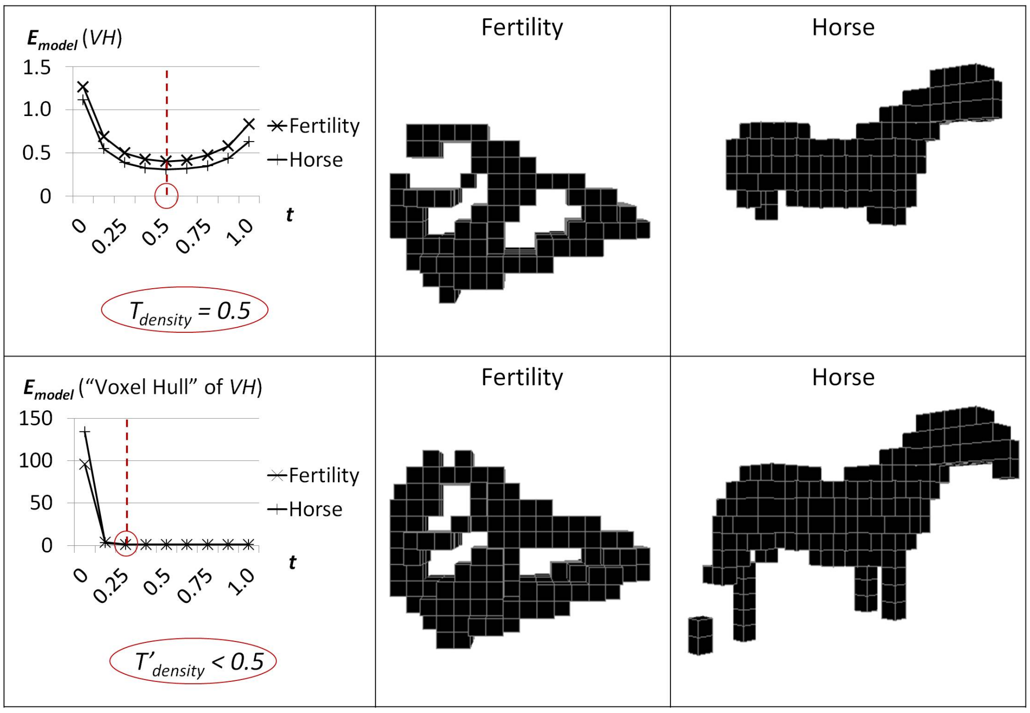


Figure 2.7: Comparison between $T_{density}$ optimized for *accuracy* (first row) and $T'_{density}$ optimized for *moderate-abstraction* (second row). The second and third columns show generated low-res results for two models. The first column shows calculated $T_{density}$ and $T'_{density}$ for two models. $T_{density}$ $(= 0.5)$ and $T'_{density}$ (< 0.5) are both calculated as the first (i.e., the minimal) t resulting in the global minimal of E_{model}. However, their E_{model} are calculated (see Eqs. 1-4) differently, i.e., using VH for $T_{density}$ and using "voxel hull" of VH for $T'_{density}$.

calculating $T'_{density}$ on the basis of "voxel hull", we apply $T'_{density}$ to filtering voxels in VH again for abstracted results. Some examples obtained in this way are shown in the second row of Fig.2.7.

2.3.3 Assumption for Deep-Abstraction

Basically, parts in a model are not constant in size: some are extremely thin like limbs, and some are regularly thick like the body and head. Fig.2.7 shows that $T'_{density}$ for *moderate-abstraction* can balance parts preserved for different sizes better than $T_{density}$ for *accuracy*. However, *moderate-abstraction* is still not enough for thin parts. The horse in Fig.2.7 is such an example. In fact, because thin parts cause extremely low *density* in low-res voxel space, their *accuracy* (low error) is almost impossible to reasonably ensure. However, to ensure their recognizability (e.g., existence and continuity), a thickness of one in low-res voxel space is generally enough. Therefore,

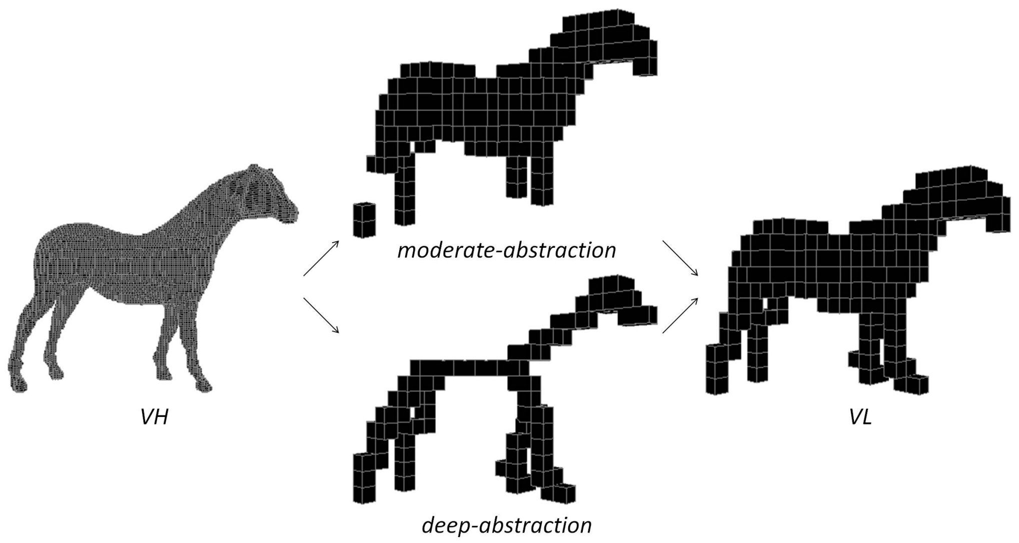


Figure 2.8: Improving low-res voxel model by a skeleton.

we suggest a *deep-abstraction* that does not explicitly focus on the constraint of *accuracy* but directly focuses on preserving the existence and continuity of thin parts.

Assumption 3 Optimized low-res voxel model can be further improved by appending the shape of a skeleton into the output model, as illustrated in Fig.2.8.

Global marking by $T'_{density}$ fails to well present certain shape details because the complexities in all parts crowded together on the surface. To solve this, many techniques for simplification can be employed to handle the complexities. Here, we choose to simplify them as a skeleton. We generate a skeleton in low-res voxel space in two steps. First, a high-res skeleton is obtained from *VH* by using a topology-preserving thinning method [54] [64] that ensures a maximum 26-connectivity in the result. To preserve topology, this method erodes voxels by matching templates in 8 subiterations. Second, the high-res skeleton is downsampled synchronously with *VH*. To ensure the recognizability of the skeleton, instead of using $T'_{density}$, we use a constant filtering threshold of $t \to 0$ (i.e., unmarking v only if $density(v) = 0$) independently for the skeleton.

2.4 User Study

Basically, a common sense of abstraction depends on a good sense of aesthetic feeling and probably a certain amount of professional training on observation of shape. Correctly recognizing and measuring abstraction is challenging, not only for a computer, but also for people with different academic and professional backgrounds. Therefore, we conducted a user study to explore people's recognition of abstract shapes. Our user study is designed as follows. We asked 20 test users to obtain their answers and feedback on questions involving some voxel models abstracted automatically. Considering that abstraction might be affected by users' professional training in art (painting, sketching, sculpting, etc), we found 10 users with a background in art and 10 without. For input mesh models, we selected three typical ones from those commonly used in research of shapes. For each input, we generated low-res voxel models viewed differently, by using six values of t for differences in thicknesses, and by appending a skeleton to each to ensure important parts. All tested models are shown in Fig.2.9: the bunny is a common watertight shape requiring *moderate-abstraction*; the cow is also a common watertight shape but requiring *deep-abstraction*; and the fertility is a torus-like watertight shape. Each of the six low-res voxel models for a input mesh model were graded by choosing a score from $A(0.9-1.0)$, $B(0.8-0.89)$, $C(0.7-0.79)$, $D(0.6-0.69)$, and $E(0-0.59)$. Considering the difficulty in evaluating a low-res voxel model, we asked each test user to give three scores on the basis of different criteria:

1. The number of *features* preserved.

2. The extent to which preserved *features* are similar to original sizes.

3. The total score compared with user's ideal design.

As expected, we observed that different users evaluated the same low-res voxel model differently. However, statistics revealed some interesting findings. We used statistical hypothesis test, T-test, to help us make more reasonable predictions.

2.4.1 t-related Good Zone

As shown in Fig.2.10 for users' ideal designs (criterion 3), although there are various changes in shapes of six low-res voxel models, a similar trend is found throughout test users' average scores shown in the first row. Moreover, what is surprising is that this trend well coincides with the curve of E_{model} for $T'_{density}$: users' highest scores in the first row of Fig.2.10, and the calculated minimal E_{model} in the second row of Fig.2.7, both mainly distribute at a t-related *good zone*. For the former, i.e., the first row of Fig.2.10, t-related *good zone* is approximately $[0.25, 0.5]$. For the latter, i.e., the second row of Fig.2.7, t-related *good zone* equals $[T'_{density}, T_{density}]$, where $T_{density}$

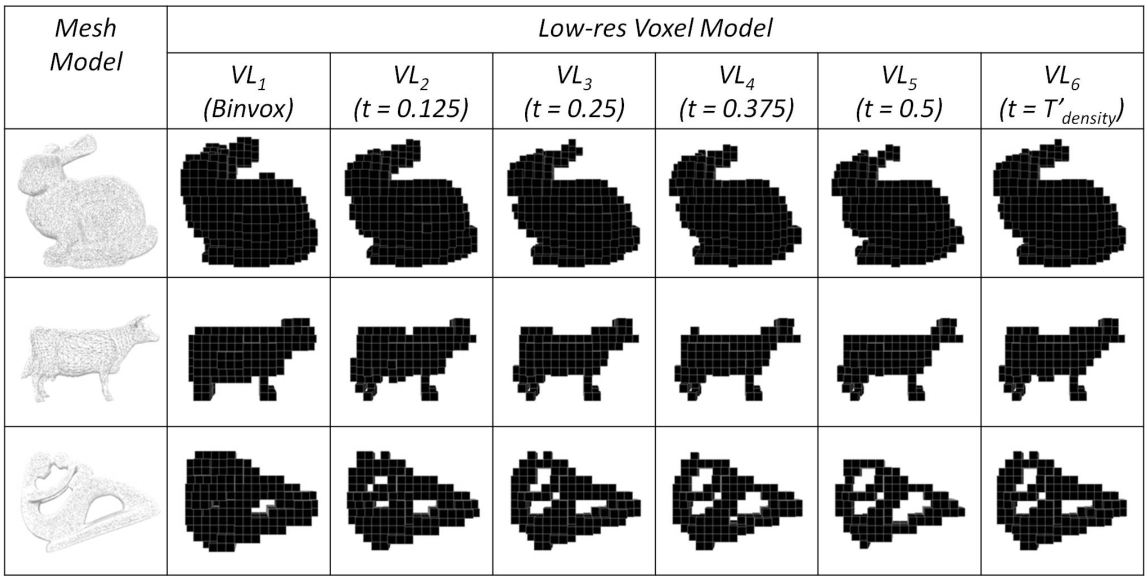


Figure 2.9: Test models used in user study. The first column of low-res voxel models is generated using a widely used voxelization tool of Binvox [53]. The other columns of low-res voxel models are downsampled from VH (voxelized from mesh model) in two steps: 1) calculating $density(v)$ as Eq.(1) and filtering $density(v)$ as Eq.(2) using the given t; and 2) appending skeleton extracted from VH.

always equals 0.5, $T^{'}_{density}$ is not fixed for all our tested models but is coincidently around 0.2 or 0.3. Due to this coincidence, we have reason to believe that $[T^{'}_{density}, 0.5]$ might be an ideal approximation of t-related *good zone* decided by test users. It also means that we can suggest automatically generated low-res voxel models to a user by considering this t-related *good zone*.

Because standard deviations shown in the second row of Fig.2.10 are not small, we are not sure whether the coincidence above about average scores can be found among more people or can only be found among our test users. To find the answer, we did T-test to help infer whether the t-related *good zone*, i.e., $[T^{'}_{density}, 0.5]$, works for more people. T-test is normally used for statistically checking if two means (averages) are reliably different from each other. In Fig.2.9, VL_1 corresponds to Binvox (coincidently being the same with a VL using $t \to 0$), VL_5 uses $t = 0.5$, and VL_6 uses $t = T^{'}_{density}$. We hope to demonstrate that, test users' average score for VL_6, is not only reliably higher than test users' average score for VL_1, but also not reliably different from test users' average score for VL_5. Suppose $X_i (i = 1, 2, ..., 6)$ to be a user's score for $VL_i (i = 1, 2, ..., 6)$, and $\bar{X_i}$ to be a value averaged among different users' X_i. We expect our T-test results to be: $\bar{X_6}$ is reliably higher than $\bar{X_1}$, but is not reliably different from $\bar{X_5}$.

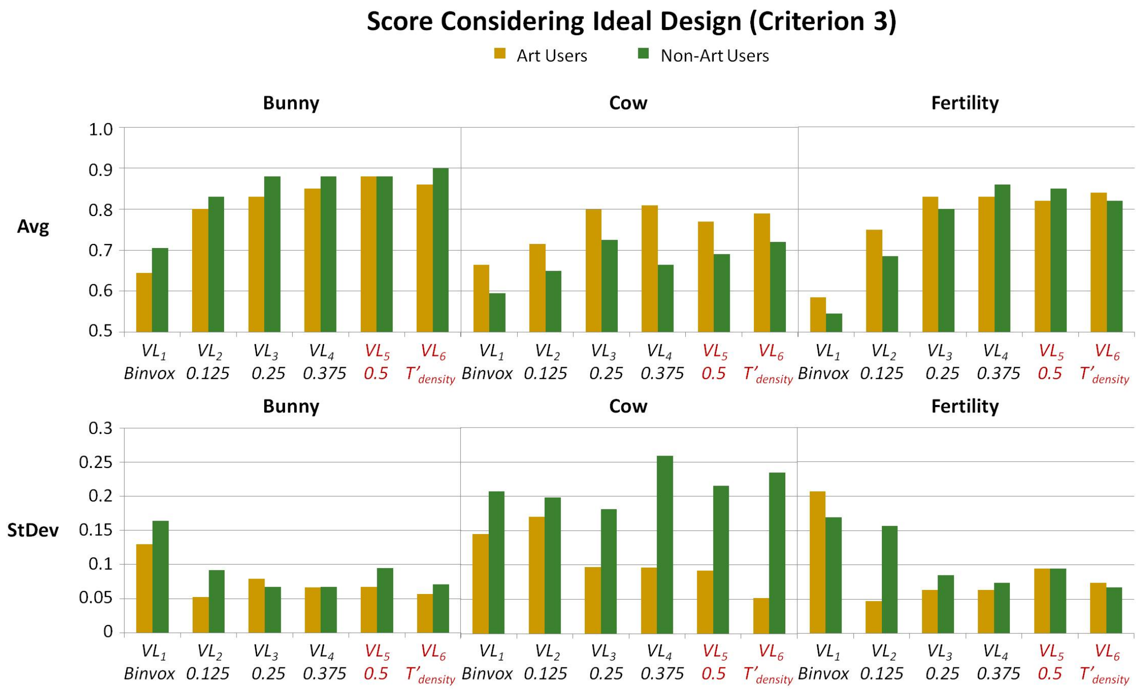


Figure 2.10: Statistics involving users' ideal designs (criterion 3).

For each of the two user-groups (10 art users, 10 non-art users), we did a paired-samples T-test to check the effectiveness of changing filtering threshold t in generating low-res voxel models. As shown in Table 2.1, p-values and t-values colored in red (in line with our expectation) pass our T-test, i.e., p-value < 0.05, t-value(9) > 1.8331. We can find that, a change from $t = T'_{density}$ (for VL_6) to $t = 0.5$ (for VL_5) does not significantly change users' average score. However, in most cases, a change from $t = T'_{density}$ (for VL_6) to $t \to 0$ (for VL_1) does lower the users' average score, although lowering in vary degrees considering different input mesh models and different users. The only exception is the input mesh model of cow for non-art users, with a reduction of average score not significant at all. Actually, it is caused by the larger standard deviations of non-art users' scores for cow, probably due to the difficulty in judging a low-res voxel model losing important features like cow's horns. In the next subsection, we will further discuss the variation between two user-groups.

2.4.2 Variation between Two User-Groups

Now let us focus on the variation between two user-groups, which is shown in the second row of Fig.2.10. Standard deviations of scores can be mostly observed as being smaller among users with a background in art (the yellow bars) than among users

Table 2.1: T-test results involving users' ideal designs (criterion 3). For all the obtained p-values and t-values, those colored in red pass our T-test, i.e., with significant difference between the two scores of $\bar{X}_i$; and those colored in black fail in our T-test, i.e., without significant difference between the two scores of $\bar{X}_i$.

		*Criterion*3			
		X_6-X_1		X_6-X_5	
		Art	Non-Art	Art	Non-Art
Bunny	t-value	4.557991	4.741079	1	0.480384
	p-value	0.000685	0.000529	0.171718	0.321208
Cow	t-value	2.747787	1.568983	0.688247	0.490990
	p-value	0.011279	0.075549	0.254323	0.317592
Fertility	t-value	3.938645	5.546414	0.612372	1
	p-value	0.001706	0.000179	0.277723	0.171718

without a background in art (the green bars), especially for the cow which requires *deep-abstraction*. This means that users with a background in art tend to evaluate the same model more similarly. Considering that scores involved in this figure are given on the basis of users' ideal designs, we can also say that users with a background in art have a better common aesthetic sense for ideal designs. It also means that we can suggest automatically generated low-res voxel models highly evaluated by users with a background in art, but leave more spaces for customized requirements from users without a background in art.

The reason for the better common aesthetic sense of users with art backgrounds can be found from scores for criteria 1 and 2. For criteria 1 and 2, we would like to know which criterion, or whether both criteria, is/are responsible for the better common aesthetic sense of users with art backgrounds. To find the answer, the T-test for criteria 3 in Section 2.4.1 was done for each of the criteria 1 and 2. As shown in Table 2.2, p-values and t-values colored in red pass our T-test.

Firstly, for art users' group and non-art users' group, T-test results are colored identically in case of criterion 1, but sometimes differently in case of criterion 2. Secondly, to further analyze the difference between t-values calculated for two user-groups, as shown in Table 2.3, compared with criterion 1, criterion 2 results in relative larger differences between two user-groups. These two evidences show that, the criterion 2, i.e., recognition of proportion among features, rather than the criterion 1, i.e., recognition of features' existences, should be more responsible for the better common aesthetic sense of users with art backgrounds. This is not difficult to explain. Similar scores can be given by users with art backgrounds probably due to their professional training in observation of shapes.

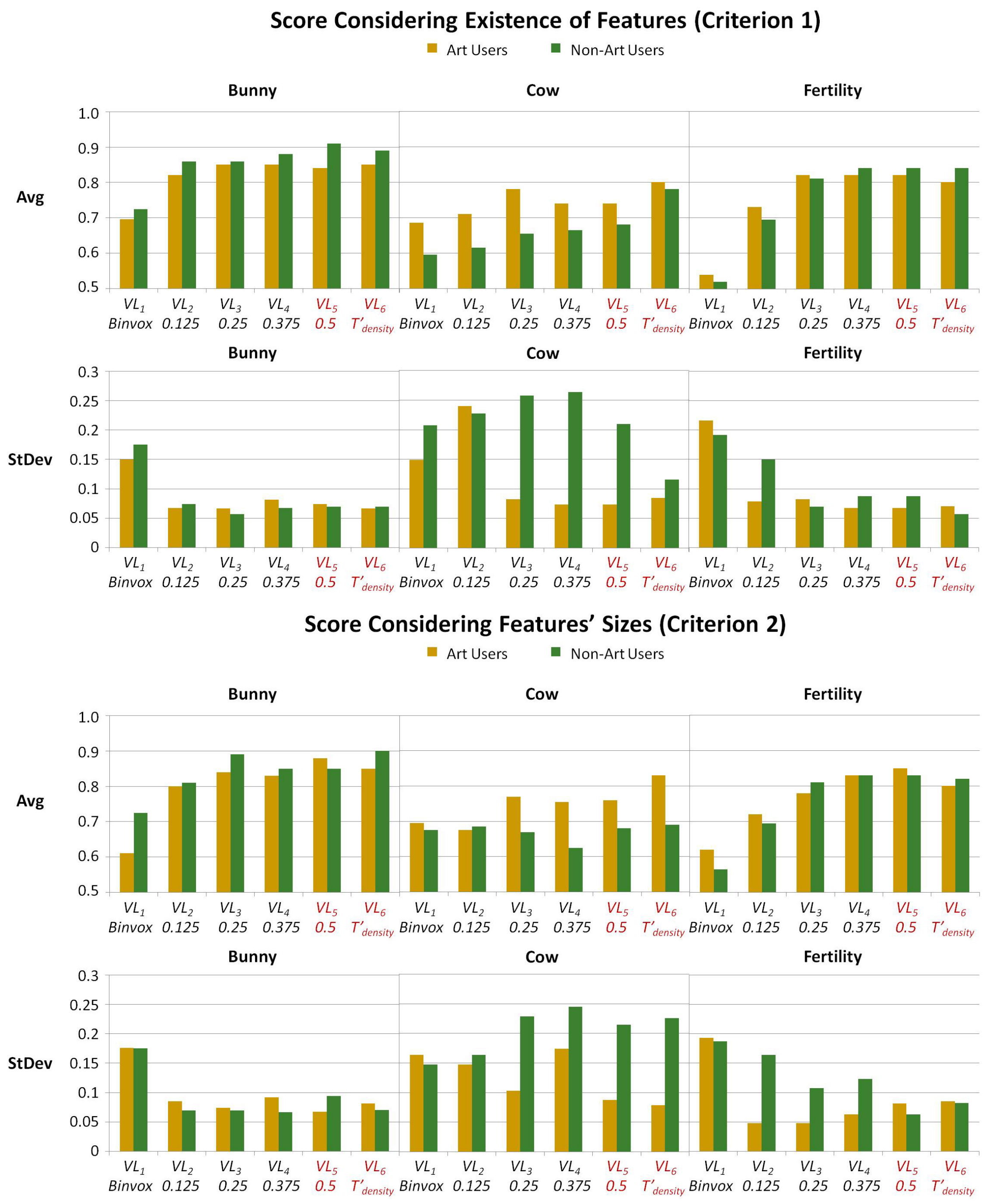


Figure 2.11: Statistics involving users' recognitions of important parts (criterion 1) and their sizes (criterion 2).

Table 2.2: T-test results involving users' recognitions of important parts (criterion 1) and their sizes (criterion 2). For all the obtained p-values and t-values, those colored in red pass our T-test, i.e., with significant difference between the two scores of $\bar{X}_i$; and those colored in black fail in our T-test, i.e., without significant difference between the two scores of $\bar{X}_i$.

		*Criterion*1				*Criterion*2			
		X_6-X_1		X_6-X_5		X_6-X_1		X_6-X_5	
		Art	Non-Art	Art	Non-Art	Art	Non-Art	Art	Non-Art
Bunny	t-value	3.119120	3.793688	0.428571	0.801784	5.709971	4.071725	1.152332	2.236068
	p-value	0.006168	0.002129	0.339155	0.221666	0.000145	0.001396	0.139436	0.026089
Cow	t-value	1.958679	3.077403	2.25	1.956984	2.104784	0.210905	1.768519	0.181237
	p-value	0.040909	0.006598	0.025502	0.041021	0.032307	0.418830	0.055382	0.430099
Fertility	t-value	4.088311	5.400422	0.612372	0	2.631799	4.364309	1.463850	0.557086
	p-value	0.001362	0.000216	0.277723	0.5	0.013639	0.000906	0.088634	0.295526

Table 2.3: Two user-groups' differences calculated by t-values in Table 2.2.

$\lvert$t-value(Art) $-$ t-value(Non-Art)$\rvert$	X_6-X_1		X_6-X_5	
	*Criterion*1	*Criterion*2	*Criterion*1	*Criterion*2
Bunny	0.674568	1.638246	0.373213	1.083736
Cow	1.118724	1.893879	0.293016	1.587282
Fertility	1.312111	1.73251	0.612372	0.906764

2.5 Results

We tested 9 models widely used in computer graphics. Values of $T'_{density}$ calculated for these models are circled in Fig.2.12. For an intuitive comparison, in Fig.2.13, we show low-res voxel models generated by a widely used voxelization tool Binvox [53], as well as using three possible abstraction methods, i.e., *accuracy* ($T_{density} = 0.5$), *moderate-abstraction* ($T'_{density} < 0.5$), and our proposed method that combines *moderate-abstraction* with *deep-abstraction*. For each marked voxel in a low-res voxel model, the calculated value of voxel error E_{voxel} (Eq.(3)) is associated with a specific color in a color map varying from red (1) to blue (0).

We can find from Fig.2.13 that large-error voxels (red voxels), which imply a possible large inaccuracy in shape, are contained most in low-res voxel models generated by Binvox [53] but are absolutely rare in low-res voxel models generated using the three possible abstraction methods. Moreover, middle-error voxels (yellow and orange voxels) are extremely effective in maintaining the existence and continuity of thin *features* inside low-res voxel models (e.g., limbs of examples in Fig.2.12 excluding the bunny). It seems that our method can maintain the existence and continuity of most

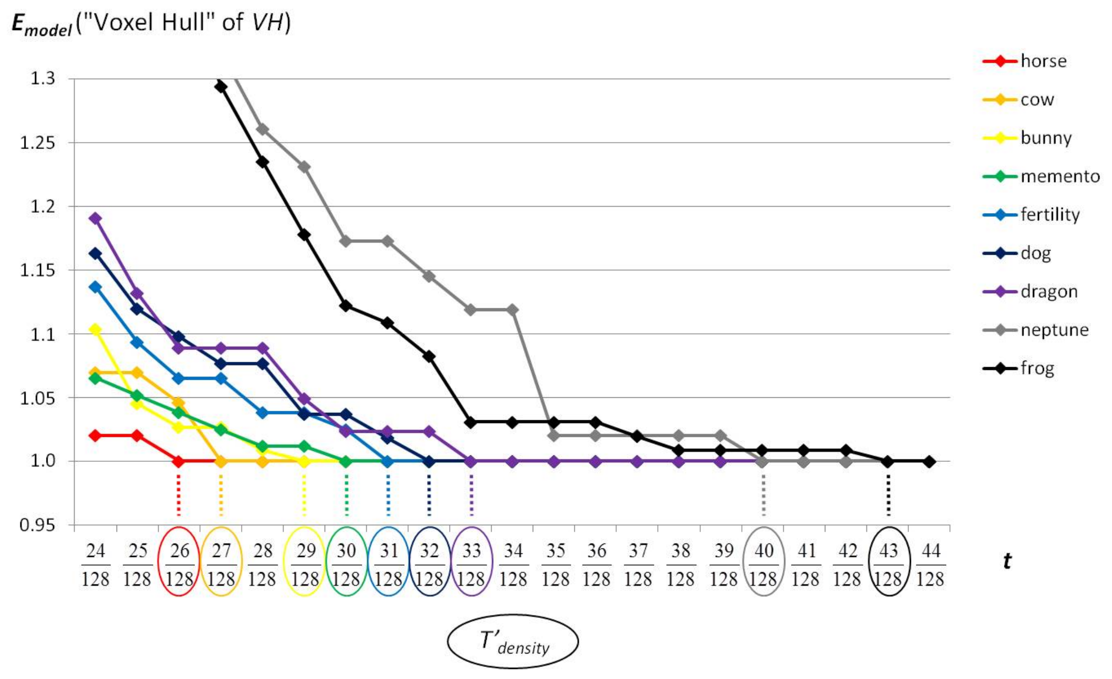


Figure 2.12: $T'_{density}$ calculated for different models in Fig.2.13. Each $T'_{density}$ is chosen as the t (discretized between 0 and 1 by 128), which starts to converge the E_{model} of "voxel hull".

features (except for extremely tiny ones like ears of the cow and horse) as well as balancing the *accuracy* and *abstraction* in shape.

2.6 Discussion of Application-Oriented Low-res Voxel Model

The automatic generation of low-res voxel model discussed in this chapter mainly considers some basic requirements on an abstracted shape represented in a low-res voxel space. However, additional requirements might be needed for different applications using the low-res voxel models. Embedding the generation of low-res voxel models into some better defined user scenario will make it more challenging or interesting even from a technical point of view. We can add more elements to the picture, e.g. per-voxel colors (starting from a colored mesh), physical constraints (constructability of a LEGO model, or in general other constrains to allow physical fabrication of a kind or another), imagined artistic creation (multi-resolution display, other voxel types instead of fully covered or fully empty), or, many possibilities, as long as the choices' justifications are rooted in something.

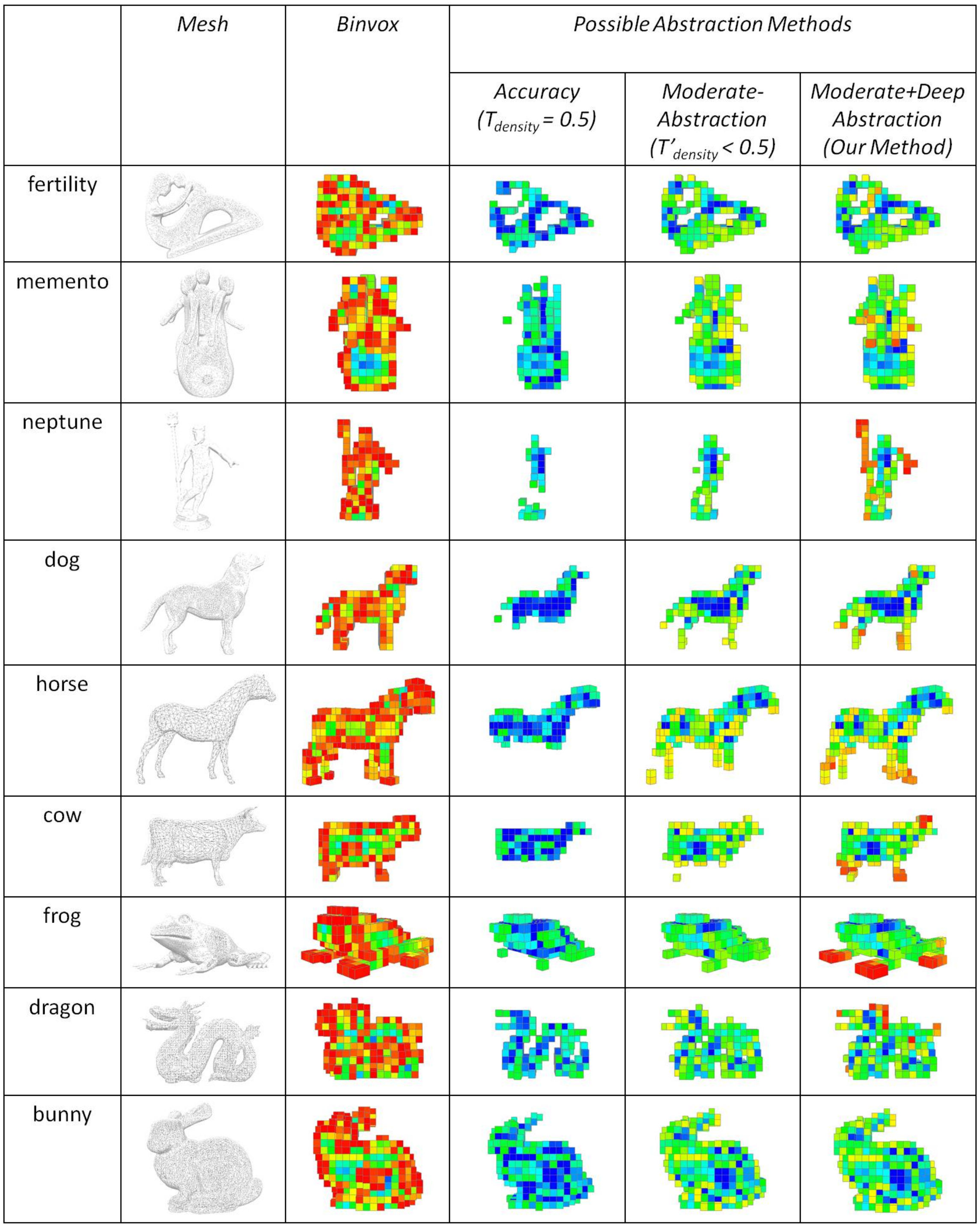

Mesh
Binvox
Possible Abstraction Methods
Accuracy ($T_{density}$ = 0.5)
Moderate-Abstraction ($T'_{density}$ < 0.5)
Moderate+Deep Abstraction (Our Method)
fertility
memento
neptune
dog
horse
cow
frog
dragon
bunny

Figure 2.13: Low-res voxel models generated in different ways.

In the next chapter, we will discuss the specific application of using low-res voxel model for our design of mini block construction.

Chapter 3

Generation of Artwork-like Block Model for Mini Block Construction

Mini block construction is a kind of well abstracted low-res block construction expecting aesthetically pleasing, artwork-like appearance and block layout. Similar with previous LEGO constructions, mini block construction requires strong interconnection among blocks, i.e., a stable layout. However, what make mini block construction 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 and smoothness of colors into abstraction. We further explore generation method for constructable layout satisfying both stability and symmetry to support our prototype design system. Mini block construction 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 construction.

The rest of this chapter is organized as follows. In Section 3.1, we will introduce some background knowledge on the generation of block model. In Section 3.2, we summarize our method of improving low-res voxel model in shape and color for mini block construction. In Section 3.3, for a low-res voxel model, we will further discuss how to generate block layout qualified for mini block construction. In Section 3.4, we show some results for evaluating our method.

3.1 Background Knowledge

We will introduce the background knowledge which includes the following two aspects: low-res voxel model generation for LEGO Models; layout generation for LEGO Models.

3.1.1 Low-res Voxel Model Generation for LEGO Models

Compared with voxelization [59] [69] [73], downsampling has more consideration for the generation of low-res discrete representation. However, research is still not mature in terms of downsampling in 3D low-res voxel space considering both the shape and the color, especially the color. As far as we know, currently, most low-res voxel models in academic use are automatically generated by the tool of Binvox [53]. However, color information is not handled. Results generated by Binvox [53] has been compared with our low-res voxel model generation method in Chapter 2, with our method demonstrated to be more suitable for results with accuracy and abstraction better considered.

A coupled handling of shape and color is preferred, but is challenging and application-oriented. Therefore, in most cases, handling of color is separated from processing of shape. When designing a LEGO model, basically block colors are limited. This restriction makes the handling of color more challenging. Quantization [18] [32] has proved to be effective in reducing the number of colors, and has been applied in our previous study [84] for coloring low-res voxel model. Additionally, clustering is also a straightforward choice. In one possible approach, one can simply map each 2D pixel (or 3D voxel) to its nearest color in the LEGO color palette [74] [73]. However, such naive method usually causes unacceptable artificial flaws: e.g., non-smoothness artifacts could easily be introduced when naively clustering colors in pixel art images. To reduce artifacts, Kuo et al. [33] formulated the problem of clustering colors in pixel art images as a color labeling that takes into account the color smoothness, and solved it using combinatorial optimization.

In automatically abstracted voxel model, there might probably be a kind of *problematic voxel*, defined as a badly placed or colored voxel resulting in a non-constructable block layout theoretically, i.e., causing disconnected parts which are also referred to as dangling parts [33]. These dangling parts can be inspected and handled along common boundary between two disconnected parts. In 2D pixel space, Kuo et al. [33] classify all the possible patterns within the local 2×2 window into four categories, and a set of pixel-level operations is proposed for each category to fix the dangling problem by altering the color of pixel(s) or adding new one(s) in the local window. Among all possible operations exhaustively searched, Kuo et al. [33] select the one that minimizes an energy function defined by themselves.

3.1.2 Layout Generation for LEGO Models

Finding a good block layout is a non-trivial and challenging problem [21], while accounting for not only shape and color in the given voxel model, but also constructability and stability in block structure.

For a long time, previous research [21] [63] [65] [73] [76] [80] have mainly focused on automatic optimization of block placement. Heuristic-driven merging which originated in a group of mathematicians [21], is a classical layout optimization method. A more stable block layout encourages classical factors [21] [65], 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). Different from the heuristic-based metrics for stability estimation, a force-based analysis (exampled by LEGO bricks) [50] for estimating physical stability of a given LEGO sculpture has also been applied in optimizing the block layout. However, due to the probable difference in block design, blocks of different brands might require different force models, which have not been discussed. Moreover, graph theory is applied in recent studies [57] [63] [73] and promotes another layout optimization method based on the detection and the repairing of flaws in layout. In summary, most recent studies optimize layout considering both heuristics and graph theory. A heuristic-driven random greedy merging algorithm [73] can be used to increase connections and larger blocks in the initialized layout. For constructability and more stable structure, they further use an iterative local random re-layout to cope with disconnected block groups and weak articulation points, both of 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 [21] 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.

3.2 Low-res Voxel Model Generation for Mini Block Construction

To better preserve shape features in the original mesh model, given a 3D mesh as input, we can use our method introduced in Chapter 2 to generate monochrome low-res voxel model. To satisfy the abstraction needs of different users, we provide two recommended voxel models (both with deep-abstraction appended, but filtered using different thresholds of $T'_{density}$, and $T_{density} = 0.5$), as well as allowing the user to adjust the threshold t for more customized abstractions.

For a low-res voxel model with abstracted shape, deciding the abstracted color is not easy. Because a low-res voxel model is largely transformed from its original shape, as observed in the head, legs and tail in Fig.3.1, 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 $3 \times 3 \times 3$ cube centered at this *empty voxel*). To calculate the color of *empty voxel*, each neighboring voxel is initially assigned a weight $w = 4d$, 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

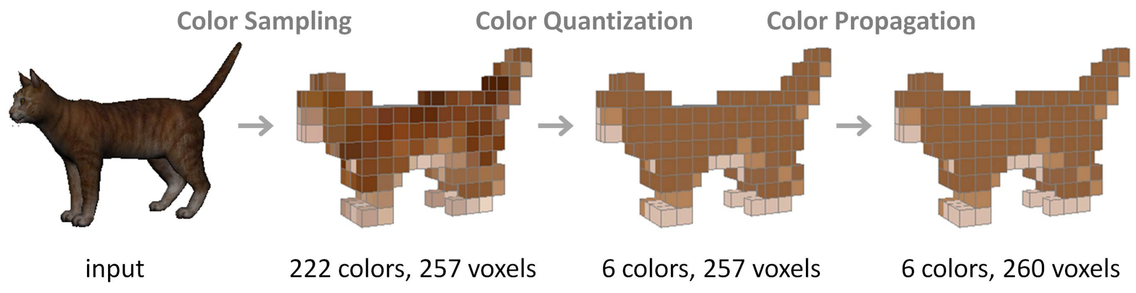


Figure 3.1: Automatic color processing flow for the cat in Fig.1.2. The numbers of valid colors and colored voxels in each voxel model are noted.

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.3.1, 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*.

3.3 Block Model Generation for Mini Block Construction

Like previous studies on block layouts in LEGO constructions, we also consider a standard LEGO brick family, consisting of bricks sized as $L \times W \times H (W = 1, 2; L = 1, 2, 3, 4, 6, 8; H = 1)$. We initially introduce some block layout processing algorithms in Section 3.3.1. After that, in Section 3.3.2, by combining these algorithms optionally, for a given low-res voxel model, we finally generate a block layout which is optimized considering constructability, stability and symmetry.

3.3.1 Layout Processing Algorithms

Conventional layout processing algorithms have been summarized by Testuz et al. [73] and Luo et al. [50]. Here we introduce our newly proposed three layout processing algorithms: *perpendicularly ordered merging*, *subpart connection*, and *layout symmetrization*.

3.3.1.1 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-res voxel model, the first seed voxel chosen to be merged (white rectangle in Fig.3.2) 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. [73], 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.3.2. 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.3.2.

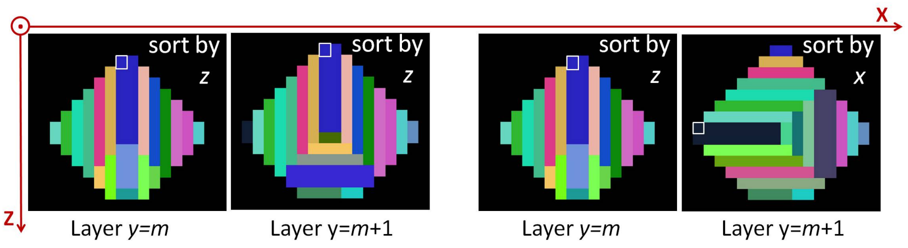


Figure 3.2: A comparison of layouts in successive layers generated by *naively ordered merging* (two layers on the left) and *perpendicularly ordered merging* (two layers on the right). The first merging seed voxel is marked by a white square.

3.3.1.2 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 conventional method involving locally-repeating random remerging [73], 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.3.3, 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.3.3 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.

3.3.1.3 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.

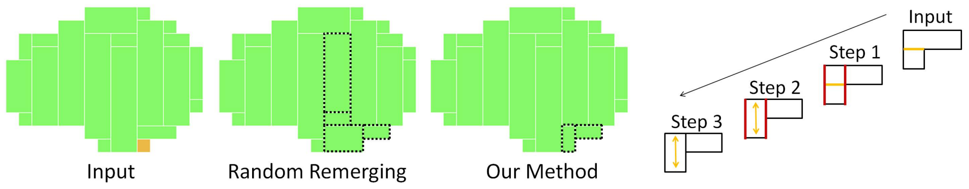


Figure 3.3: Combining subparts using random remerging algorithm [73] and our subpart reconnection algorithm. To reconnect orange block, by using three steps, our method results in less change in layout (black rectangles with dotted border).

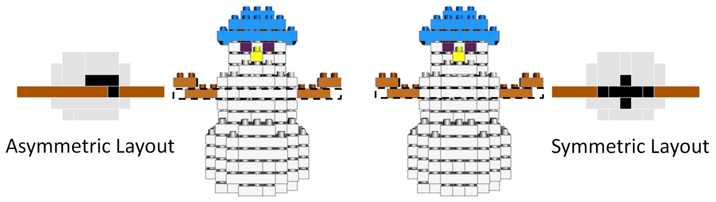


Figure 3.4: An asymmetric layout (left) is modified to a symmetric layout (right).

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.3.4. 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.

3.3.2 Layout Generation Method for Mini Block Construction

In this subsection, we introduce our generation method for constructable layout satisfying both stability and symmetry for mini block construction. 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.

3.3.2.1 Calculation of Stability for Mini Block Construction

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.

$$Stability = 1 - \frac{C_s/N_s + C_o/(1+N_o) + C_eR_e + C_uR_u + C_p/R_p + C_nR_n}{C_s + C_o + C_e + C_u + C_p + C_n}, \tag{5}$$

Our stability measure for each block is modified from that defined by Petrovic [65]. 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.(5) 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.(5) 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., $C_iR_i = 1$ for $\forall i$. We further define index i's discrimination as D_i, and then have $C_iR_i = D_i$, 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 [73], naively ordered merging and perpendicularly ordered merging in Section 3.3.1). To separate the effect of each index, we calculated the stability as Eq.(5) 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).

3.3.2.2 Proposed Layout Optimization

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 [73] 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 [73] 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* $L1$ times ($L1 \leq 5$). If it fails in constructability, the smaller block beside each *separating section* is split smaller for extra $L1$ 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. [73]

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.

3.4 Results and Discussion

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., Lego_Man, headphone, and sunglass), were taken from free sources available online. Binvox [53] was used for low-resolution voxelization. For coloring, we compared our algorithm with naive alternatives. For layout generation, we compared our method with a similar previous method [73].

3.4.1 Coloring

To evaluate our system based on quantization and sampling, we tested meshes with texture and surface color. Table 3.1 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.3.5 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.

Table 3.1: 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

3.4.2 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.3.6 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.3.6 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: conventional algorithm of random greedy merging [73], and two of our layout merging algorithms (naively ordered and perpendicularly ordered introduced in Section 3.3.1). Fig.3.6 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.

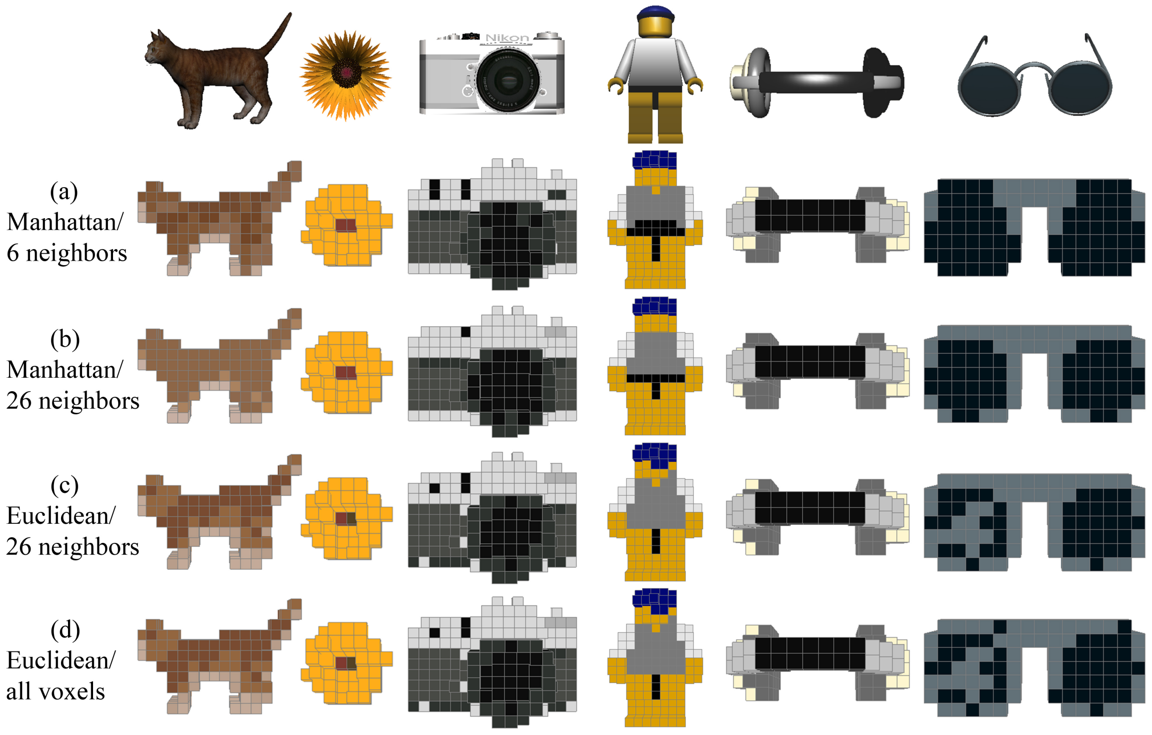


Figure 3.5: Automatically abstracted voxel models without manual editing, considering three naive alternatives for color sampling strategy.

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 3.2) 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.

3.4.3 Layout Optimization

In regards to a comprehensive optimization for constructability and symmetry, we compared our layout generation method discussed in Section 3.3.2 with the method of Testuz et al. [73]. In our experiment for the models in Table 3.3, 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 method of Testuz et al. [73]. Though symmetrization is normally at the cost of stability, we can also find that almost half of our optimization results (rows in Table 3.3 from "dolphin_asym" to "dolphin_sym") exhibite larger stability than those of the method of Testuz et al. [73]. Some intuitive layout comparison can be viewed in Fig.3.7.

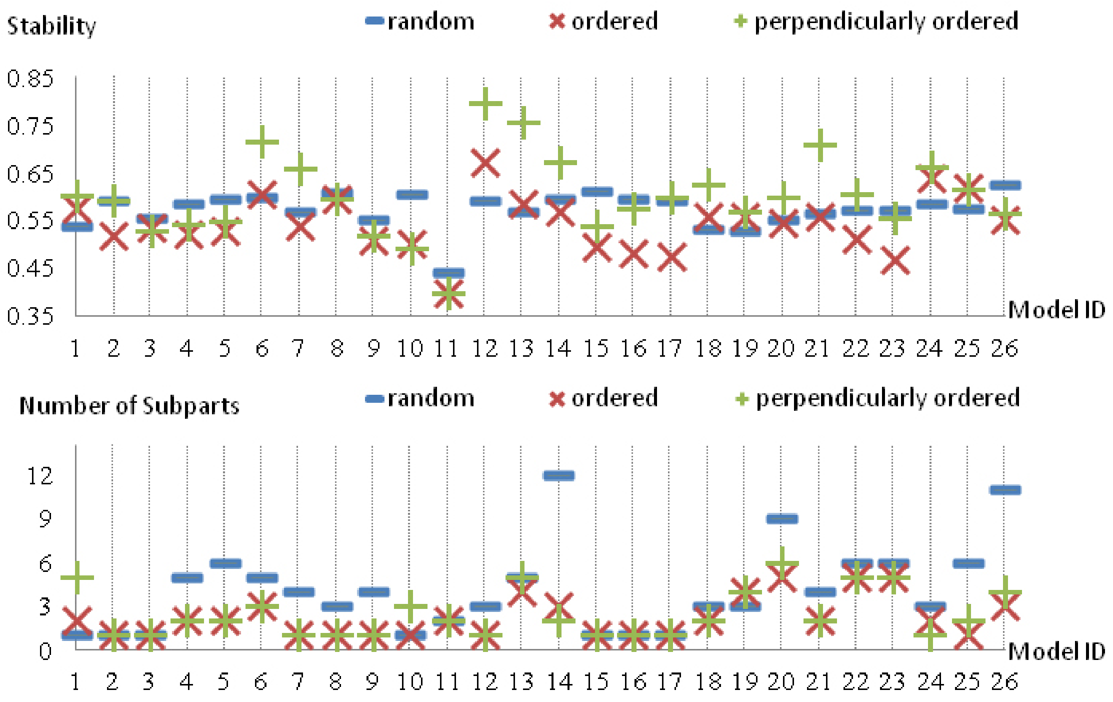


Figure 3.6: Automatically abstracted voxel models without manual editing, considering three naive alternatives for color sampling strategy.

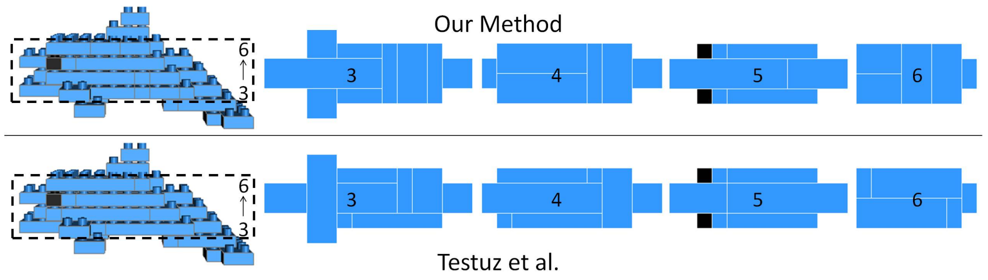


Figure 3.7: Automatically abstracted voxel models without manual editing, considering three naive alternatives for color sampling strategy.

Table 3.2: 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
Lego_Man_sym	0.584, 5	0.535, 2	0.535, 2			0.551, 3
Lego_Man_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

Table 3.3: 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
Lego_Man_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
Lego_Man_sym	0.550	0.583	0.567	1.000, 0.891	1.000, 0.647	0.714, 0.684

Chapter 4

Component-based Building Instructions for Assembly of Mini Block Construction

A less breakable LEGO sculpture hopes for thickness throughout the model. To ensure enough thickness at thin part, a LEGO sculpture is prone to be designed in high resolution. However, for mini block construction which is normally designed in low resolution, it is easy to be fragile at some spots because only a small number of blocks can be used.

A LEGO sculpture with fragile constructions of blocks might easily fall to pieces during assembly. For an enjoyable assembly time, a well-designed set of building instructions is crucial. Although several studies related to LEGO exist, most of them are focusing on designing block structures [25] [33] [50] [73] [84]. In this chapter, we focus on the assembling order of blocks without adding any modification to the structures of target model.

The rest of this chapter is organized as follows. In Section 4.1, we will introduce some background knowledge on different kinds of assembly tasks, including block assembly and some other general assembly tasks. In Section 4.2, we summarize our method proposed for the automatic generation of component-based building instructions. In Section 4.3, we show some results of our method and a user study for evaluating our generated building instructions.

4.1 Background Knowledge

Assembly tasks are common in manufacturing industry, especially for making machines. To facilitate the manufacturing industry, traditional design for products basically ensures that, a multigraph data structure [22] for parts is maintained in the conceptual product model. Such a data structure is powerful for generating smart views to uncover occluded but important parts, therefore can effectively reduce the burden of assembly. Generation of smart views (e.g., cutaway views [44], ghosted views [14], or exploded views [43]) belongs to illustration techniques [77]. Beside of smart views, other illustration techniques for assembly can also produce step-by-step illustrations [2] , or illustrations displayed by projector [15] or AR equipment [79].

With the development of automatic or semi-automatic fabrication-aware design, fabrication becomes less limited in manufacturing industry, but more in daily life. Most of these designs have no clearly defined multigraph data structure. Typical fabrication targets can be furniture [35], interlocking 3D puzzles [47] [82], papercraft toys [55], plush toys [56], beady toys [28], and LEGO constructions [73] as well. The popularizing of rapid prototyping even increases the possibility of such fabrication [6]. However, different kinds of fabrication face different characteristics in structure. When higher quality and higher efficiency are required for the fabrication, the assembly might also be more challenging.

In this section, we will review the block assembly, and then introduce some academic findings for designing building instructions for a general assembly task.

4.1.1 Block Assembly

A commercial block product generally provides a manual for users to assemble it smoothly. Virtual models of completed LEGO products are built concurrently with the writing of the user instructions. Instructions in LEGO manuals can be observed to be designed following certain rules.

1. As shown in Fig.4.1(a), LEGO manuals essentially contain a series of figure-based static illustrations, corresponding to four typical types of views: package view, group view, step view, block-list view.

2. The inner structure of a 3D block model is mainly illustrated by the step view, which is similar with a cross-sectional view (e.g., cutaway illustration [44] and exploded views [43]).

3. As shown in Fig.4.1(b), during the assembly of a 3D block model, step views do not always share the same view angle.

4. As shown in Fig.4.1(c), arrows of different colors are important in correctly guiding the building operation. Generally, a black arrow means to insert the block group shown inside the rectangle linked with this arrow; a red arrow means to align or connect two pieces; two blue curved arrows forming a circle means upside down or rotation.

5. As shown in the left image of Fig.4.1(d), in a large LEGO project, blocks assembled in current step are highlighted as pieces with yellow borders.

6. As shown in the right images of Fig.4.1(d), similar block pieces easy to be mistaken will be tagged with number, or put together for comparison.

Manual generation of building instructions requires design experience. Generation of a user-friendly building instruction might require interactions among several tools for many subtasks. A series of specialized applications (e.g., the modeling tool of MLCAD [34], the model displaying tool of LDView [7], and the page layout tool of LPub [66] for building instruction) use the LDraw [62] open-source library to represent block parts in 3D. These applications serve a large user community. Moreover, to facilitate the manual task, a convenient user interface is essentially required for free control of a block model (e.g., pose changing, viewing/hiding a block). Therefore, some generalized tools for modeling (e.g., BrickSmith [70], Sketchup [45], and SolidWorks [10]) are also shared among LEGO fans. These tools support the manual creation of block building instructions as well.

Tools for automatically generated instructions mainly suffer from the lack of grouping rules thought over by experienced designers, which is the biggest flaw in assembling mini block construction that is fragile. Moreover, these tools basically support a step-by-step ordered simulation (allowing rollback to earlier steps); however, they are still not well considered for efficient and user-friendly assembly. LEGO Instruction Creator [16] and Testuz et al. [73] use a naive bottom-up building order calculated layer-by-layer, which is common in block designing tools. However, such a naive approach cannot handle the floating blocks illustrated in Fig.1.4. LEGO Digital Designer [36] is the free tool officially provided by LEGO, enabled for the manual design of a LEGO construction and the automatic simulation of assembling it. The automatic simulation considers the connection to earlier-built blocks and a change in viewing angle to ensure that each upcoming block is visible. However, such a simulation adds only one block in each step, and the added block might frequently come in an unpredictable direction, making the assembly not user-friendly enough.

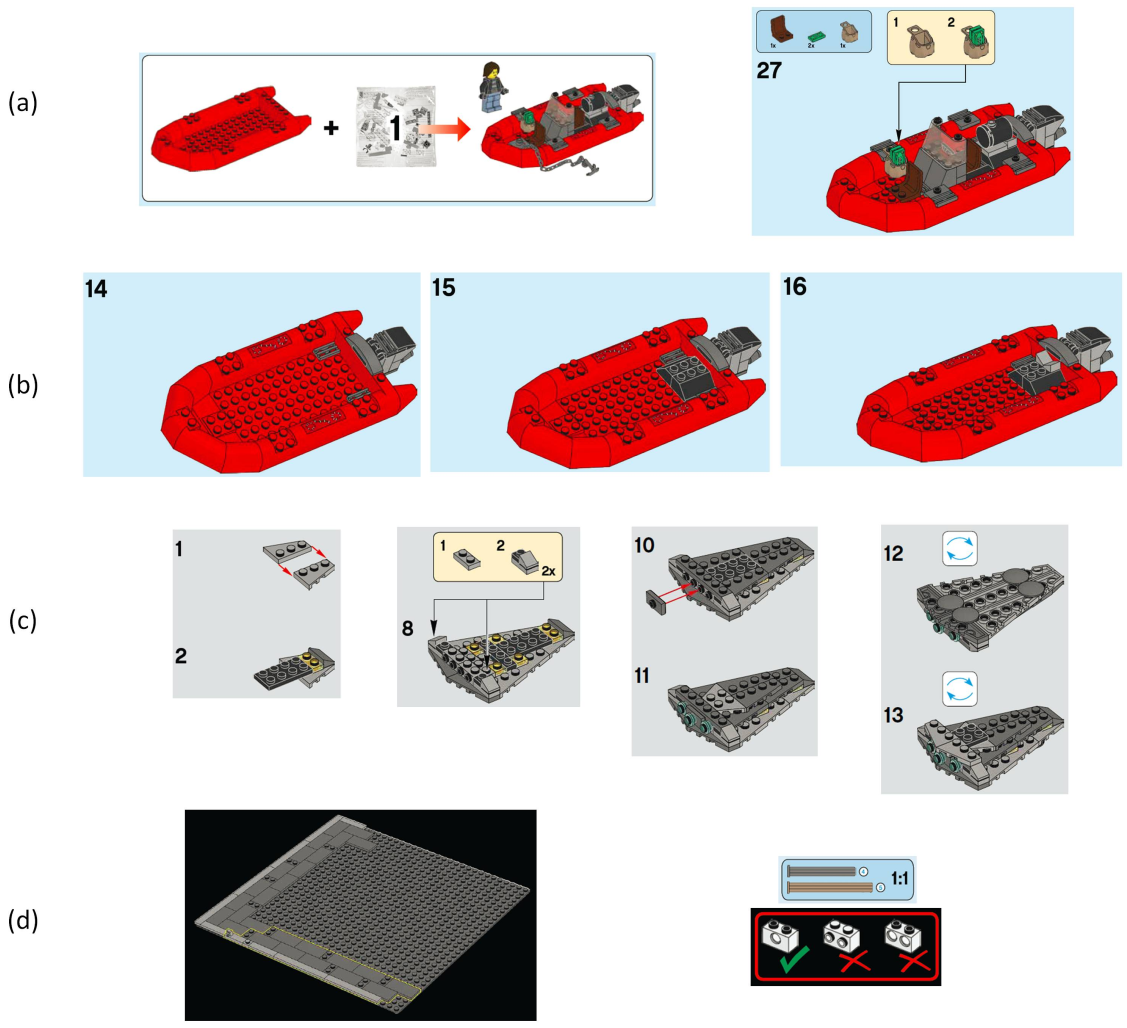


Figure 4.1: Observations on LEGO Manuals [39]. Images are excerpts from manuals of LEGO 60129, LEGO 30277, LEGO 75827, LEGO 42048, LEGO 21026.

Block assembly can also be guided in new style. Gupta et al. [23] proposed a novel Kinect$^{®}$-based augmented system for guiding block assembly. Unlike conventional systems using a block model as input, this augmented system requires the troublesome tracking of a designer's real-time modeling to generate a building guide.

4.1.2 Academic Findings for Designing Building Instructions

To produce visually comprehensible and accessible instructions for different assembly tasks (e.g., block assembly, furniture assembly), Heiser et al. [24] and Agrawala et al. [2] [1] have investigated a series of design principles through cognitive psychology experiments. Among these design principles, one refers to the hierarchy and grouping of parts. For example, disjoint parts are more likely to be segmented [2], and parts are also typically grouped (e.g., the legs of a chair or the drawers of a desk) [75]. As covered by Agrawala et al. [2], it is preferable that parts within a group be assembled simultaneously or continuously.

Another important design principle discussed by Agrawala et al. [2] involves the hierarchy of attachment, which is required at the higher levels for combining separate subassemblies and at the lowest level for attaching smaller parts to the more significant parts. The significance of a part depends on a number of factors including function, size, and symmetry.

In addition to discussing theoretical design principles, Agrawala et al. [2] developed algorithms to evaluate these design principles. Their experiments for different assembly tasks have shown that design principles are extremely useful for choosing a user-friendly sequence of assembly operations.

The design principles mentioned above are for general assembly tasks. However, for a specific type of fabrication, domain-specific knowledge is required, especially when automatically predicting the hierarchy and grouping of parts. Mesh segmentation [68] typically focuses on surface decomposition driven by geometric properties. For fabrication by 3D printing [49], basic requirements are the printable part size and the assemblability. In this paper, for block assembly, we propose a method to automatically generate layout-driven assembly parts in a block model.

4.2 Proposed Method

We aim at the user-friendly assembly of fragile block models, i.e., block models with weakly-connected blocks. The input of our method is an assembled shape of a block model, which can be easily obtained using existing LEGO design software. All blocks are assumed to be rectangular solids having the same height, similarly to the previous methods, e.g., [73]. The output of our method is a step-by-step set of 3D instructions which can be viewed from any angle. Firstly, we will introduce a method for dividing a model into components in Section 4.2.1. Then we will introduce a method for generating building instructions by deciding the assembly order of the components in Section 4.2.2.

4.2.1 Generation of Components

We define a *block segment* as a set of blocks treated as a basic element for generating component. To facilitate operations (e.g., intersection, union) among basic elements, each basic element requires blocks inside to be interconnected as one. Our basic approach for generating components is to initially segment the input model into *block segments* deliberately and then make components by merging some unnecessarily-small *block segments.* For the segmentation, our aim is twofold: to separate the input model at weakly-connected blocks; and to separate the input model to eliminate *floating blocks.* These two types of segmentation are implemented independently, and are described in Section 4.2.1.1 and 4.2.1.2 respectively. In Section 4.2.1.3, we describe how the *block segments* generated by both segmentations are merged into components.

4.2.1.1 Segmentation at Weakly-Connected Blocks

We detect weakly-connected blocks as blocks corresponding to the previously defined "weak articulation points" [73]. Note that a block model can be abstracted as a graph, where the vertices represent individual blocks and the edges indicate brick linkage by studs. An articulation point in graph theory is such a vertex that when removing it the graph generates more than one disconnected subgraph. For a block model, to identify important articulation points, Testuz et al. [73] define a "weak articulation point" as an articulation point that connects each subgraph owning a size (number of edges in subgraph) greater than one.

When designing a block model, previous research [73] detected "weak articulation points" for reducing them in optimized model; however, not all "weak articulation points" can be avoided when thin part exists. In this paper, we find "weak articulation points" as weakly-connected blocks to help us to divide a model into *block segments.* Definition of "weak articulation point" [73] decides that, by removing each weakly-connected block, the model can be separated into multiple disconnected parts, with

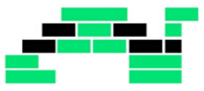
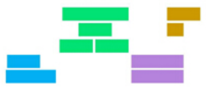
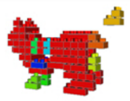
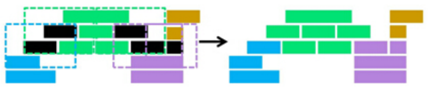
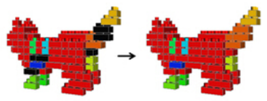
	Operation	*2D Illustration*	*3D Illustration*
Step 1	Detect weakly-connected blocks *W* in input model *M*.		
Step 2	Divide *M* into several *block segments* by subtracting *W* from *M*.		
Step 3	Merge each block $w \in W$ into a *block segment* that has the largest number of connections to *w*.		

Figure 4.2: Algorithm and illustrations for segmentation at weakly-connected blocks (black blocks).

each part containing more than one block. Inspired by this property, in our segmentation (see algorithm in Fig.4.2), all the weakly-connected blocks detected in Step 1 are removed from the initial model in Step 2. Then in Step 3, each weakly-connected block is merged into a *block segment* that has the largest number of connections to the block.

4.2.1.2 Segmentation Avoiding Floating Blocks

We first extract the blocks that will be in the floating state during a layer-by-layer, bottom-up assembly. Such *floating blocks* are easily detected as follows. As shown in Fig.4.3(a), we visit connected blocks from each bottommost block to the top. The allowed visiting direction is only upward, because in a LEGO model two blocks are directly connected to each other only if they overlap each other. The blocks that have not been visited by the end are *floating blocks*. Fig.4.3(a) illustrates a simple LEGO model in 2D with *floating blocks* (colored in red).

This process for detecting *floating blocks* uses basically a breath-first search algorithm. By default, as shown in Fig.4.3(a), there is only one bottom, hence the search starts from all the bottommost blocks (i.e., initial search keys) and travels through the whole model. If allowing one more bottom, as shown in Fig.4.3(b), the original model is segmented, causing an independent search inside each *block segment*.

Our final target is to ensure that there are no *floating blocks* in each component generated. Here, we introduce two strategies to achieve this goal. One is a direct way, the other indirect. The direct way is to explicitly separate the *floating blocks* as components. Note that, in Fig.4.3(a), if we treat the *floating blocks* (colored in red) and the rest (colored in green) as two different components, each component contains

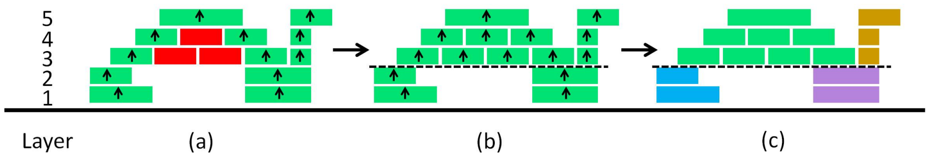


Figure 4.3: Segmentation avoiding *floating blocks*. (a) Detecting *floating blocks* (red blocks) along arrows. (b) A *pseudo floor* is inserted between the 2nd and 3rd layers. (c) The model is divided into four segments.

no *floating blocks* inside. On the contrary, the indirect way is separating the model by horizontal planes until no *floating block* exists as illustrated in Fig.4.3(b). The dashed line shows the horizontal plane used to separate the model into components. This separation works as if we had inserted a working floor between the 2nd and 3rd layers. We call this separating plane a *pseudo floor*. In Fig.4.3(c), due to the separation by the *pseudo floor*, four independent components are obtained, and each can be assembled from the bottom to the top without any *floating block*. So we know that we have eliminated the *floating blocks* by inserting a *pseudo floor* at an appropriate location. In Fig.4.3, such a location is between the 2nd and 3rd layers. However, sometimes one *pseudo floor* might be not enough to eliminate all the *floating blocks*; we might require more, or in an extreme case, one *pseudo floor under each layer*.

Now we have two strategies to generate components with no *floating blocks* inside. We further combine both strategies to reduce the amount of *block segments*, which will benefit a more precise instruction. We do so by applying indirect way first, however, not for eliminating all *floating blocks*, but for reducing *floating blocks* reasonably by inserting a few effective *pseudo floors*. After that, we use the direct way to handle the unreduced *floating blocks*.

The problem now becomes how to select effective *pseudo floors*. We find that inserting a *pseudo floor* at an appropriate location (e.g., between the 2nd and 3rd layers in Fig.4.3) is important. This location is important because if a *pseudo floor* is inserted elsewhere, *floating blocks* below the *pseudo floor* cannot be eliminated. To find such an appropriate location, we introduce an index $C_{floating}(l)$, which equals the number of *floating blocks* had by inserting a *pseudo layer* between the l-th and $(l+1)$-th layers for $l = 0, 1, 2, ...$ ($l = 0$ means the ground floor). By evaluating the value of $C_{floating}$ for all possible l, we choose to insert *pseudo floors* where the value of $C_{floating}$ takes on local-minima. We do so because each local-minimum of $C_{floating}$ indicates a horizontal separation which can better reduce *floating blocks* in the local

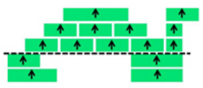
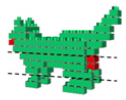
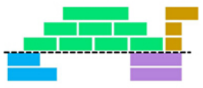
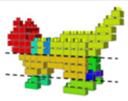
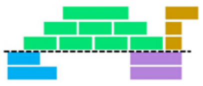
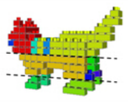
	Operation	*2D Illustration*	*3D Illustration*
Step 1	For each l, calculate a $C_{floating}(l)$ by pre-inserting one horizontal *pseudo floor* between the l-th and $(l+1)$-th layers. Then insert *pseudo floors* where $C_{floating}(l)$ takes on local-minima.		
Step 2	Indirect segmentation: Divide the input model by the inserted *pseudo floors*.		
Step 3	Direct segmentation: detect *floating blocks*, and then explicitly separate them as new *block segments*.		

Figure 4.4: Algorithm and illustrations for segmentation avoiding *floating blocks*.

area around the *pseudo floor*. By applying all these separations at the same time, the initial model is segmented into several *block segments*. The algorithm described above is summarized in Fig.4.4 with 2D and 3D illustrations. Note that all *floating blocks* are not eliminated always by Step 1. The red blocks in the top row of 3D illustration are *floating blocks* when the model is separated by two *pseudo floors* (illustrated by dashed lines). On the other hand, no *floating blocks* exist in the example illustrated in 2D after inserting a *pseudo floor*. Therefore, there are no differences between middle and bottom rows of 2D illustrations.

These separations caused by this algorithm result in unnecessarily-small *block segments* (e.g., the brown and the purple *block segments* in 2D illustration in Fig.4.4, these *block segments* can be combined without generating any *floating block*). In next subsection, we will describe a strategy to adjust these over-segmentations.

4.2.1.3 Making Components

The initial model is segmented using the two approaches mentioned above. Based on these segmentations, the components are generated. As described in the algorithm in Fig.4.5, we first apply the two segmentations to the model (Step 1). Then we generate components by dividing the model along the boundaries of the segmentations (Step 2). If *floating blocks* remain, we separate them as individual components. Because this generates tiny components, we merge them to reduce the number of components (Step 3). This step is divided into following four sub-steps.

1. Find a component "A" which touches a *pseudo floor*, or contains only one or two blocks;

2. Find a component "B" which connects component "A";

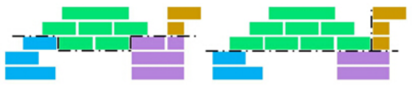
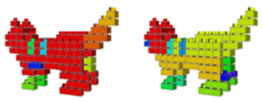
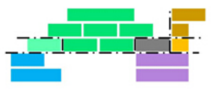
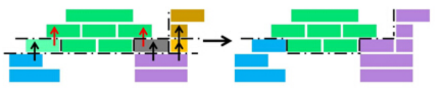
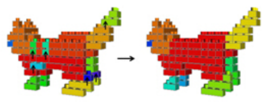
	Operation	*2D Illustration*	*3D Illustration*
Step 1	Apply the two segmentation algorithms to the model: driven by weakly-connected blocks (left), and avoiding *floating blocks* (right).		
Step 2	Generate components by dividing the model along the boundaries in both segmentations generated by Step 1.		
Step 3	Merge components as much as possible, ensuring no *floating block* is generated.		

Figure 4.5: Algorithm and illustrations for making components from *block segments*. In Step 3, black arrows upward indicate successful merging. We also show some failed merging indicated by red arrows in 2D illustration.

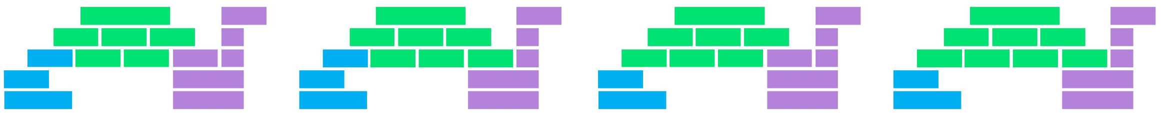


Figure 4.6: Different results for components containing no *floating block*.

3. Merge component "A" and "B" only if the merged component does not generate additional *floating blocks*;

4. Repeat Sub-step 1 to 3 until all possible merge operations are done.

Fig.4.5 illustrates our way to obtain components containing no *floating block*. However, theoretically, other results for components (see Fig.4.6) can achieve the same goal if other features (e.g., recognizability of a component, equilibrium of a component) are not considered. In the future, improvements can be made to satisfy more beneficial features in a component.

4.2.2 Making a Component-Driven Instruction

After the LEGO model has been divided into components, we start to make an instruction guide for assembly. By now it is ensured that no *floating blocks* exist in any component. Hence, we can simply assemble each component in bottom-to-top order, and focus only on the order of combining components. Among the components, we define a *joint component* as one connecting two or more other components. The assembly order of components is decided according to the following priorities:

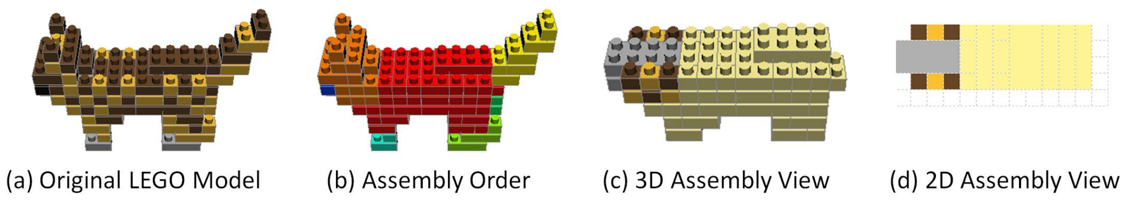


Figure 4.7: Our graphical instruction guide.

1. the number of connected components;
2. the number of blocks contained in the component;
3. the number of connected *joint components*;
4. the distance from the bottommost block (smaller has higher priority).

If value 1 is the same for each component, value 2 is used to decide the priority. Furthermore, if value 2 is the same for each component, value 3 is used, and so on. Finally, if symmetrical components-pairs exist, the order of components is further adjusted to ensure successive assembly of such symmetric component-pairs.

After deciding the assembly order of the components, we generate a graphical instruction guide. To prepare the user for the assembly flow, the guide firstly switches from the original LEGO model (Fig.4.7(a)) to the completed model showing colors assigned from blue to red to the components according to their priority (Fig.4.7(b)). After that, the user begins assembling the first component (the red one in Fig.4.7(b)). The assembly procedure of each component is displayed in an interactive 3D view (Fig.4.7(c)) and a static top-view (Fig.4.7(d)). Both views are simultaneously updated step-by-step. In both views, blocks in the active component (the component being assembled) are rendered in the original color, but already assembled components are rendered in a customized color (beige in Fig.4.7(c,d)). Showing the already assembled components with the active component helps users to understand their relative positions. Visibility of blocks during assembling process is important. Because each component generated in our method can be assembled layer-by-layer, a 2D view which shows blocks in current assembling layer (Fig.4.7(d)) always ensures the visibility of all blocks, even though these blocks might be occluded in 3D view.

4.3 Results and Discussion

We developed a prototype system to evaluate our method. It 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.

As far as we know, benchmark containing different block models (e.g., block models designed in different resolutions, block models fragile to varying degrees) has not been discussed before. To build such a benchmark is not easy. In this paper, our proposed method is mainly designed for fragile block models. Because the fragile structure is normally found in block models designed in low resolution, low-resolution block models are used to evaluate our proposed method. We prepared seven low-resolution block models (see Fig.4.8) created by a mini block artwork design system [84]. These test examples are fragile to varying degrees, i.e., weakly-connected blocks in these block models are counted differently (see Table 4.1).

4.3.1 Generation and Ordering of Components

Segmentation in our method is driven by weakly-connected blocks and *pseudo floors* found in input model. As shown in Table 4.1, the number of weakly-connected blocks ranged from 0 (camera) to 14 (sunglasses) in our test models. After the segmentation at weakly-connected blocks, the number of *block segments* ranged from 1 (camera) to 12 (cat). Fig.4.9 shows the graph of $R_{floating}(l)$, which is the normalized value of $C_{floating}(l)$ divided by total number of blocks in the model so that it takes between 0 to 1. For example, if no *floating block* exists when a *pseudo floor* is inserted between l-th and $(l+1)$-th layers, $R_{floating}(l)$ takes zero; if half of all blocks are floating, $R_{floating}(l)$ takes 0.5. By observing the graph, we can find that most test models (except for Lego_Man) have only one local-minimum or two local-minima. Detailed information is shown in middle column of Table 4.1. Although both segmentation steps result in unnecessarily tiny components (Fig.4.8(a, b)), our merging strategy successfully combines tiny components into large ones for better results (Fig.4.8(c)). Details can be found in right column of Table 4.1.

Note that Fig.4.8 reveals an important feature of our component generation method: segmentation along the horizontal *pseudo floor/floors* might be locally unwise sometimes (see sunglasses, dolphin and cat in Fig.4.8(b); however, segmentation along horizontal *pseudo floor/floors* is able to be revised locally, because a wise remerging is possible due to a wise segmentation of existing *block segments* at local disjunctions. Currently, local disjunctions are identified by weakly-connected blocks in fragile block model. In the future, for a block model being not fragile, other efficient segmentation methods can also be easily integrated into our current method.

Assembly order of components determined by our method is illustrated in Fig.4.8(c) as well. As expected, *joint components* are to be built earlier (in a warmer color), and symmetric component-pairs to be built successively are in similar colors. This demonstrates the effectiveness of our ordering.

Table 4.1: Statistics of test models.

Test Model	Segmentation at weakly-connected blocks		Segmentation avoiding floating blocks		Making Components	
	# of weakly-connected blocks	# of block segments generated (Fig.4.8(a))	# of pseudo floors (local-minima)	# of block segments generated (Fig.4.8(b))	# of components by overlapping block segments	# of components after merging (Fig.4.8(c))
sunglasses	14	7	1	5	11	7
flower	12	4	1	4	6	6
dolphin	1	2	1	5	6	5
cat	7	12	2	16	20	10
headphones	10	3	2	11	13	11
Lego_Man	5	4	3	6	8	6
camera	0	1	1	3	3	3

4.3.2 Auto-generation of Instruction Guide

Table 4.2 compares the instruction guide generated by our method with those generated by LEGO Digital Designer [36] and LEGO Instruction Creator [16]. We used in our test the cat model shown in Fig.4.8, which consists of 93 blocks and 7 weakly-connected blocks in total. On the one hand, we found that the instruction guide generated by LEGO Instruction Creator showed some steps with *floating blocks*, while our layer-by-layer assembly inside each component avoided *floating blocks*. On the other hand, we found in the test that the assembly starting from the feet was breakable, because earlier-built weak parts of the feet interfered with the smooth assembly of the rest. However, unlike the other two systems which did not generate separate components for the feet, in our instruction guide, the assembly of the separate foot components was near the end, and thus, it seldom affected the assembly of other components.

We recruited four undergraduate volunteers to test the time efficiency of the instruction guides generated by our system and LEGO Digital Designer. The results showed that all subjects completed the cat model in much less time when using our instructions. The average time needed to complete the model with our instructions

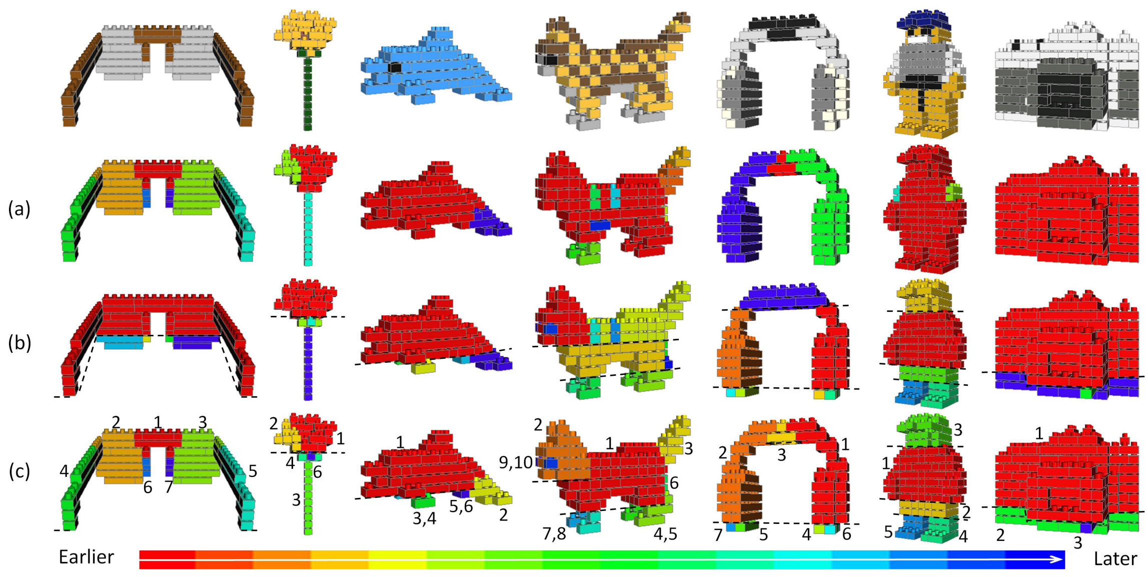


Figure 4.8: Generation and ordering of components in different test models (top row). (a) *Block segments* separated at weakly-connected blocks. (b) *Block segments* avoiding *floating blocks*. (c) Final components generated. Assembly order of final components, as marked by numbers, is associated with a specific color in a color map varying from red (built first) to blue (built last).

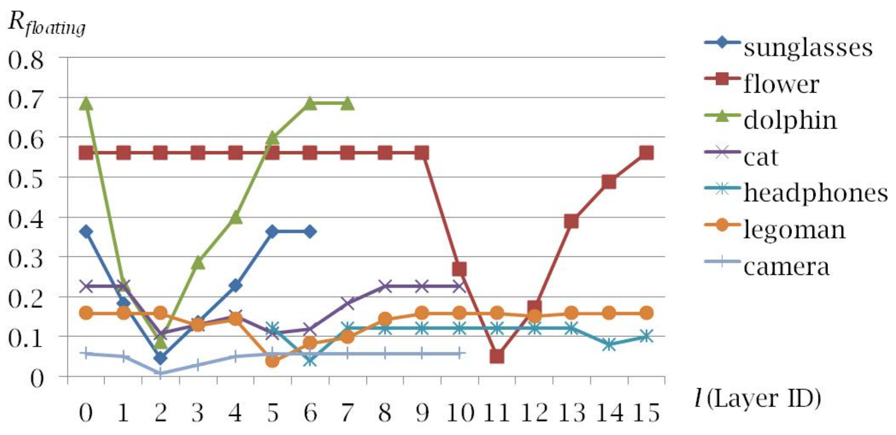


Figure 4.9: Graph of $R_{floating}(l)$, which takes ratio of *floating blocks* in whole blocks.

Table 4.2: Step-by-step instructions created by three systems.

	# of components	Max/Min # of blocks in component	Instructions steps for component	With first block for
Our system	10 (see Fig.4.8)	53/1	layer-by-layer	body
LEGO Digital Designer	2 (body & tail)	89/4	block-by-block	foot
LEGO Instruction Creator	1 (whole)	93	layer-by-layer	foot

was 17 min, which is about 60% of the time needed with LEGO Digital Designer's instructions.

Chapter 5

Conclusions and Future Work

By now, we have introduced a lot on mini block construction. The mini block construction we hope to create can be treated as a mini LEGO construction designed in a *simple* way, i.e., by only using block pieces in a standard LEGO brick family. As a great British writer (William Golding) said, “the greatest ideas are the simplest”. There are many merits in popularizing such a *simple* mini block construction. For example, for general people, building mini block construction saves block resources and players’ time; for researchers, observing original designs of mini block constructions might help to unravel the secret of how people abstract an object. Moreover, mini block constructions designed by different people are extremely comparable due to the regularity in block pieces used. It makes mini block constructions extremely fit for academic use, e.g., building a benchmark for evaluating different algorithms.

To encourage more people to join in the creation of mini block constructions, supplying a computer-aided tool is not a bad idea. This chapter concludes our main efforts made for building such a tool, as well as discussing the limitations and the future work.

5.1 Conclusions and Limitations

In this paper, we proposed several automatic solutions for different steps in the computer-aided creation of mini block construction. Firstly, we can automatically generate a monochrome low-res voxel model which is reasonably abstracted from an input mesh model. Secondly, we can automatically color the voxel model and revise its shape, color and block layout for a block model. Thirdly, we can create a user-friendly building guide to help converting a virtual design step-by-step into a tangible block construction for fun.

In the first step of voxel model generation, given a target to be designed, we automatically generate a low-res voxel model which is downsampled from a high-res voxel model pre-voxelized from the target. To balance *accuracy* and *abstraction*, a shaped-related voxel attribute of *density* is calculated for each voxel in low-res voxel space, and is further used for deciding shape in low-res voxel model. During this step, a downsampling of color has not been applied synchronously, because the abstraction of color should additionally take into account the LEGO palette to reduce the total number of colors in input model, making the processing for color much more complicate.

In the second step, for generating a block model, we consider a coloring and a revising of low-res voxel model in shape, color and block layout. Coloring a low-res voxel model proposed in this paper can satisfy some basic requirements; however, it is still challenging to realize the automatic abstraction for some complicate color patterns in input models, such as the dotted pattern in Fig.1.2(c). For layout generation, we achieve a high-quality block layout for low-res voxel model. During our block layout generation, not only the stability, but also the symmetry is considered to satisfy the delicate design of a mini block construction qualified for artwork.

In the third step of generating user-friendly building instructions, we get over the difficulty of fragile structure in low-res block model, and focus more on people's assembly habits. To avoid fragmentation during assembly, we proposed a solution for automatic generation of component-based building instructions. Components are obtained considering segmentation at both the weakly-connected blocks and the incoherent spots identified by *floating blocks*. Due to the generation of these well-considered components, block assembly habits of "from bottom to top" and "layer-by-layer" can now be combined in a user-friendly way.

We implemented our proposed method and developed a prototype system. To evaluate our proposed method, we compared with conventional techniques like, a popular tool [53] for generating voxel models, a widely referenced method [73] for generating block layout, and a LEGO official tool [36] for block assembly. To explore more user-friendly choices for our developed prototype system, we have also conducted two user studies.

5.2 Future Work

In the future, to improve the quality of automatic abstraction in shape and color, the effectiveness of saliency information in model can be explored. If a benchmark containing well-abstracted examples can be prepared, we can also try the technique of deep learning [19] to explore more all-powerful methods for abstraction. A recent study of Wu et al. [81] has led to demonstrate that voxelized geometric shapes in their 3D CAD model dataset can be used for deep learning, and learned data can be successfully used as shape features for classification, retrieval, and recognition. Therefore, in the future, by learning a benchmark of well-abstracted voxel models, useful abstracted features can be expected to be obtained, as a support to the abstraction of other models similar with those in the benchmark.

In order to enable our prototype system with practical use, a graphic user interface supporting a convenient surface editing on voxel model should be considered. Considering the limitations of the-state-of-the-art techniques, there might be a long way before achieving an automatic abstraction with stable performance, especially for abstracting from an input model with complicate shape features, color features or even more complex texture map features [83]. To make up for the limitations in current automatic abstraction, an efficient voxel-level editing can be powerful in low-res voxel space. In our current system, although one-click coloring for single voxel is supported, in the future more types of premier editing can be considered, e.g., a symmetry-considered editing, a fast creation of color pattern on the surface of voxel model.

For the generation of block layout in mini block construction, our system has optimized for the calculation of heuristic-based stability. However, to ensure the stability in a tangible block construction, it is better to use a physical stability. Though the state-of-the-art method using physical stability has only been tested for LEGO blocks, the possibility of extension to blocks of other brands can also be explored.

To make our proposed method for block assembly more compatible with various kinds of block models, we can evolve our component generation to satisfy more requirements, e.g., new definition of weakly-connected blocks in high-res block constructions, components with more perceivable shapes, the static equilibrium of component during assembly. To move a step further, we can also explore the possibility of extending our block assembly method for the self-assembly robots. To make the automatically generated building instructions more user-friendly, as summarized in Section 4.1.1, diverse notations shown in manually drawn instructions might be considered.

Acknowledgements

This research project would not have been possible without the support of many people. Initially, I would like to thank my advisor, Prof. Jun Mitani, who constantly reviewed my numerous oral reports and paper work as well as ingeniously feeding back on them. He not only helped make sense of my confusion in academic area, but also taught me how to improve my skills of technical writing and presentation. Moreover, I would like to thank Prof. Yuki Igarashi, who insisted on resourcefully advising me through email these years even on her pregnancy. Also thank Prof. Yoshihiro Kanamori for his cautiousness, which helped discipline me more strictly in my life. Of course, my gratitude goes to all the reviewers for advising my papers, and all NPAL members including graduated students as well for their companionship and support, especially for Shuang Tang, who was my tutor and helped me a lot in my daily life.

I am grateful for the Government of Japan and China. They offered me the chance of living and studying in Japan during these three years supported by the MEXT scholarship. Thank University of Tsukuba for providing me such a fully equipped international environment, which helped enrich my personal life in campus, and helped me make a lot of international friends. Finally, I will always remember that, I could not enjoy my life so much if without the love and support from my family and friends. The appreciation to you is beyond words.

Sep. 2016
Man Zhang

Bibliography

[1] Maneesh Agrawala, Wilmot Li, and Floraine Berthouzoz. Design principles for visual communication. *Commun. ACM*, 54(4):60–69, April 2011.

[2] Maneesh Agrawala, Doantam Phan, Julie Heiser, John Haymaker, Jeff Klingner, Pat Hanrahan, and Barbara Tversky. Designing effective step-by-step assembly instructions. In *ACM SIGGRAPH 2003 Papers*, SIGGRAPH '03, pages 828–837, New York, NY, USA, 2003. ACM.

[3] BlockCAD. Blockcad. `http://www.blockcad.net`. Accessed: 2016-07-14.

[4] BrickRecycler. Brickrecycler.com. `http://www.brickrecycler.com/lego-fun-facts`. Accessed: 2016-05-28.

[5] Burton. Aerial burton. `http://burton-jp.com/en/index.htm`. Accessed: 2016-06-15.

[6] Lujie Chen and Lawrence Sass. Generative computer-aided design: New design tool for direct physical production. In *Proc. of CAD'16*, pages 1–4, 2016.

[7] Travis Cobbs. Ldview. `http://ldview.sourceforge.net`. Accessed: 2016-07-14.

[8] Daniel Cohen-Or and Arie Kaufman. Fundamentals of surface voxelization. *Graph. Models Image Process.*, 57(6):453–461, November 1995.

[9] Kaufman Arie Cohen-Or, David. *Scan-conversion Algorithms for Linear and Quadratic Objects*. IEEE Computer Society Press, 1991.

[10] Solid Works Corp. Solid works. `http://www.solidworks.com`. Accessed: 2016-07-14.

[11] Jesus Diaz. Gizmodo. `http://lego.gizmodo.com/this-incredible-full-scale-lego-x-wing-is-the-largest-m-509484787`, 2013. Accessed: 2016-05-28.

[12] Elmar Eisemann and Xavier Décoret. Single-pass gpu solid voxelization for real-time applications. In *Proceedings of Graphics Interface 2008*, GI '08, pages 73–80, Toronto, Ont., Canada, Canada, 2008. Canadian Information Processing Society.

[13] Shiaofen Fang and Hongsheng Chen. Hardware accelerated voxelization. *Computers & Graphics*, 24(3):433 – 442, 2000.

[14] Steven K. Feiner and Dorée Duncan Seligmann. Cutaways and ghosting: Satisfying visibility constraints in dynamic 3d illustrations. *The Visual Computer*, 8(5):292–302, 1992.

[15] Markus Funk, Andreas Bächler, Liane Bächler, Oliver Korn, Christoph Krieger, Thomas Heidenreich, and Albrecht Schmidt. Comparing projected in-situ feedback at the manual assembly workplace with impaired workers. In *Proceedings of the 8th ACM International Conference on PErvasive Technologies Related to Assistive Environments*, PETRA '15, pages 1:1–1:8, New York, NY, USA, 2015. ACM.

[16] Remi Gagne. Lego instruction creator. `http://bugeyedmonkeys.com/lic_info`, 2010. Accessed: 2016-05-28.

[17] Michael Garland and Paul S. Heckbert. Surface simplification using quadric error metrics. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '97, pages 209–216, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[18] Timothy Gerstner, Doug DeCarlo, Marc Alexa, Adam Finkelstein, Yotam Gingold, and Andrew Nealen. Pixelated image abstraction. In *Proceedings of the Symposium on Non-Photorealistic Animation and Rendering*, NPAR '12, pages 29–36, Aire-la-Ville, Switzerland, Switzerland, 2012. Eurographics Association.

[19] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. Deep learning. Book in preparation for MIT Press, 2016.

[20] Google and LEGO. Google & lego: Build with chrome. `https://www.buildwithchrome.com`. Accessed: 2016-07-14.

[21] Rebecca Gower, Agnes Heydtmann, and Henrik Petersen. *LEGO: Automated Model Construction.* Jens Gravesen and Poul Hjorth, 1998.

[22] Jin-Kang Gui and Martti Mäntylä. Functional understanding of assembly modelling. *Computer-Aided Design*, 26(6):435 – 451, 1994.

[23] Ankit Gupta, Dieter Fox, Brian Curless, and Michael Cohen. Duplotrack: A real-time system for authoring and guiding duplo block assembly. In *Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology*, UIST '12, pages 389–402, New York, NY, USA, 2012. ACM.

[24] Julie Heiser, Doantam Phan, Maneesh Agrawala, Barbara Tversky, and Pat Hanrahan. Identification and validation of cognitive design principles for automated generation of assembly instructions. In *Proceedings of the Working Conference on Advanced Visual Interfaces*, AVI '04, pages 311–319, New York, NY, USA, 2004. ACM.

[25] Jhen-Yao Hong, Der-Lor Way, Zen-Chung Shih, Wen-Kai Tai, and Chin-Chen Chang. Inner engraving for the creation of a balanced lego sculpture. *The Visual Computer*, 32(5):569–578, 2016.

[26] Hugues Hoppe. New quadric metric for simplifying meshes with appearance attributes. In *Proceedings of the 10th IEEE Visualization 1999 Conference (VIS '99)*, VISUALIZATION '99, pages –, Washington, DC, USA, 1999. IEEE Computer Society.

[27] Jian Huang, Roni Yagel, Vassily Filippov, and Yair Kurzion. An accurate method for voxelizing polygon meshes. In *Proceedings of the 1998 IEEE Symposium on Volume Visualization*, VVS '98, pages 119–126, New York, NY, USA, 1998. ACM.

[28] Yuki Igarashi, Takeo Igarashi, and Jun Mitani. Beady: Interactive beadwork design and construction. *ACM Trans. Graph.*, 31(4):49:1–49:9, July 2012.

[29] Arie Kaufman. Efficient algorithms for 3d scan-conversion of parametric curves, surfaces, and volumes. *SIGGRAPH Comput. Graph.*, 21(4):171–179, August 1987.

[30] Arie Kaufman, Daniel Cohen, and Roni Yagel. Volume graphics. *Computer*, 26(7):51–64, July 1993.

[31] Johannes Kopf and Dani Lischinski. Depixelizing pixel art. *ACM Trans. Graph.*, 30(4):99:1–99:8, July 2011.

[32] Johannes Kopf, Ariel Shamir, and Pieter Peers. Content-adaptive image downscaling. *ACM Trans. Graph.*, 32(6):173:1–173:8, November 2013.

[33] Ming-Hsun Kuo, You-En Lin, Hung-Kuo Chu, Ruen-Rone Lee, and Yong-Liang Yang. Pixel2brick: Constructing brick sculptures from pixel art. *Comput. Graph. Forum*, 34(7):339–348, October 2015.

[34] Michael Lachmann. Mike's lego cad. `http://mlcad.lm-software.com`. Accessed: 2016-07-14.

[35] Manfred Lau, Akira Ohgawara, Jun Mitani, and Takeo Igarashi. Converting 3d furniture models to fabricatable parts and connectors. *ACM Trans. Graph.*, 30(4):85:1–85:6, July 2011.

[36] LEGO. Lego digital designer. `http://ldd.lego.com`. Accessed: 2015.

[37] LEGO. Lego mindstorms. `http://mindstorms.lego.com`. Accessed: 2016-05-28.

[38] LEGO. Lego minifigures. `http://www.lego.com/en-us/minifigures`. Accessed: 2016-05-28.

[39] LEGO. Lego building instructions. `https://wwwsecure.us.lego.com/en-us/service/buildinginstructions`, 2014. Accessed: 2016-05-28.

[40] LEGO. Lego company profile. `http://www.lego.com/en-us/aboutus/lego-group/company-profile`, 2014. Accessed: 2016-05-28.

[41] LeoCAD. Leocad. `http://www.leocad.org`. Accessed: 2016-07-14.

[42] Yuen-Shan Leung and Charlie C. L. Wang. Conservative sampling of solids in image space. *IEEE Computer Graphics and Applications*, 33(1):32–43, Jan 2013.

[43] Wilmot Li, Maneesh Agrawala, Brian Curless, and David Salesin. Automated generation of interactive 3d exploded view diagrams. In *ACM SIGGRAPH 2008 Papers*, SIGGRAPH '08, pages 101:1–101:7, New York, NY, USA, 2008. ACM.

[44] Wilmot Li, Lincoln Ritter, Maneesh Agrawala, Brian Curless, and David Salesin. Interactive cutaway illustrations of complex 3d models. In *ACM SIGGRAPH 2007 Papers*, SIGGRAPH '07, New York, NY, USA, 2007. ACM.

[45] Trimble Navigation Limited. Sketchup. http://www.sketchup.com. Accessed: 2016-07-14.

[46] Juncong Lin, Xiaogang Jin, Zhengwen Fan, and Charlie C. L. Wang. Automatic polycube-maps. In Falai Chen and Bert Jüttler, editors, *Advances in Geometric Modeling and Processing: 5th International Conference, GMP 2008, Hangzhou, China, April 23-25, 2008. Proceedings*, pages 3–16. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[47] Kui-Yip Lo, Chi-Wing Fu, and Hongwei Li. 3d polyomino puzzle. *ACM Trans. Graph.*, 28(5):157:1–157:8, December 2009.

[48] Kawada Co. Ltd. Nanoblock. http://www.diablock.co.jp/nanoblock/catalog/minicollection. Accessed: 2016-05-28.

[49] Linjie Luo, Ilya Baran, Szymon Rusinkiewicz, and Wojciech Matusik. Chopper: Partitioning models into 3d-printable parts. *ACM Trans. Graph.*, 31(6):129:1–129:9, November 2012.

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

[51] Ravish Mehra, Qingnan Zhou, Jeremy Long, Alla Sheffer, Amy Gooch, and Niloy J. Mitra. Abstraction of man-made shapes. *ACM Trans. Graph.*, 28(5):137:1–137:10, December 2009.

[52] Daniel Mendes, Pedro Lopes, and Alfredo Ferreira. Hands-on interactive tabletop lego application. In *Proceedings of the 8th International Conference on Advances in Computer Entertainment Technology*, ACE '11, pages 19:1–19:8, New York, NY, USA, 2011. ACM.

[53] Patrick Min. Binvox. http://www.google.com/search?q=binvox. Accessed: 2016-05-28.

[54] Patrick Min. Thinvox. http://www.google.com/search?q=thinvox. Accessed: 2016-05-28.

[55] Jun Mitani and Hiromasa Suzuki. Making papercraft toys from meshes using strip-based approximate unfolding. *ACM Trans. Graph.*, 23(3):259–263, August 2004.

[56] Yuki Mori and Takeo Igarashi. Plushie: An interactive design system for plush toys. *ACM Trans. Graph.*, 26(3), July 2007.

[57] Stefanie Mueller, Tobias Mohr, Kerstin Guenther, Johannes Frohnhofen, and Patrick 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*, CHI EA '14, pages 187–188, New York, NY, USA, 2014. ACM.

[58] Lee R. Nackman and Stephen M. Pizer. Three-dimensional shape description using the symmetric axis transform i: Theory. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-7(2):187–202, March 1985.

[59] Fakir S. Nooruddin and Greg Turk. Simplification and repair of polygonal models using volumetric techniques. *IEEE Transactions on Visualization and Computer Graphics*, 9(2):191–205, April 2003.

[60] Yoichi Ochiai, Takayuki Hoshi, and Jun Rekimoto. Pixie dust: Graphics generated by levitated and animated objects in computational acoustic-potential field. *ACM Trans. Graph.*, 33(4):85:1–85:13, July 2014.

[61] Yoichi Ochiai, Kota Kumagai, Takayuki Hoshi, Jun Rekimoto, Satoshi Hasegawa, and Yoshio Hayasaki. Fairy lights in femtoseconds: Aerial and volumetric graphics rendered by focused femtosecond laser combined with computational holographic fields. *ACM Trans. Graph.*, 35(2):17:1–17:14, February 2016.

[62] Estate of James Jessiman. Ldraw. `http://www.ldraw.org`. Accessed: 2016-07-14.

[63] Sumiaki Ono, Alexis André, Youngha Chang, and Masayuki Nakajima. Lego builder: Automatic generation of lego assembly manual from 3d polygon model. *ITE Transactions on Media Technology and Applications*, 1(4):354–360, 2013.

[64] Kálmán Palágyi and Attila Kuba. Directional 3d thinning using 8 subiterations. In *Proceedings of the 8th International Conference on Discrete Geometry for Computer Imagery*, DCGI '99, pages 325–336, London, UK, UK, 1999. Springer-Verlag.

[65] Pavel Petrovic. Solving lego brick layout problem using evolutionary algorithms. In *Proc. of Norsk Informatik Konferanse*, pages 87–97, 2001.

[66] Soren Rolighed, Miguel Agullo, Hideaki Yabuki, and Kevin Clague. Lpub. `http://lpub.sourceforge.net`. Accessed: 2016-07-14.

[67] Tiago Santos, Alfredo Ferreira, Filipe Dias, and Manuel J. Fonseca. Using sketches and retrieval to create lego models. In *Proceedings of the Fifth Eurographics Conference on Sketch-Based Interfaces and Modeling*, SBM'08, pages 89–96, Aire-la-Ville, Switzerland, Switzerland, 2008. Eurographics Association.

[68] Ariel Shamir. A survey on mesh segmentation techniques. *Computer Graphics Forum*, 27(6):1539–1556, 2008.

[69] Luís F. M. S. Silva, Vitor F. Pamplona, and Jo ao L. D. Comba. Legolizer: A real-time system for modeling and rendering lego representations of boundary models. In *2009 XXII Brazilian Symposium on Computer Graphics and Image Processing*, pages 17–23, Oct 2009.

[70] Allen Smith. Bricksmith. `http://bricksmith.sourceforge.net`. Accessed: 2016-07-14.

[71] Alexander Spröwitz, Rico Moeckel, Massimo Vespignani, Stephane Bonardi, and Auke J. Ijspeert. Roombots: A hardware perspective on 3d self-reconfiguration and locomotion with a homogeneous modular robot. *Robotics and Autonomous Systems*, 62(7):1016 – 1033, 2014. Reconfigurable Modular Robotics.

[72] Marco Tarini, Kai Hormann, Paolo Cignoni, and Claudio Montani. Polycube-maps. *ACM Trans. Graph.*, 23(3):853–860, August 2004.

[73] Romain Testuz, Yuliy Schwartzburg, and Mark Pauly. Automatic generation of constructable brick sculptures. In *Proc. of Eurographics 2013*, volume 13, pages 81–84, 2013.

[74] Reichling Tobias and Schütz Adrian. Pictobrick. `http://www.pictobrick.de/en/pictobrick.shtml`, 2010. Accessed: 2016-07-14.

[75] Barbara Tversky and Kathleen Hemenway. Objects, parts and categories. *Journal of Experimental Psychology: General*, 113(2):169–193, June 1984.

[76] Lynette Van Zijl and Eugene Smal. Cellular automata with cell clustering. In *Proc. of Automata '08*, pages 425–441, 2008.

[77] Ivan Viola and Meister E. Gröller. Smart visibility in visualization. In *Proceedings of the First Eurographics Conference on Computational Aesthetics in Graphics, Visualization and Imaging*, Computational Aesthetics'05, pages 209–216, Aire-la-Ville, Switzerland, Switzerland, 2005. Eurographics Association.

[78] Sidney W. Wang and Arie E. Kaufman. Volume sampled voxelization of geometric primitives. In *Proceedings of the 4th Conference on Visualization '93*, VIS '93, pages 78–84, Washington, DC, USA, 1993. IEEE Computer Society.

[79] Sabine Webel, Ulrich Bockholt, and Jens Keil. Design criteria for ar-based training of maintenance and assembly tasks. In Randall Shumaker, editor, *Virtual and Mixed Reality - New Trends: International Conference, Virtual and Mixed Reality 2011, Held as Part of HCI International 2011, Orlando, FL, USA, July 9-14, 2011, Proceedings, Part I*, pages 123–132. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[80] David V. Winkler. Automated brick layout. `http://tinyurl.com/plutj8`, 2005. BrickFest 2005.

[81] Zhirong Wu, Shuran Song, Aditya Khosla, Fisher Yu, Linguang Zhang, Xiaoou Tang, and Jianxiong Xiao. 3d shapenets: A deep representation for volumetric shapes. *Proc. of CVPR'15*, 2015.

[82] Shiqing Xin, Chi-Fu Lai, Chi-Wing Fu, Tien-Tsin Wong, Ying He, and Daniel Cohen-Or. Making burr puzzles from 3d models. *ACM Trans. Graph.*, 30(4):97:1–97:8, July 2011.

[83] Bailin Yang, Frederick W. B. Li, Xun Wang, Mingliang Xu, Xiaohui Liang, Zhaoyi Jiang, and Yanhui Jiang. Visual saliency guided textured model simplification. *The Visual Computer*, pages 1–18, 2015.

[84] Man Zhang, Yuki Igarashi, Yoshihiro Kanamori, and Jun Mitani. Designing mini block artwork from colored mesh. In *Proc. of Smart Graphics 2015*, 2015.

[85] Man Zhang, Yuki Igarashi, Yoshihiro Kanamori, and Jun Mitani. Component-based building instructions for block assembly. In *Proc. of CAD'16*, pages 55–59, 2016.

[86] Man Zhang, Yuki Igarashi, Yoshihiro Kanamori, and Jun Mitani. Component-based building instructions for block assembly. *Computer-Aided Design & Applications*, 14(a), 2017. (To appear).

[87] Man Zhang, Jun Mitani, Yoshihiro Kanamori, and Yukio Fukui. Blocklizer: Interactive design of stable mini block artwork. In *ACM SIGGRAPH 2014 Posters*, SIGGRAPH '14, pages 18:1–18:1, New York, NY, USA, 2014. ACM.

[88] Yahan Zhou, Shinjiro Sueda, Wojciech Matusik, and Ariel Shamir. Boxelization: Folding 3d objects into boxes. *ACM Trans. Graph.*, 33(4):71:1–71:8, July 2014.