國立臺灣大學電機資訊學院資訊工程學系

碩士論文

Department of Computer Science and Information Engineering

College of Electrical Engineering and Computer Science

National Taiwan University

Master Thesis

將三維立體模型轉換成可建造之樂高積木模型

Transform 3D Models Into Constructible LEGO Brick Sculpture

廖航緯

Liao Han-Wei

指導教授：陳炳宇 博士

Advisor: Chen Bing-Yu, Ph.D.

中華民國 101 年 8 月

August, 2012

致謝

這份研究，其實一開始要先感謝博班的鄭鎧尹學長，打槍我在碩二暑假展開的立體視覺與手勢人機互動研究。而當我自己也發現 這題目真的做不下去，而手邊有興趣的題目也又難又少的時候，感謝在十一月初的時候，博班的羅聖傑阿山學長 和博班的吳昱霆Kevin學長，與我在506實驗室門口的腦力激盪。藉由變形金剛組合動畫與模型機器人化的拋磚引玉之下，我們成功 地想到了這個題目，也感謝我的指導教授Robin，可以讓我展開這個研究。在想好題目之後，有點跌跌撞撞地邊修課邊研究， 因此好像沒有實質進展，隨著許多相關文獻一一在寒假時期揭露與閱讀之後，之前認為這個題目會因為樂高積木太有名而沒希望的 想法也漸漸消失，開始轉而想真正嘗試先實作一篇相關文獻來測試想法，再次感謝我的小指導教授阿山，他幫了我在這個時期的文獻收集，與實作工具的選擇指引。在碩二下這半年，有一個時期緩慢地在完整實作前人的方法，一直到大約五月中後才明瞭 前人方法的效果，感謝耐心等待我完成的阿山學長與指導教授。在口試前一個半月，努力掙扎地想辦法解決連接性不穩定的狀況， 時間花費到甚至第一版論文是在口試前三天完成，而口試投影片更在短短兩天內完成，並且與指導教授經過一次預演與修改。在 這最後衝刺階段，要特別感謝跟我同一時間口試的書漾、玉芯、還有亮岑，沒有她們在旁處理口試委員的餐點與聯絡等等事項， 大概我就沒辦法成功口試了吧。還有要感謝蔡佩恆同學緊急在我口試前一天可以讓口試委員有實際組裝的人形樂高可以看，實際 組裝複雜模型過就知道，真的也是挺累人的。另外，在這容易低潮的衝刺時期，感謝CMLAB每個與我吃過飯的人，感謝小畢典時送我禮物的虹諭學妹、筱青學妹和兆懷學弟、幫我拍照的健庭學弟，感謝幫我的樂高兔子種紅蘿蔔的宇婷，感謝 心怡學姊的免費宵夜與早餐，感謝在臉書上祝我生日快樂的人們，感謝臉書上吃喝玩樂社團結交的朋友們，感謝讓我喇賽的DSP組， 感謝路過打屁的大學同學們與動物園，感謝兔將公司的實習薪水，更要感謝，我的父母對一直以來的一路支持。

真誠地想說，感謝，曾在我生命中，出現的每一個人。絕對不是因為要感謝的人太多，才說那就感謝天喔。（天音：真沒誠意）

中文摘要

本研究呈現了如何運用電腦圖學中，各種豐富的3D模型，轉換成形狀相近且趣味豐富的樂高積木模型。 轉換出來的樂高積木模型，不僅可以運用在虛擬的電腦圖學動畫當中，更可以藉由用人手進行 實際堆積組裝，產生出令人驚嘆的創作成果。 我們的方法主要分成兩個部分。首先，我們將三角形組成的立體模型描述檔，轉換成外型相似，由 方格組成的立體模型。由於參考的方格近似演算法並沒有達成我們期待的狀態，所以我們自行提出 了一種混和以前方格近似演算法的方法來取得更好的方格立體模型。接著，我們讓轉換好的立體方格模型， 經過簡單的預處理以後，進行連接擺放最佳化的計算。我們參考以前的作法，將此問題轉換成 細胞演化自動機的形式進行模擬與最佳化。但同樣的，此方法仍然沒有達成我們期待的完美結果，因此 我們提出後續補強的方法，既能提昇完全連接的機率，又能改善方塊使用數量的控制。最後，我們附加 地提出了一個簡易的演算法來產生方便人們組裝的建造順序。這或許可以提供一個研究出發點，來改善 如何視覺化樂高堆積順序，使人們能清楚了解與實際操作。

關鍵字： 樂高益智積木， 立體模型方格化， 連接擺放最佳化，幾何組裝推理

Abstract

In this work, we present a system, “legolizer”, to transform 3D models into buildable LEGO sculptures. Our legolized 3D model has various purposes. It can not only be used in the production of LEGO-style computer animations, but also can be actually built by hand to amuse people. Our system consist of two parts. The first part converts the input triangular 3D mesh model into grid-like voxel representation. Because the utilized voxelization methods did not reach our expectation, we purposed a hybrid voxelization approach in attempt to obtaining a better voxelized model. Then, after simple preprocessing, we put the voxelized model into the second part of our system - connectivity placement optimization. We reduced this placement problem into the form of cellular automaton similar to the most recent previous work. Again, due to imperfection of cellular automaton optimization, we later proposed our refinement approaches to improve not only the probability of fully connecting, but also the brick usage control. Finally and additionally, we compute an easy-to-build instruction sequence of our obtained LEGO sculpture. This may provide a researching starting point of improving the visualization of computed LEGO brick assembly process for people easily understanding.

Keyword: LEGO brick puzzle, 3D model voxelization, connectivity placement optimization, geometric assembly reasoning

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Motivation

The LEGO company, founded in Denmark since 1932, continuously produce colorful interlocking plastic bricks to fill many people's childhood memory. During the assembly process, we overcame the exponential growth of construction possibilities step-by-step, and eventually finished the shape we want. Like many other puzzle games, building LEGO model is a fascinating, intriguing and entertaining activities for adults and kids [1]. It even evolved into an artform, for instances, Dispatchwork project [2], Nathan Sawaya‘s creations [3], and LEGOLAND's around the world [3, 4]. These complex LEGO sculptures really make people wonder how to build them. This mysterious puzzle later became known as ”LEGO construction problem.” In other words, ”Given any 3D body, how can it be built from LEGO bricks?” (see Figure 1.1 for illustration) [5, 6]

Over the last few years, the advancement of computer graphics and vision has made making and retrieving 3D models easy. Especially, with the advent of depth camera based 3D geometry scanning technology [7], it is expected that many real-world objects will be scanned and put into online 3D model repositories. By using the state-of-the-art 3D model retrieval techniques [8, 9], getting any 3D model from these repositories is not difficult anymore. However, in order to fit these models into scenes of different styles, artists often need to remake these models. This has trigger some of computer graphics researchers to study the topic - ”3D Model Style Transfer”. Transforming a 3D model into constructible

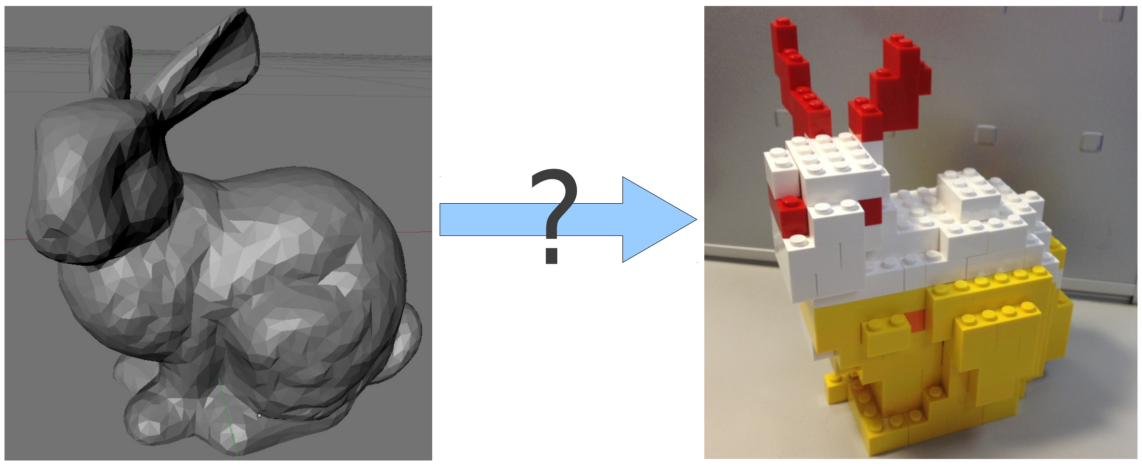


Figure 1.1: Problem illustration. The left image: input 3D shape represented by triangle meshes. The right image: buildable LEGO sculpture instructions.

LEGO sculpture is such a topic related. In addition, if a 3D model can be converted into LEGO buildable model, this may bring many other computer graphics applications such as LEGO sculpture force stability simulation, automatic generation of LEGO assembly process animation, and more. These are really interesting and challenging topics.

1.2 Background - The LEGO construction problem

The ”LEGO construction problem” has already been presented to the scientific community at large in 1998 [6] and 2001 [10]. In essence, what the LEGO company wished for, was a computer program that can generate LEGO building instructions for any real-world object within a reasonable amount of time [5]. After a rough analysis, this problem has relationships to a number of fields in computer sciences and mathematics such as computer graphics, computer vision, and combinatorial optimization. Although representing and visualizing 3D shape model can be handled by the techniques in computer graphics and vision, LEGO construction problem is still difficult for computer to solve. Because for a given 3D shape, there are multiple choices of LEGO bricks and multiple ways to assemble them. This combinatorial explosion nature quickly make this problem intractable.

This problem, like most area filling problems, has the properties of an optimization problem. In general, it has to achieve the following objectives [5, 6, 11] :

- Connectivity: The constructed LEGO sculpture should be one single connected object. (i.e., the sculpture should not contain easy-pick LEGO bricks and floating LEGO bricks. See Figure 1.2 for illustration.)
- Shape Similarity: The constructed LEGO sculpture should look similar to the original input 3D shape.
- Cost Optimiality: To save money and time when building with actual LEGO bricks, the program should try to minimize the number of LEGO bricks used to build the sculpture. If possible, try to match the quantity limit of every brick.
- Runtime Limit: As previously mentioned, this problem should be solved in reasonable time.

The first and second optimization objectives directly affect the composability of output LEGO sculpture if there is no quantity constraint on every type of LEGO brick. The third objectives can effectively reduce the actual construction time, but in fact, we are often unable to finish the sculpture because of not enough specific kinds of LEGO brick, rather than the construction itself takes time. Therefore, we change the third objective into the following objective:

- Quantity Constraint: The solution should not violate the quantity limit of each LEGO brick.

To solve this problem, the program - ”legolizer” (see Figure 1.3), expect two inputs: the family of usable LEGO brick set and the 3D geometry model. And often with a quantity constraint for each kind of LEGO brick. The program should output the building instructions of the sculpture if this model is realizable. Otherwise, it should report the reason of the failure synthesis. The reason can be not enough specific kinds of LEGO brick, or unable to find the placement that can fully connect these bricks.

The LEGO construction problem can be seen as a three dimensional volume filling problem. The search space is extremely large and there can even be more than one viable solution. In fact, the two dimensional area filling problem is already considered to be NP-complete [5]. Furthermore, to ensure the stability and visual similarity of the sculpture,

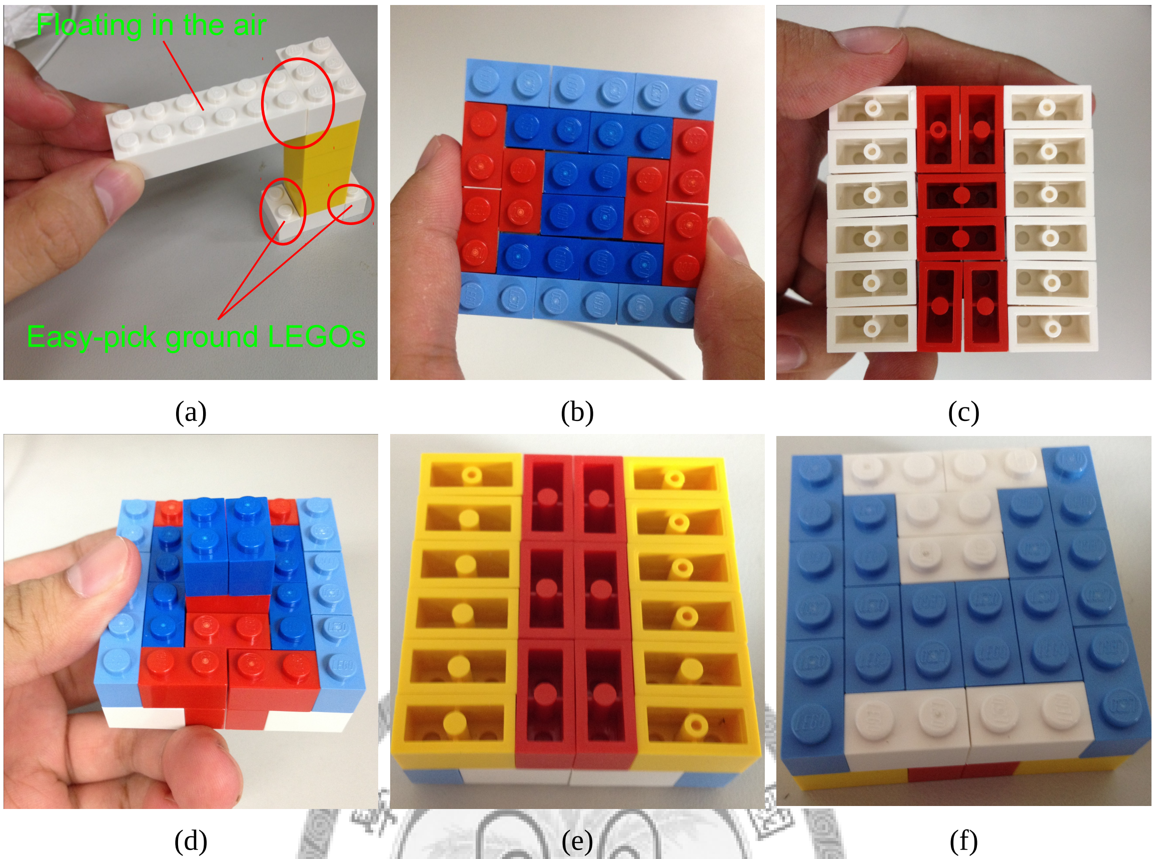


Figure 1.2: Connectivity objective. (a) shows what are disconnected LEGO bricks. Red circles indicate there are vertical cutting edge spliting white LEGOs from yellow pillar. Easy-pick LEGO bricks are also known as floating bricks on the ground. These bricks will cause the entire sculpture unable to leave the ground. (b), (c) and (d) demonstrate the case that it looks like fully connected but in fact it is disconnected. This also shows the difficulty of the LEGO construction problem. (e) and (f) is the correct solution for 6x6x2 plate using only 1x2 LEGOs.

it needs to consider shape similarity and connectivity objectives when filling LEGO brick into the target volume. This increase the complexity and difficulty of the problem. However, these constraints can also be exploited and devise an efficient approximation method to deliver acceptable solutions. We will discuss previous approaches to the problem in the related work chapter.

1.3 Contributions

In this work, we present a system to convert a 3D triangle mesh model into a buildable LEGO sculpture. We extend previous work [5] by adding several new features that may help not only obtaining better automatic results, but also designing better optimization

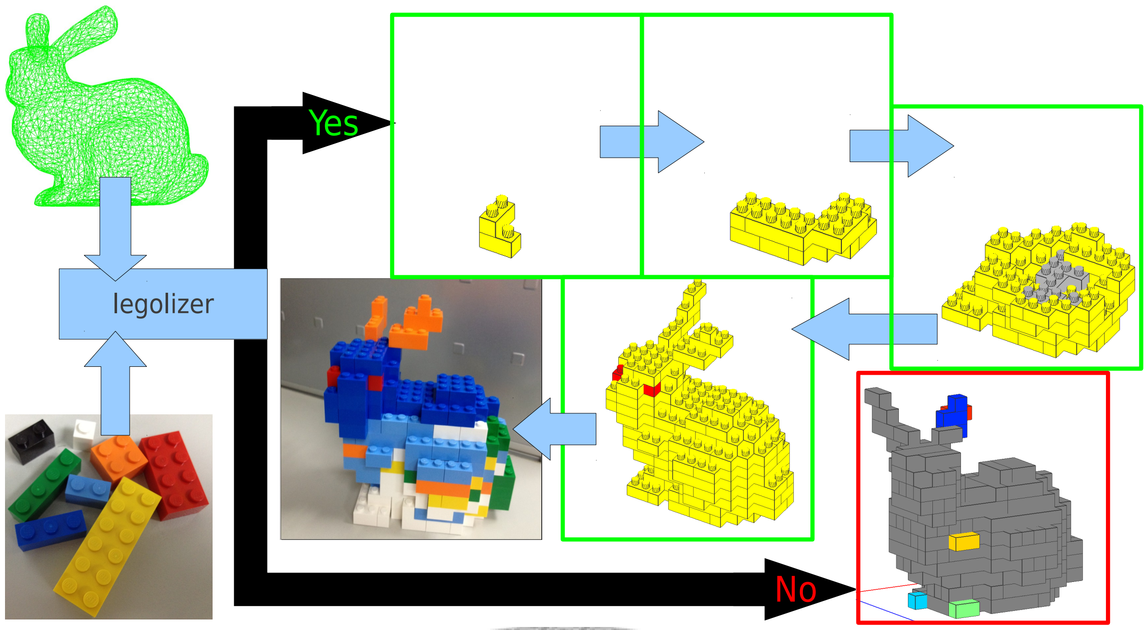


Figure 1.3: System flowchart.

algorithms. These features are:

- Combine two different techniques of voxelizing 3D triangular mesh model into a hybrid voxelization approach, described in the section 3.2.
- Post-optimization connectivity refinement based on connectivity risks information, described in the sectionin section 3.4.
- Post-optimization quantity constraint refinement, described in the section 3.4.
- Connectivity-aware and positional-aware building instruction order generation, described in the section 3.4.

Chapter 2

Related Work

2.1 LEGO construction automation and stylization

Since the presentation of LEGO construction automation problem [6, 10] , there were a number of previous works trying to solve this problem. Most recent work is presented by van Zijl and Smal [5] in 2008, which is our main reference work. Their work purposed a new optimization approach using cellular automaton and obtained comparable results to the ones using beam search. Cellular automaton is a computation model originated from John von Neumann [12]. This concept is applied in various scientific fields nowadays. They formulated this problem by viewing LEGO bricks as cells that lives in a grid world which represents a layer of voxelized model. By defining certain rules of cellular interaction, they can evolved into larger LEGO bricks, and eventually produce a complete layout of a LEGO structure. Beam search is an another optimization-by-search algorithm, firstly presented by David V. Winkler, in a LEGO fans convention called "BrickFest" in 2005. Conceptually, it is a breadth-first search with tree width limit, hence visually like shooting a "beam" during the expansion of search tree. The tree width K means the algorithm will try to store K currently best solutions. To solve LEGO construction layout problem, it searches step-by-step LEGO brick placement and evaluate the cost of placement. Finally, they implemented a completed Java package consist of 3D model voxelization and the above two optimization techniques for producing an constructible LEGO structure. The application can be found at `http://lsculpturer.sourceforge.net`

More earlier attempts are works by Gower et al. [11], Petrovic [10], and Na [13]. Gower et al first formulated the LEGO construction as a combinatorial optimization problem. They proposed various observation in designing the connectivity cost function, and described some methods such as local search and simulated annealing to obtain connected LEGO structure. Petrovic applied energy terms by Gower et al. into his three genetic algorithms, but accroding to [5], their attempts required significant more execution time which fails the expectation of running in reasonable time. Na later extended the indirect gene representation of Petrovic by allowing the builing patterns to extend in two directions instead of one. At last, although not directly solving the exact LEGO construction problem, Lambrecht presented LSculpt [14], which produces constructible LEGO sculpture using oriented LEGO plates instead of standard LEGO bricks. The output sculpture has improved detail than standard pixel-like look LEGO structure.

The above previous works were focus on solving the LEGO construction problem, they often utilized many combinatorial optimization techniques such as simulated annealing, genetic algorithm and so on. However, in computer graphics, it is also important to visualize realistic LEGO bricks. The work “legolizer”, done by Silva et al [15], is such one work. They successfully render a 3D triangular mesh model into similar shaped LEGO sculpture realistically in real-time. In essence, they present a LEGO-style volumtric rendering technique by implementing four GPU shader programs. These shader programs are triangle subdivison shader, voxelization shader, polygonization shader, and rendering shader. It is important to note that their work didn’t consider actual construction, that is, no brick connectivity consideration.

2.2 LEGO computer-aided design toolkits

The official LEGO Group has provided the LEGO Digital Designer, which recently incooprates the Australia Google Map service in parternship with Google. Users can use Google Chrome web browser to build their virtual LEGO structure on the Australia land. The LEGO Digital Designer can be found in [16] and [17] (the web version). Another popular LEGO CAD system is called “LDraw” [18], which is a community-maintained software project by James Jessiman since 1995. It is similar to the LEGO Digital Designer, but with many other documentation trying to standardize LEGO brick representation in a computer. Until now, various related LEGO CAD tools are created using LDraw as a reference. For example, L3P [19] is a conversion utility between LDraw file format and the scene file used in POV-Ray ray-tracing renderer [20]. For other related LEGO CAD tools, please refer to books [21, 22].

2.3 Recreational Computer Graphics and 3D Model Style Transformation

If not restricted to LEGO bricks, there are many efforts trying to make virtual 3D models realizable in many different kinds of form. This ia a related field called "Recreational Computer Graphics" [23]. Like the ultimate goal of LEGO construction problem, the objective of this field is to make a computer-aided geometric design tool that converts virtual 3D objects into a piecewise buildable object with step-by-step construction instructions. In recent literture, Lo et al [24] presented a new genre of 3D puzzle called "3D polyomino puzzle". A well-known example of polyomino puzzle tiling is the popular video game "Tetris" invented by the Russian Mathematician Pajitnov in 1985. In short, they designed a specialized interlocking tetromino bricks to assemble the virtual 3D model into real object. step-by-step building instructions. Xin et al [1] converts a 3D model into a 3D burr puzzle that consists of interlocking pieces with a single-key property. Shigeo et al [25] unfolds 3D triangular meshes onto a white paper so they can use glues and scissors to build the 3D papercraft models. Finally, Hildebrand et al [26] introduce an algorithm and representation for fabricating 3D shape abstractions using mutually intersecting planar cut-outs. The assembly process is like inserting pieces of paper into each other.

Because obtaining a realizable procedure from virtual 3D models directly implies that it can be represented by computers, we can view such realization process as a kind of 3D model style transformation. However, the results from 3D model style transformation need not to be realizable, this is another related field called "3D Model Style Transformation”. For instances, Xu et al. [27] extract the silhouette of object in a picture collection, then they morph a generic 3D model to fit the given silhouette. The generic 3D model and the object in the picture must have the same semantic meaning, if not, matching may fail. Shen et al. [28] converts a 3D character model into a style called “super-deformed”, abbreviated as SD. SD is a specific artistic style for Japanese manga and anime. The goal of SD style is making characters to be appeared cute and funny. They use energy optimization guided by a number of constraints that can not only capture the essence of SD style, but also stabilize deformation artifacts. Their system provides customizable parameter settings, and can convert character models into visually pleasing SD models in seconds.

Chapter 3

Methodology

3.1 System Overview

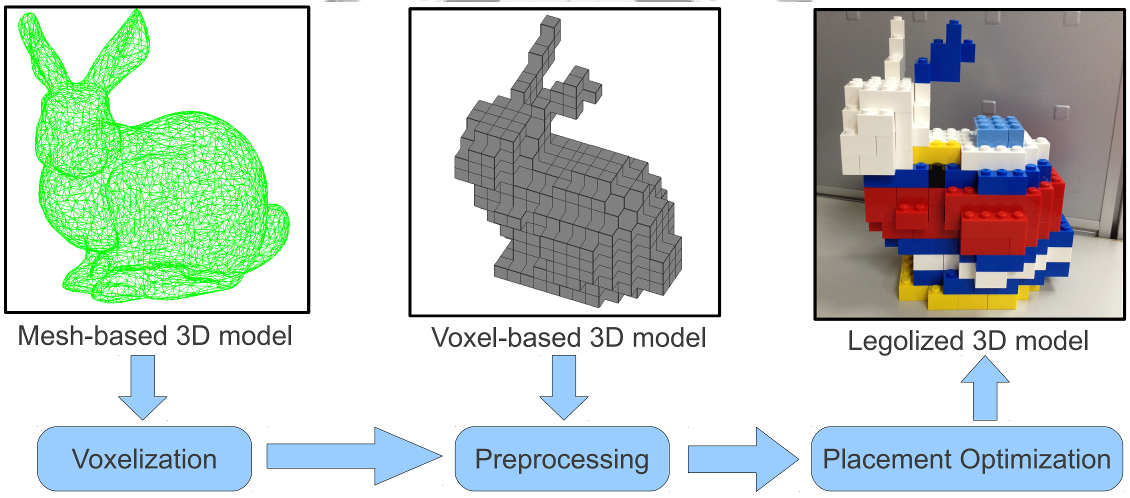


Figure 3.1: System overview.

Our system "legolizer" can be decomposed into three components, illustrated as Figure 3.1. Users can pick two kinds of input: mesh-based 3D models or voxel-based 3D models. When user provides 3D models consist of triangle meshes, we first voxelize it to get voxel-based 3D models. Then, we will preprocess these voxel-based 3D models such as checking connectivity and removing interior voxels. These steps will assist later steps to achieve our objectives. Finally, these preprocessed voxel models will put into placement optimizer to obtain LEGO building instructions. Users can then use our graphical user interface to actually build Legolized 3D models.

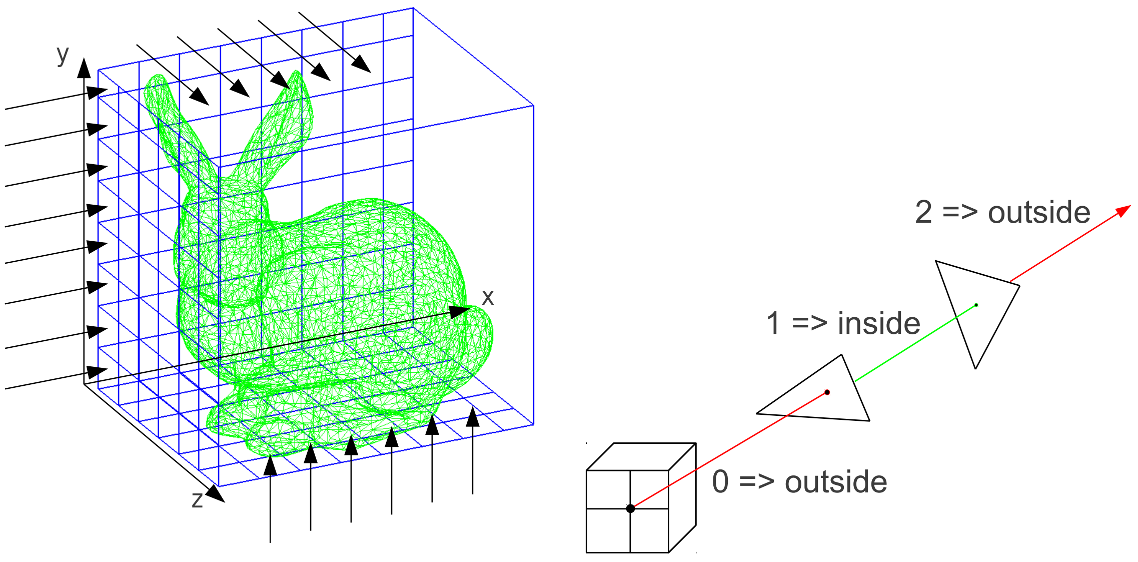
The implementation of our system is closely follow the paper [5]. We reference their system as our basis implementation, then add some new features to further refine connectivity and LEGO brick usage of the LEGO sculpture. We also try to generate brick-by-brick construction instructions rather than layer-by-layer.

3.2 Voxelization

Using standard cuboid LEGO brick to build a sculpture is an approximation by nature. The completed LEGO structure often have ”pixel-like”, ”voxel-like”, or ”grid-like” look. This observation gives a key direction for us to simplify our problem. That is, to apply voxelization algorithm on the triangle mesh 3D models.

The voxelization method can produce 3D voxel structures similar to the input mesh-based models, which solves the shape similarity goal of the LEGO construction problem. Besides, it reduces the search space of LEGO construction problem. Because the problem of finding where to place a LEGO brick which intersects the mesh model is delegated to the voxelizer, it effectively simplified the LEGO construction problem.

Among all popular voxelization methods, we utilize the voxelization algorithm described in the paper by Nooruddin et al. [29]. Essentially, it traces ray along all three axes (X, Y and Z) from corresponding perpendicular plane at zero. For example, if we trace ray along x-axis, then the ray origin is at (0, y, z), where Y and Z coordinate is the center of a pixel at yz-plane. During the ray-tracing, the algorithm records all the intersections with the triangles from the input triangular mesh model. The data structure used to record intersection is similar to a z-buffer. But instead of overwriting Z position when a closer intersection is found, the algorithm uses sequential container to record all the Z position of intersections at a particular xy-plane coordinate (See Figure 3.2 for illustration). After computing the z-buffer for three axes, the algorithm performs parity counting to determine the interior of a geometry body. That is, during intersection point counting, if the counting variable is odd, the algorithm marks the voxels until the counting variable become an even number. At last, the algorithm consider a voxel inside the body when at least two of three axes agreed (got at least 2 marks at that voxel position).



(a) (b)

Figure 3.2: Voxelization by ray-triangle intersection test. (a) Rays are shooted from z=0, x=0, y=0 planes. (b) The center of square-shaped pixel is where the ray shoots from, and hit many triangles. Parity counting identifies interior of the body.

The following pseudocode implements the above voxelization algorithm.

Algorithm 1 Ray-tracing voxelization

```
for all 3 principal axes: X,Y,Z do
    rename current ray-tracing axis to z axis        ▷ We pretend to trace ray along z axis
    for all triangle facets in the 3D mesh model do
        scan convert the triangle facet to find effective ray-tracing pixels on xy-plane.
        for all effective pixels on xy-plane do
            intersection $\leftarrow$ rayTriangleIntersect( ray, triangle )
            depthBuffer( pixel(x,y), axis ).insert( intersection )
                                        ▷ z-buffer can also be called as ”depth buffer”
        end for
    end for
    for all pixels on xy-plane do
        sort intersections in depthBuffer( pixel(x,y), axis ) from near to far.
    end for
end for
initialize 3D matrices $Count$, $Voxel$ to zero.
for all 3 principal axes: X,Y,Z do
    for $parityCount = 0$ to depthBuffer( pixel(x,y), axis ).size do
        intersection $\leftarrow$ depthBuffer( pixel(x,y), axis ).at( $parityCount$ ).
        $++Count$(rename(x,y,round(intersection.z))).
    end for
end for
mark $Voxel$(x,y,z) $= 1$ if corresponding $Count$(x,y,z) $\geq 2$
```

In the main reference paper [5], they used AABB-triangle intersection test (AABB: axis-aligned bounding box) and octree space subdivision to implement their own voxelizer. The intersection test they borrow from was developed by Akenine-Möller [30]. Basically, it was derived from the *separating axis theorem* (SAT). The theorem states that two convex polyhedra, A and B, are disjoint if one can find an axis along which the projection of the two polyhedra does not overlap. The axis is the one that

1. is parallel to a normal of a face of either A or B.

2. is formed from the cross product of an edge from A with and edge from B.

The separating axis theorem tests if two polyhedra collide by enumerating all axes that satisfies the conditions above. Then for each axis, vertices on these two polyhedra will project to the axis. Each polyhedron has their own range of projection. If these ranges do not intersect, then this axis is a separating axis. Instead of visually separating two objects, the axis is used as a normal vector to form a plane that separating these two objects. (Note: the ”bounding” B in AABB can be neglected if the box is not actually bounding an object. So we may use the term ”AAB” for generality). see Figure 3.3 for illustration.

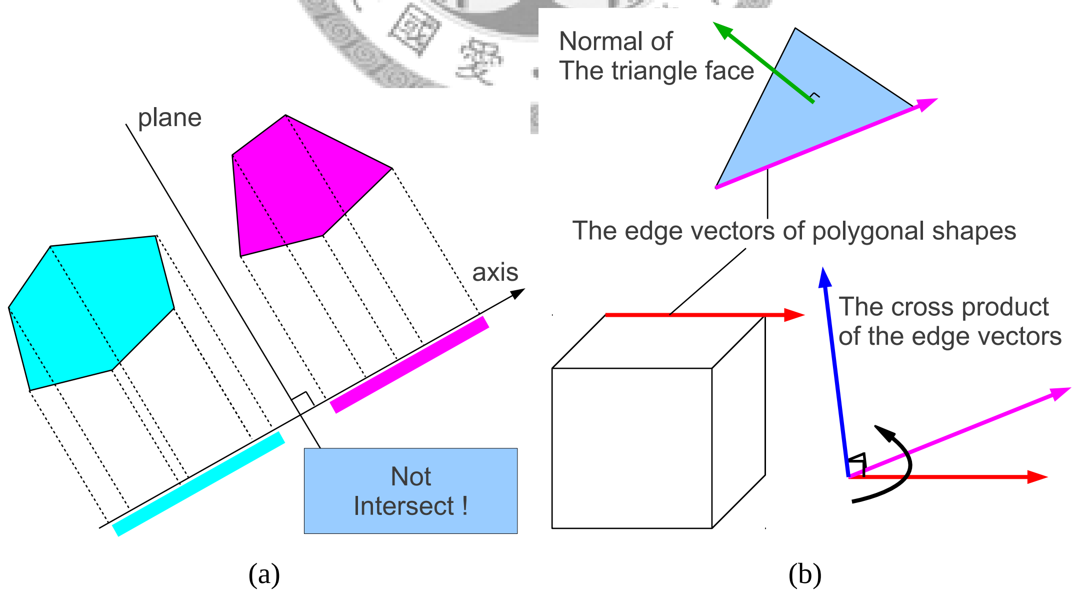


Figure 3.3: Separating axis theroem (SAT). (a) is a 2D version of SAT. The projected vertices ranges of the two polygon do not overlap, therefore not intersected. (b) shows that AABB-triangle intersection test is one of SAT simplest example. The green and blue vectors represents what kinds of axes are used in SAT intersection test.

In the implementation of Akenine-Möller's fast AAB-triangle intersection test, the axis-aligned box (AAB), defined by center point $\mathbf{c}$, and a vector of half lengths $\mathbf{h}$, is tested against the triangle $\Delta\mathbf{u}_0\mathbf{u}_1\mathbf{u}_2$ with a normal $\mathbf{n}$. The lengths in $\mathbf{h}$ can have different values along different axis, since axis-aligned nature firmly defines a box. Figure 3.4 shows the notation used for the AAB and the triangle.

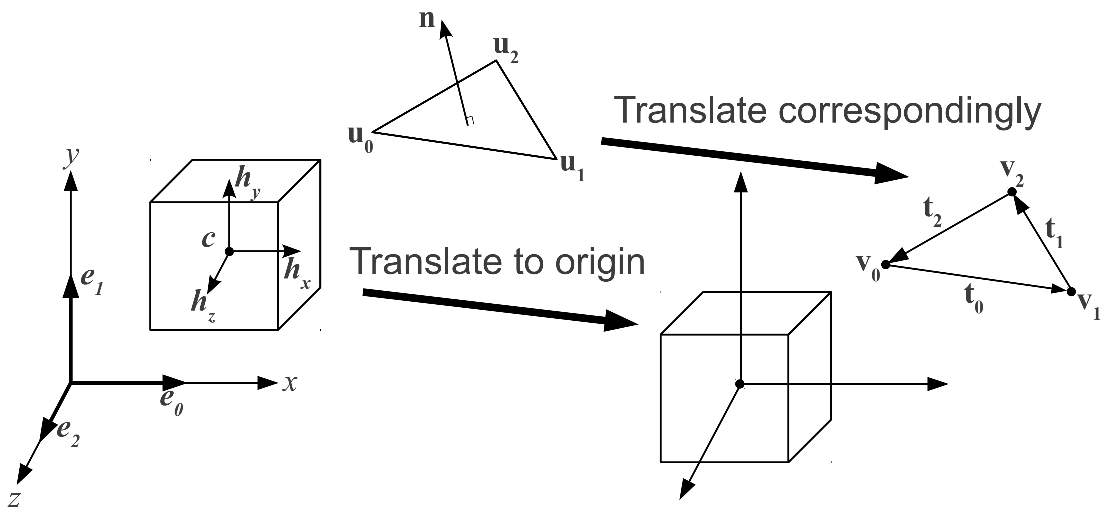


Figure 3.4: Notations for explanation of AAB-triangle intersection algorithm.

Initially, the triangle is translated so that the box is centered around the origin in order to simplify the tests. In mathematical notation, $\mathbf{v}_i = \mathbf{u}_i - \mathbf{c},\ \ i \in \{0,1,2\}$. Then based on SAT, the algorithm tests the following thirteen axes, which can be grouped in three categories and are performed according to the order of bullet numbering.

1. (9 tests) $\mathbf{a}_{ij} = cross\ product(\mathbf{e}_i\,,\mathbf{t}_j),\ \ i,j \in \{0,1,2\}$, where $\mathbf{t}_0 = \mathbf{v}_1 - \mathbf{v}_0$, $\mathbf{t}_1 = \mathbf{v}_2 - \mathbf{v}_1$, and $\mathbf{t}_2 = \mathbf{v}_0 - \mathbf{v}_2$. After computing the axis $\mathbf{a}_{ij}$, we need to project vertices of the triangle and the AAB onto the axis. The projection of the triangle vertices are performed by a regular dot product. That is, $p_k = \mathbf{a}_{ij} \cdot \mathbf{v}_k,\ \ k \in \{0,1,2\}$. Thanks to many zeros in the unit vector $\mathbf{e}_0\mathbf{e}_1\mathbf{e}_2$, the projection often results in one of p_k equal to another $p_q,\ \ k \neq q$. This observation leads to removing one conditional branch during the finding minimum and maximum of p_k, thus faster. Now it turns to the projection of AAB vertices. They compute a "radius", called r, of the box projected on $\mathbf{a}_{ij}$ (hereafter call $\mathbf{a}$) as $r = h_x|a_x| + h_y|a_y| + h_z|a_z|$. This is valid computation

reduction because the eight vertices of the translated AAB enumerate complete combination of signs in their own dot product projection. It can be proven that there must exists two vertices on the AAB, $\mathbf{x_{min}}$ and $\mathbf{x_{max}}$, such that $\mathbf{x_{max}} \cdot \mathbf{a} = r$ and $\mathbf{x_{min}} \cdot \mathbf{a} = -r$. Finally, the axis test is performed logically as:

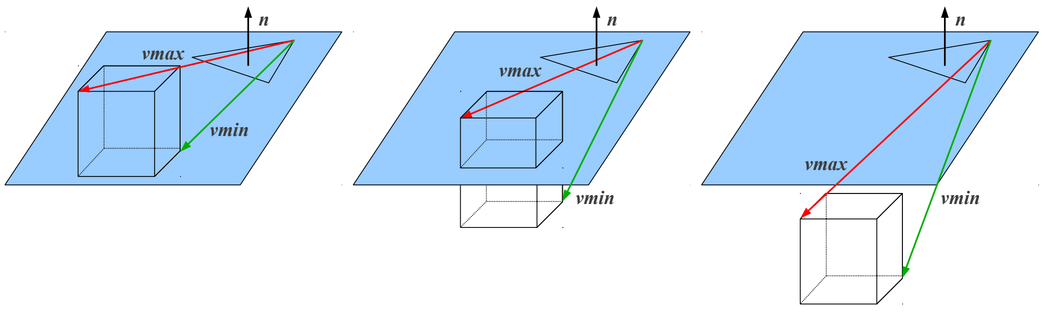
```
if  min( $p_0$ , $p_1$ , $p_2$ ) $> r$  or  max( $p_0$ , $p_1$ , $p_2$ ) $< -r$
    return "non-overlapping"
```

2. (3 tests) $\mathbf{e}_0 = (1, 0, 0)$, $\mathbf{e}_1 = (0, 1, 0)$, and $\mathbf{e}_2 = (0, 0, 1)$ (the normals of the axis-aligned (bounding) box). It is equivalent to test the AAB against the minimal AABB around the triangle. If we follow standard SAT procedures, the projection of vertices on unit vectors directly simplify the test a lot. It can be expressed as:

```
for all  axis $a \in X, Y, Z$  do
    if  min( $v_{0a}$ , $v_{1a}$ , $v_{2a}$ ) $> h_a$  or  max( $v_{0a}$ , $v_{1a}$ , $v_{2a}$ ) $< -h_a$
        return "non-overlapping"
end for
```

3. (1 test) $\mathbf{n} = cross\ product(\mathbf{t}_0 , \mathbf{t}_1)$ (the normal of the triangle). As previous SAT illustration, this test is equivalent to asking if the separating plane is the triangle itself. If this is the last test, then this test looks like asking if the triangle is cutting through the AAB. Because previous failed tests implied that all vertices of AAB can be projected onto the triangle (onto the plane defined by triangle normal and within the triangle area). Akenine-Möller simpify this test using fast plane-AAB intersection test, illustrated in Figure 3.5. (Therefore, readers should be noted that this test is not only for the last test. This test is a regular SAT test because a plane intersecting with an AAB does not imply the corresponding triangle also intersects the AAB.) The algorithm take a reference point $\mathbf{x}$ and $\mathbf{n}$ to form the plane, and the vector of half lengths $\mathbf{h}$ since the AAB is origin-translated. $\mathbf{x}$ can be any vertex of the triangle, they use $\mathbf{v}_0$ as $\mathbf{x}$.

At first, it finds the maximum and minimum distance vectors, $\mathbf{vmax}$ and $\mathbf{vmin}$, from all vertices on the AAB to the reference point $\mathbf{x}$. The following pseudocode



(a) ”not overlapping” (b) ”overlapping” (c) ”not overlapping”

Figure 3.5: SAT third test illustration.

takes advantage of translated AAB to achieve the same logic.

```
for all axis $a \in X, Y, Z$ do
    if $\mathbf{n}_a > 0$ then
        $\mathbf{vmin}_a \leftarrow -\mathbf{h}_a - \mathbf{x}_a$; $\mathbf{vmax}_a \leftarrow \mathbf{h}_a - \mathbf{x}_a$;
    else
        $\mathbf{vmin}_a \leftarrow \mathbf{h}_a - \mathbf{x}_a$; $\mathbf{vmax}_a \leftarrow -\mathbf{h}_a - \mathbf{x}_a$;
    end if
end for
```

Geometrically, it will always obtains $\mathbf{vmax}$ and $\mathbf{vmin}$ from two diagonal vertices. After finding $\mathbf{vmax}$ and $\mathbf{vmin}$, the algorithm then projects $\mathbf{vmax}$ and $\mathbf{vmin}$ to the $\mathbf{n}$ by dot product (i.e., $\mathbf{r}_{max} = \mathbf{vmax} \cdot \mathbf{n}$ and $\mathbf{r}_{min} = \mathbf{vmin} \cdot \mathbf{n}$). This test report ”non-overlapping” if $\mathbf{r}_{max}$ and $\mathbf{r}_{min}$ are both positive or both negative. The following pseudocode implements the above logic using early termination.

```
if $\mathbf{vmin} \cdot \mathbf{n} > 0$ return ”non-overlapping”
if $\mathbf{vmax} \cdot \mathbf{n} \geq 0$ return ”overlapping” (if not last test, plane and AAB only!)
return ”non-overlapping”
```

If any of these tests report non-overlapping, then the triangle and the AAB are not intersected. Otherwise they are overlapping. They put the ”(9 tests)” first because empircally it makes entire test faster (it may be seen as early termination code optimization).

We also developed a similar voxelizer using AABB-triangle intersection test, but without octree space subdivision. Instead, for each triangle, we linear search the voxel range that the triangle is located at. This process is similar to scan conversion in three dimensional space. After we found the range, we extend the range by two voxels in all six directions. Finally, we examine the voxels within the range by using the box-triangle intersection test. If the test reports “overlapping”, we mark the corresponding voxel as one to indicate occupancy. Figure 3.6 demonstrates some examples of this process.

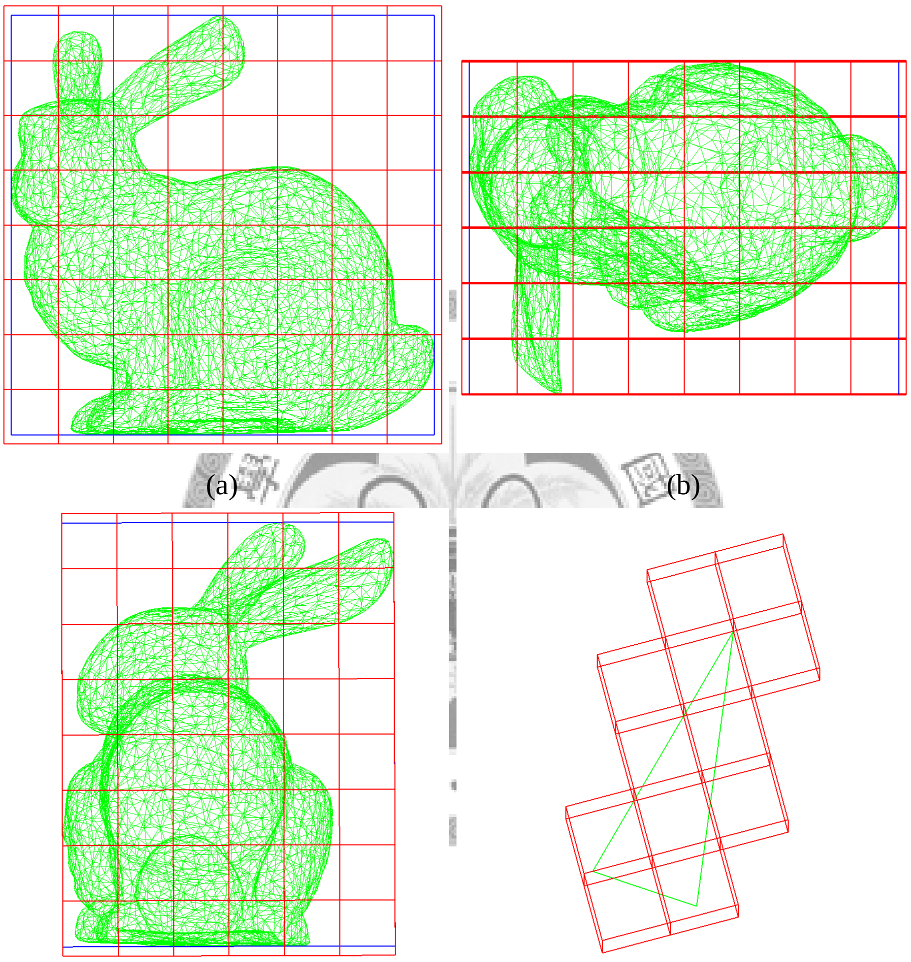


Figure 3.6: Voxelization by box-triangle intersection test. (a), (b), (c) shows red voxelization grids, the blue box is the tight bounding box of the input mesh model. Note that the starting position of the voxelization needs adjustment. (d) shows scan converting box-triangle intersection test (Red boxes are also intersected boxes).

We compared two voxelization algorithm, and discovered there are pros and cons in both methods. The ray-triangle voxelizer can produce visually more similar voxelized 3D model than the box-triangle voxelizer under the same voxelization resolution. But it also produces many holes in the voxelized model. On the other hand, the box-triangle voxelizer is just the opposite. It may be more desirable to have a more visually similar voxelized model than a fewer holes one. Although models with fewer holes usually make entire sculpture more stable, these thicker models also use more LEGO bricks implicitly, thus reduce the chance of completing a LEGO sculpture. See Figure 3.7 for comparision and evaluation.

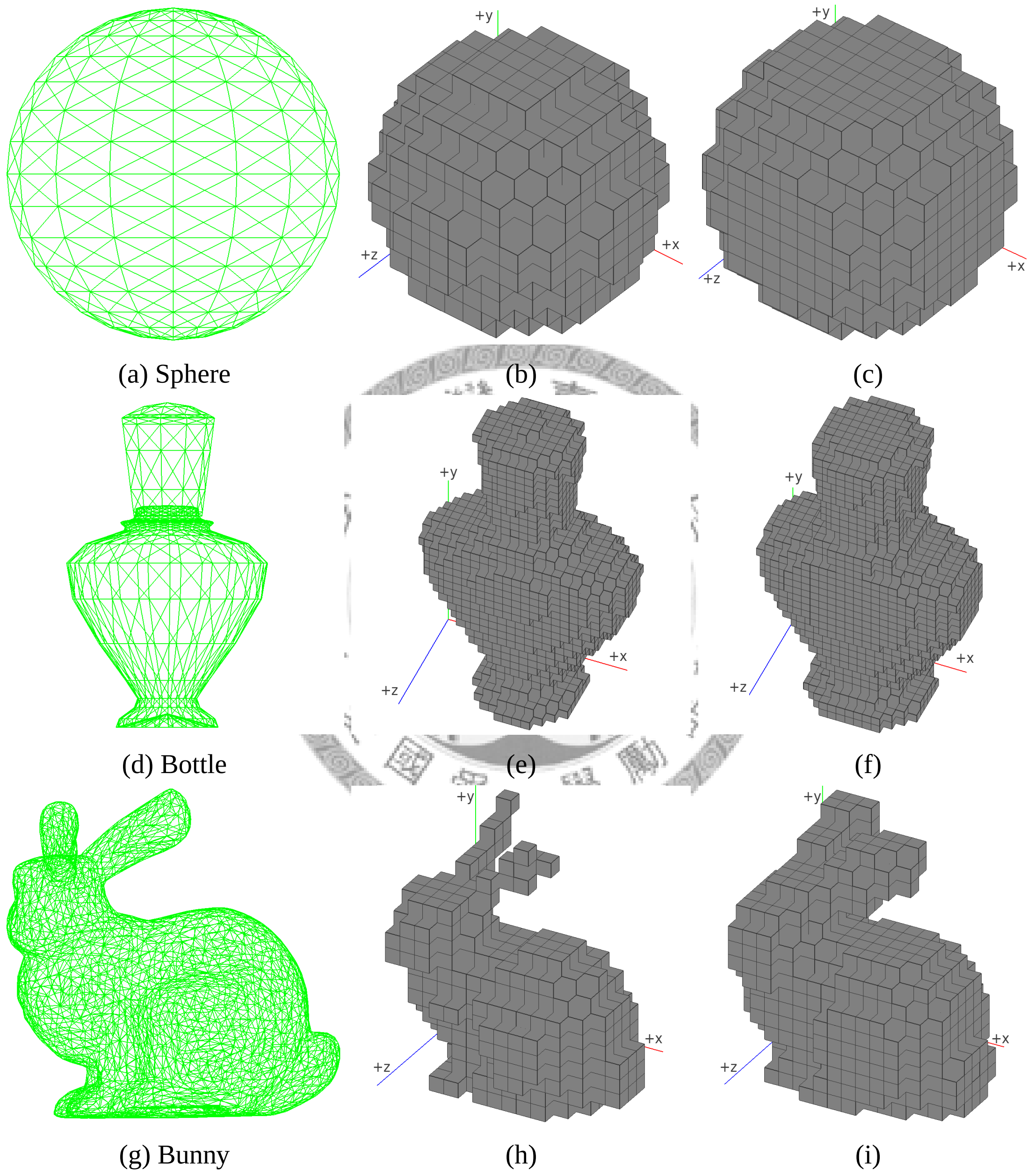


Figure 3.7: Voxelization results. Left column is the original triangular mesh models. Middle column is voxelized using ray-triangle intersection test. Right column is voxelized using box-triangle intersection test. Both voxelization uses same resolution setting. (a), (b), (c): Sphere, resolution = (x: 12, y: 12, z: 12). (d), (e), (f): Bottle, resolution = (x: 23, y: 38, z: 12). (g), (h), (i): Bunny, resolution = (x: 15, y: 15, z: 12).

The observed dilemma between two voxelization algorithms make us wonder if this is really a tradeoff (see Figure 3.8 for artifact illustration), or there will be a hybrid approach to further improve the voxelized model. To ease the explanation, we first define two terms, VM_{ray} and VM_{box}, to represent the two voxelization results obtained from previous stage. VM_{ray} and VM_{box} are both 3D matrices representing voxel occupation. They are consist of only zeros and ones. The ones represent occupied voxels, while the zeros represent empty void voxels. VM_{ray} is voxelized by ray-triangle intersection test. VM_{box} is voxelized by box-triangle intersection test.

After we obtain two voxelization results, we combine the results into one voxelized model using the following three repairing procedures in attempt to solve the dilemma.

1. Make the interior of shell-only voxelized model VM_{box} solid, thus obtainin $VM_{solidbox}$.

2. Improve neighborhood connectivity of VM_{ray} within 3x3x3 range using $VM_{solidbox}$ as an aid.

3. Fill the fragmented interior and surface holes. If a hole is surrounded by $VM_{solidbox}$ neighboring voxels.

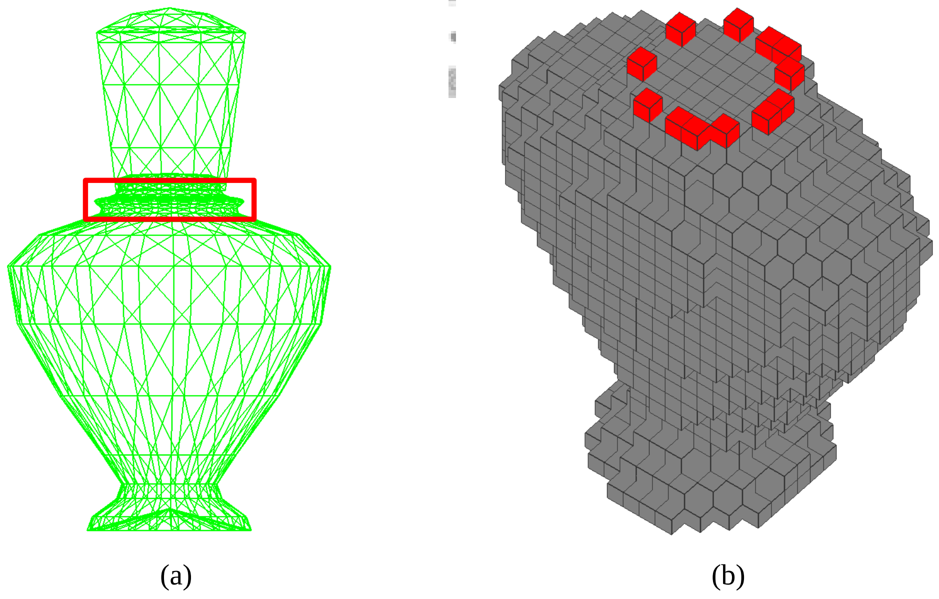

(a) (b)

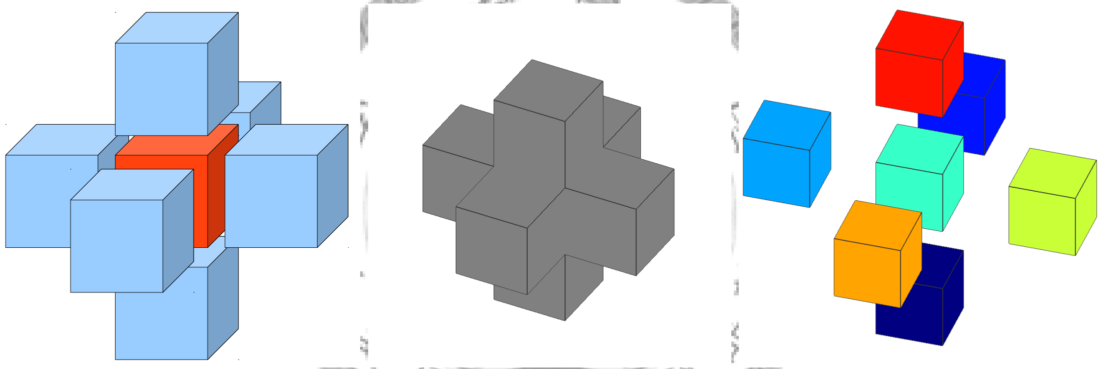
Figure 3.8: Artifacts of a ray-triangle voxelized model. (a) and (b) show the flower bottle has a failure voxelization part in ray-triangle intersecting voxelization, but not in the box-triangle intersecting voxelization.

If users supply voxel-based 3D model directly, then the above three repairing procedures can still be used, but there will be no distinction between VM_{box}, VM_{ray} and $VM_{solidbox}$ since there is only one voxelized 3D model. In other word, these procedures use the input itself as the reference voxelization to repair itself. We have experimented that it will still achieve some repairing result.

Before explaining each step of propressing, we define "voxel connectivity" as follows:

Definition 1. Voxel Connectivity (see Figure 3.9)

A voxel at (x, y, z) is automatically connected to its six 3D direction neighboring voxels (i.e., voxels at coordinates $(x-1, y, z)$, $(x+1, y, z)$, $(x, y-1, z)$, $(x, y+1, z)$, $(x, y, z-1)$ and $(x, y, z+1)$).



(a) (b) Connected voxels (c) Disjoint voxels

Figure 3.9: Voxel connectivity. (a) shows the spatial neighboring relationship for a voxel to check its connectivity. (b) and (c) are examples in six direction.

In mathematical graph theory, if we view a voxel as a node, then the Definition 1 means that the node always equipped with edges toward six 3D direction. This implies that if a voxel is nearby another voxel in 3 dimensional Euclidean geometry sense, then they are automatically connected. These edges can be directed or undirected, because if two voxels are connected using directed edges, it forms a bidirectional edge.

After we know how to express the connectivity of a voxel, we can then devise an connected component algorithm, **VOXEL_CONN_COMP**, based on depth first search. It takes a voxelized 3D model representation Vm consist of zeros and ones. The algorithm will return how many connected components the voxelized 3D model has, and a 3D matrix $Setid$ consist of component set integer identifiers. The $Setid$ has the same size of Vm, and each unique integer in $Setid$ is a ranking order according to the size of each connected component. In other words, The smaller the $Setid(x, y, z)$ is, the larger the connected component is. The ranking order property of $Setid$ can be used to map each identifier to a color. We use Jet color mapping for this purpose. Therefore, if the color is close to red, then the connected component has few voxels. If the color is close to blue, then the connected component has many voxels. This visualization will be used throughout the entire paper, as in Figure 3.10. The running time does not put pressure on our system, therefore we use the algorithm in many other parts of our system.

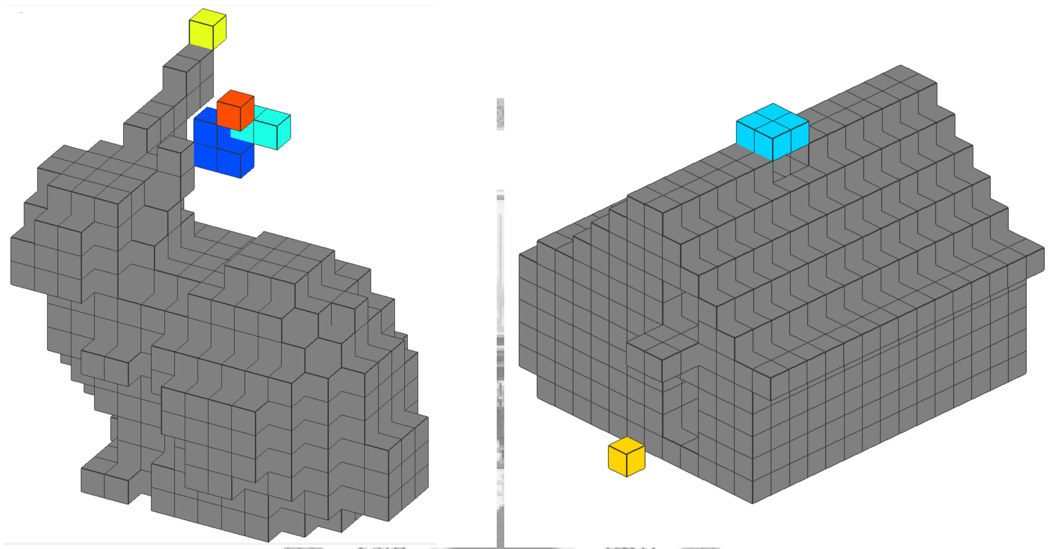

(a) (b)

Figure 3.10: Voxel connected components. The color code in (a) and (b) represent different connected components of the voxels, and the gray color often represent the main connected component (that is, a connected component with most number of voxels). (a) shows the bunny has broken ears, but thay can be repaired by simply adding a voxel nearby to achieve fully connected voxels. (b) shows the yellow and cyan voxels are isolated from the main connected component.

The repairing procedure 1 is essentially a procedure to solidify a voxelized model. This procedure is mainly used after voxelization using AAB-triangle intersecting voxelizer, because it only tests against the triangles used to described the surface of a 3D object. It does not detect whether a voxel is in the interior/exterior of a 3D model. The algorithm 2 describes the above procedure.

The above procedure do successfully solidify the AAB-triangle voxelized 3D models. The reason why it works is that there are always some of rays, shot from the exterior

Algorithm 2 Soldify shell-only voxelized 3D model

```
Require: A voxelized 3D model $Vm$ with no holes on the shell surface.
                                        ▷ (i.e., a fully enclosed 3D model)
  for all $Vm(x,y,z) = 0$ (empty void voxels) do
      $hitcount \leftarrow 0$
      for all rays shooting toward 6 direction do
          if the ray hit an occupied voxel
              $++hitcount$
      end for
      if $hitcount = 6$
          $Vm(x,y,z) \leftarrow 1$
  end for
```

empty void voxels , hitting nothing but continuing travels. While the interior empty void voxels, with the enclosure assumption, will make *hitcount* reaches six. The results are shown in the Figure 3.11.

After we obtain solidified voxel representation, $VM_{solidbox}$, we use it to perform repairing procedure 2. As previous section mentioned, VM_{box} is often thicker than VM_{ray}, while VM_{ray} often produces the artifacts like Figure 3.8. This observation strongly suggests that we should devise a heuristic voxel connectivity repairing algorithm using a thicker reference $VM_{solidbox}$ in attempt to eliminate the disconnecting artifacts in VM_{ray}. The method is described in the algorithm pseudocode 3. Basically, it will utilize the previously described voxel connected component algorithm to detect if there is a voxel at $VM_{solidbox}(x,y,z)$ and adding it to $VM_{ray}(x,y,z)$ cause the voxel connectivity of nearby 3x3x3 space dropping, then we add it. Otherwise we reject this adding.

Algorithm 3 Voxel connectivity repair using a thicker reference $VM_{solidbox}$

```
  for all $VM_{ray}(x,y,z) = 0$ (empty void voxels) do
      $VM_{3x3x3} \leftarrow$ extract nearby 3x3x3 of $VM_{ray}$ at $(x,y,z)$
      $oldConnectivity \leftarrow \mathbf{VOXEL\_CONN\_COMP}(VM_{3x3x3})$
      $VM_{ray}(x,y,z) \leftarrow 1$
      $VM_{3x3x3} \leftarrow$ extract nearby 3x3x3 of $VM_{ray}$ at $(x,y,z)$
      $newConnectivity \leftarrow \mathbf{VOXEL\_CONN\_COMP}(VM_{3x3x3})$
      if $newConnectivity \geq oldConnectivity$
          $VM_{ray}(x,y,z) \leftarrow 0$                    ▷ reject the adding.
  end for
```

We use a reference voxelization $VM_{solidbox}$ because simply adding voxels for connec-

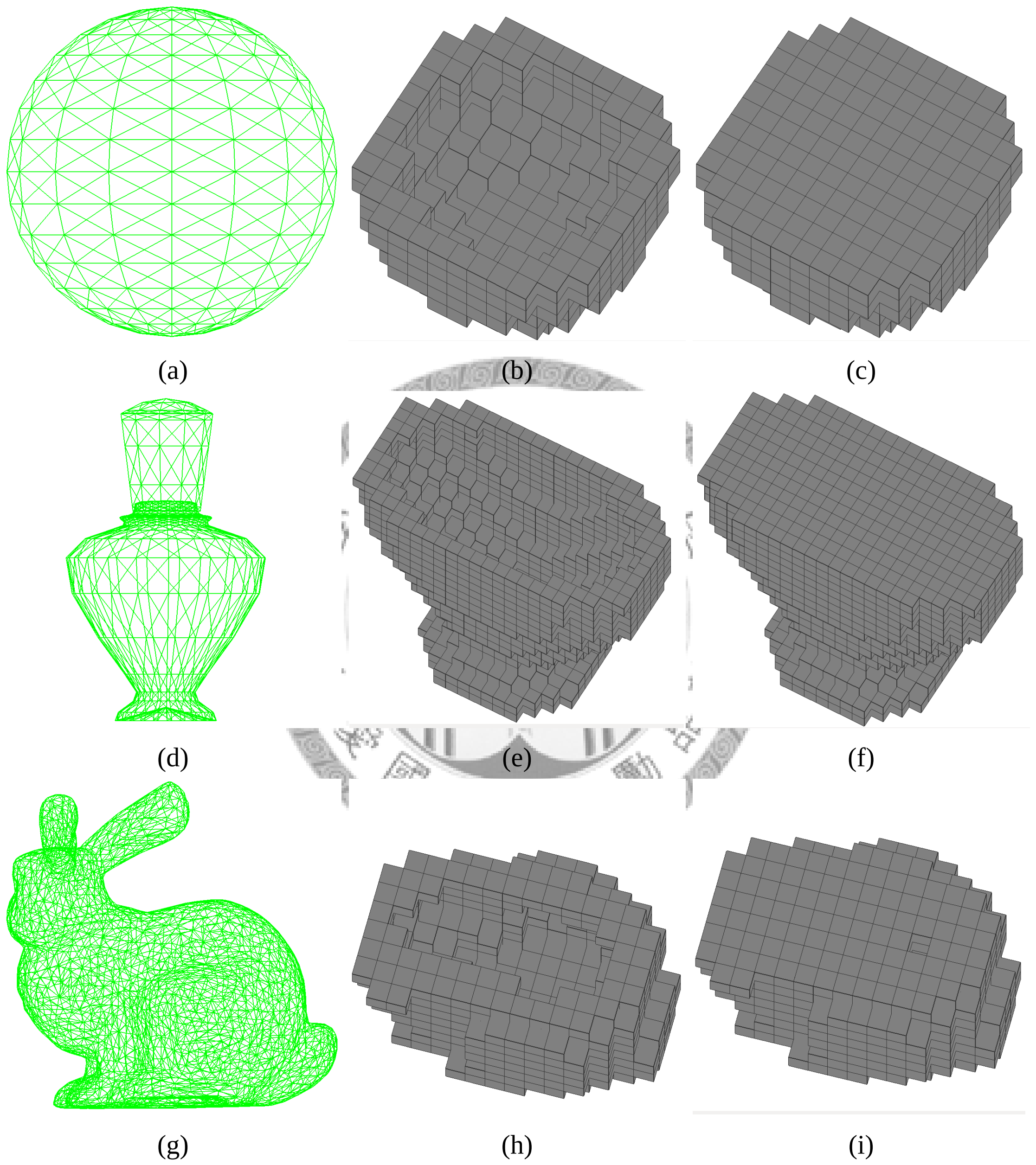


Figure 3.11: Solidify a voxelized 3D model. The resolutions of these models are the same as Figure 3.7. The bunny has a failure voxel inside, it is because the model has bottom holes which violate the assumption of our procedure.

tivity will not respect to the visual similarity. And we use $VM_{solidbox}$ instead of VM_{box} because there are extreme cases that the shell of VM_{box} does not intersect with the shell of VM_{ray}. These cases may occur if the intersection of triangle is very close to the boundary of a voxel.

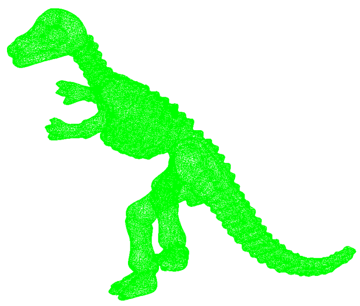
The results are shown in Figure 3.12. The algorithm often successfully repairs the voxelized model with long parts. These models are often humanoid characters, biped animals, and so on. Models with smoother surface often do not need such connectivity repairing. Because we only apply the connectivity repairing within 3x3x3 cube space (that is, it checks the Definition 1 and diagonal voxels). It cannot actually connect the case shown in Figure 3.9 (c), since their 3x3x3 space do not include each other.

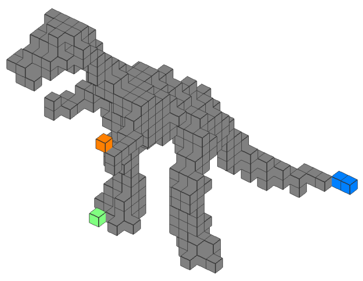
The repairing procedure 3 also takes the advantage of the observation that VM_{ray} is thicker than $VM_{solidbox}$. It could happen that VM_{ray} will be entirely included by $VM_{solidbox}$, and the we can use the solid interior of $VM_{solidbox}$ to repair surface and interior holes of VM_{ray}. The algorithm 4 implements this procedure.

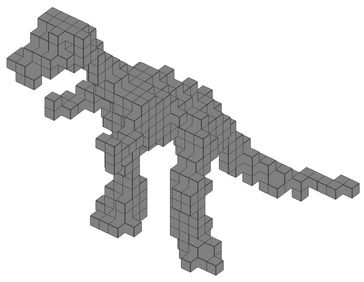
Algorithm 4 Repair surface/interior holes of VM_{ray} using a thicker reference $VM_{solidbox}$

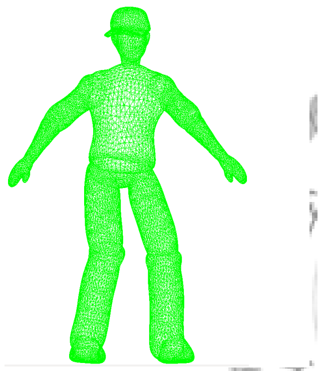
```
for all $VM_{ray}(x, y, z) = 0$ (empty void voxels) do
    $surroundingCount \leftarrow 0$
    (check if $(x, y, z)$ is surrounded by $VM_{solidbox} = 1$ in six directions):
    if $VM_{solidbox}(x - 1, y, z) = 1$        $++surroundingCount$
    if $VM_{solidbox}(x + 1, y, z) = 1$        $++surroundingCount$
    if $VM_{solidbox}(x, y - 1, z) = 1$        $++surroundingCount$
    if $VM_{solidbox}(x, y + 1, z) = 1$        $++surroundingCount$
    if $VM_{solidbox}(x, y, z - 1) = 1$        $++surroundingCount$
    if $VM_{solidbox}(x, y, z + 1) = 1$        $++surroundingCount$
    if $surroundingCount = 6$                  $VM_{ray}(x, y, z) \leftarrow 1$
end for
```

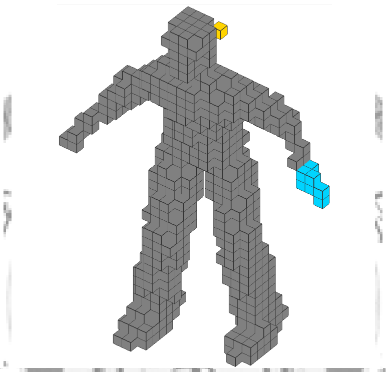
In fact, we not only check $VM_{solidbox}$ but also check VM_{ray} for surrounding. This is just for some subtle cases that the $VM_{solidbox}$ is not successfully solidified. The algorithm successfully applied to different resolutions of the bottle model. At this step, our attempt to repair a voxelization result with artifacts is basically finished. The following preprocessing steps are focused on providing a better initial solution for placement optimization.

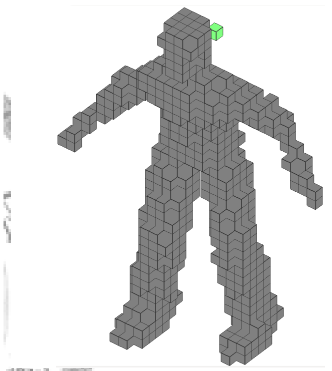

(a) Dinosaur

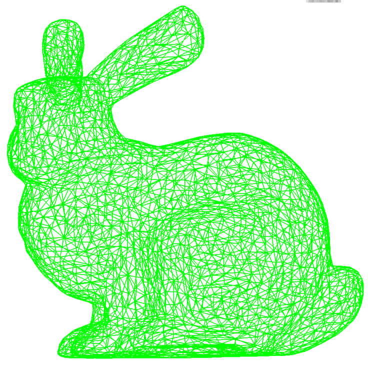

(b) 4 connected components

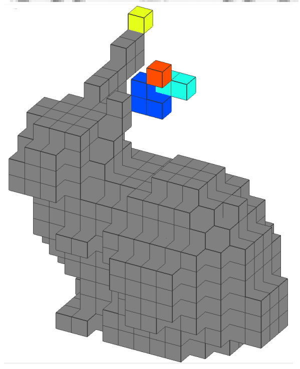

(c) entire fully connected


(d) Child


(e) 3 connected components


(f) 2 connected components


(g) Bunny


(h) 5 connected components

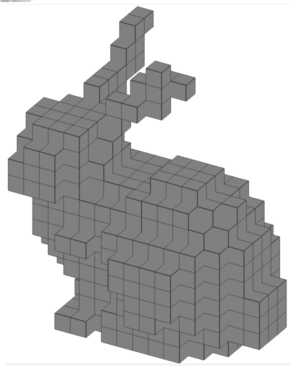
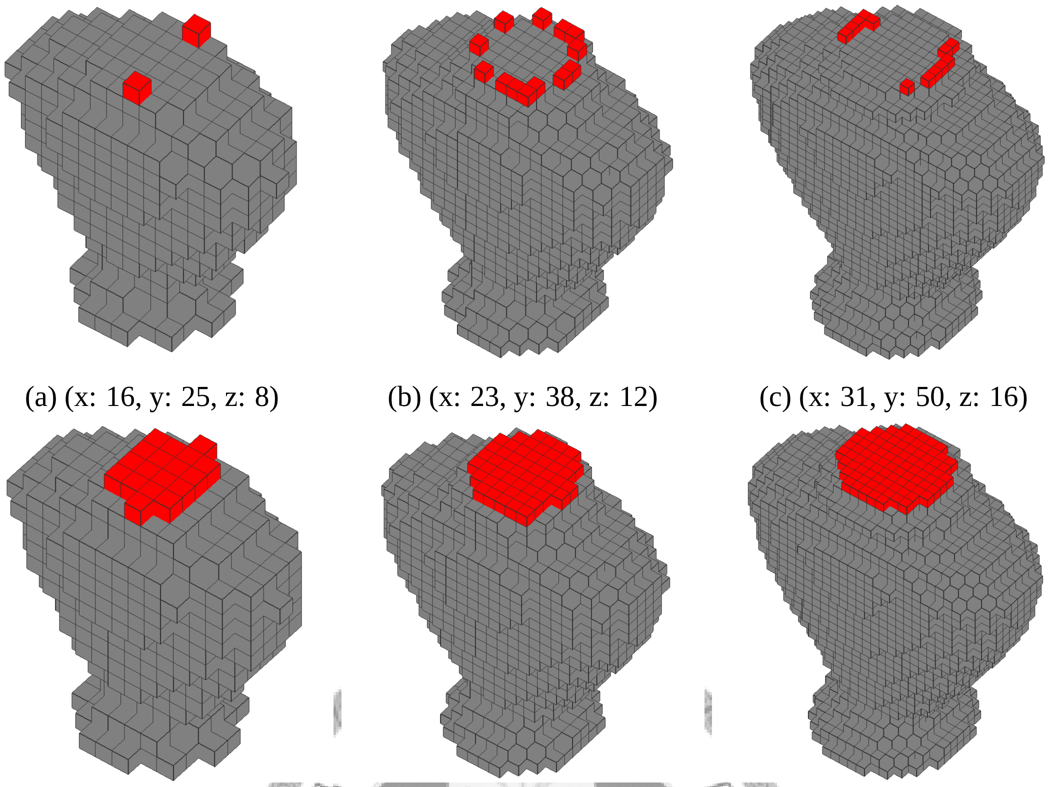

(i) entire fully connected

Figure 3.12: Repair the connectivity of VM_{ray}. Left column is the original triangular mesh models. Middle column is the voxelized models before connectivity repairing. Right column is the voxelized models after connectivity repairing. (a), (b), (c): Dinosaur, resolution = (x: 18, y: 25, z: 26). (d), (e), (f): Child, resolution = (x: 25, y: 37, z: 12). (g), (h), (i): Bunny, resolution = (x: 15, y: 15, z: 12). In the child model case, the yellow voxel in (e) cannot be rescued due to 3x3x3 cube space constraint.



(a) (x: 16, y: 25, z: 8) (b) (x: 23, y: 38, z: 12) (c) (x: 31, y: 50, z: 16)

(d) (e) (f)

Figure 3.13: Repair surface and interior holes of VM_{ray}. First row shows the broken part of the “Bottle” model. The captions in (a), (b), and (c) are their voxelization resolution. Second row shows the repaired results.

3.3 Preprocessing

Although there are previous voxel repairing efforts, still we are unable to fully repair all kinds of artifacts in these voxelized models. The remaining isolated voxels, as shown in Figure 3.10, will never connect to the main trunk of LEGO sculpture. If we do not get rid of these voxels, we cannot clearly determine our legolized results are valid or not in the later placement optimization step. In addition, albeit we change the cost optimality goal into quantity constraint goal, it is still worth making our voxelized model as hollow as possible. Especially for large LEGO construction projects, removing interior voxels can effectively minimize time and money spent on these projects.

The above two factors lead to the design of the following two operations.

1. Remove isolated voxels automatically.

2. Remove the interior voxels of the combined result, thus making it hollow.

The preprocessing step 1 is basically removal of smaller isolated connected components. As our **VOXEL_CONN_COMP** algorithm will find the major connected component with most number of voxels, this step is as easy as reuse of the **VOXEL_CONN_COMP** algorithm. After cleaning fragmented voxels, we eliminate one of reasons that could fail the placement optimization.

The preprocessing step 2, removing the interior voxels to make the voxelized model hollow, is the inverse operation of making model solid. However, the algorithm is quite different. Instead of traversing all empty void voxels, we use ray-tracing like algorithm to traverse voxels. Similar to the voxelization algorithm by ray-triangle intersection test, we also use parity counting to determine the interior, but with a higher counter bound for classifying interior between shell. The ray-tracing way provides the flexibility to adjust the thickness of the hollow voxelized model for every side, since we perform six sides ray-tracing to determine the interior. The algorithm requires one assumption, that is, the input voxelized model must be fully solidified (i.e., no holes inside the model). The procedure is implemented in Algorithm 5.

Algorithm 5 Remove interior voxels of a voxelized model Vm.

```
Require: A voxelized 3D model $Vm$ with no holes inside it.
                                        ▷ (i.e., a fully solidified 3D model)
  initialize 3D matrix $countSix$ to zero.
  for all ray shooting directions $\in \{+x, -x, +y, -y, +z, -z\}$ do
      for all $pixel(i, j) \in$ corresponding plane perpendicular to the ray do
          Get 3D coordinate $(x, y, z)$ by interpreting $pixel(i, j)$ and the ray.
          if $Vm(x, y, z) = 0$ then
              $countThickness \leftarrow 0$
          else
              $++countThickness$
              if $countThickness > thickness_{rayDirection}$
                  $++countSix(x, y, z)$
          end if
      end for
  end for
  $Vm(x, y, z) \leftarrow 1, \forall\, countSix(x, y, z) = 6$
```

The hollowed-out voxelized model results can be seen in Figure 3.14. We found that it is sufficient for placement optimizer to run in thickness of two voxels. If we use only one voxel thick, the shell of voxelized model will produce zigzag structure. These diagonal artifacts will greatly reduces the chance to use LEGO bricks connect the entire voxel representation. And making the voxelized model thicker does not gain any benefit but increase the consumption of LEGO bricks.

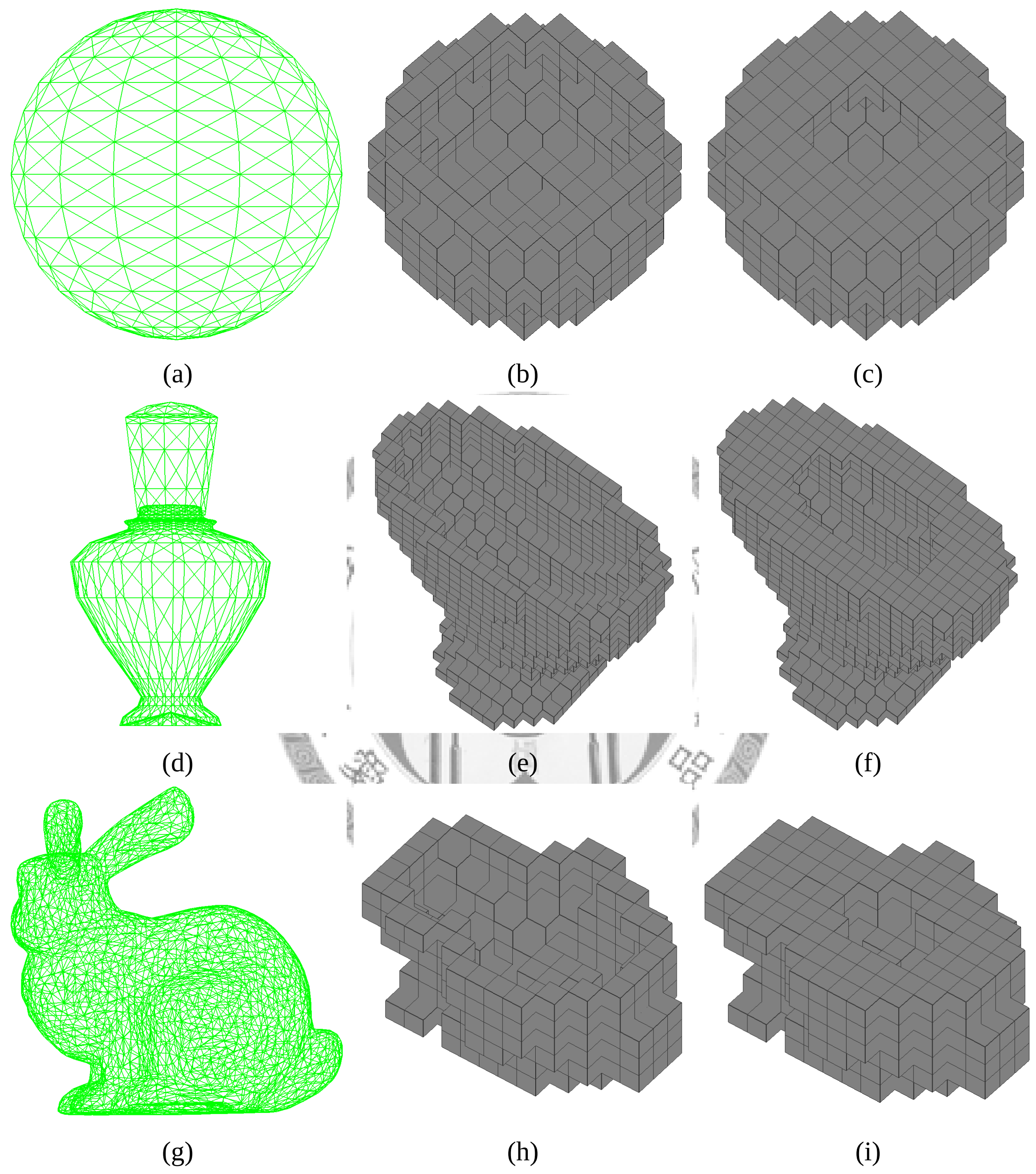


Figure 3.14: Hollowing out a voxelized 3D model. The resolutions of these models are the same as Figure 3.7. Left column is the original triangular mesh models. Middle column is the voxelized models of one voxel thick. Right column is the voxelized models of three voxel thick.

3.4 Placement Optimization

After voxelization and preprocessing stages, our voxelized model is ensured to be a single connected voxel representation according to the Definition 1. But such a model cannot be considered as “legolized”. A simple attempt of legolizing the voxelized model is to map every voxel into generic 1x1 LEGO brick. This attempt, however, is definitely unable to succeed. Because this will only give you many disjoint 1x1 LEGO pillars, and some of pillars are even floating in the air, as Figure 1.2 shows. A truly legolized object should at least satisfy the connectivity-one objective, which is one of three goals we want to achieve in order to solve the “LEGO construction problem”.

But before we describe our approach of placement optimization, we first have to define what is the connectivity between LEGO bricks. As we did in the Preprocessing section, in this way we can have a better understanding of our problem in terms of mathematical graph theory.

Definition 2. LEGO Brick Connectivity (see Figure 3.15)

A LEGO brick is connected to the bricks directly above it and the bricks below it. If a LEGO brick X has a dimension of $(width \times height \times length)$ and positioned at (x, y, z) using the lowest voxel part of LEGO brick as the pivot voxel. Then we know the LEGO brick X has two rectangular interlocking surfaces. They can be expressed as a voxel coordinate set $S = \{(x_r,\ y-1,\ z_r)\} \cup \{(x_r,\ y+height-1,\ z_r)\}$, where $x \leq x_r < x+width$ and $z \leq z_r < z+length$. Any LEGO brick covers these coordinates in S are said to connect to the LEGO brick X.

The above definition assumes the LEGO bricks are cuboid shapes. In graph theory terms, if we view a LEGO brick as a node, then the Definition 2 means that the node always equipped with edges toward above and below. The LEGO brick connectivity definition is somewhat more strict than the voxel connectivity definition, as LEGO bricks only allow vertical connection instead of horizontal connection. Again, the edge can be directed or undirected using the same bidirectional edge argument in Definition 1. But directed edges can be use for computing a better building order, later in the section we

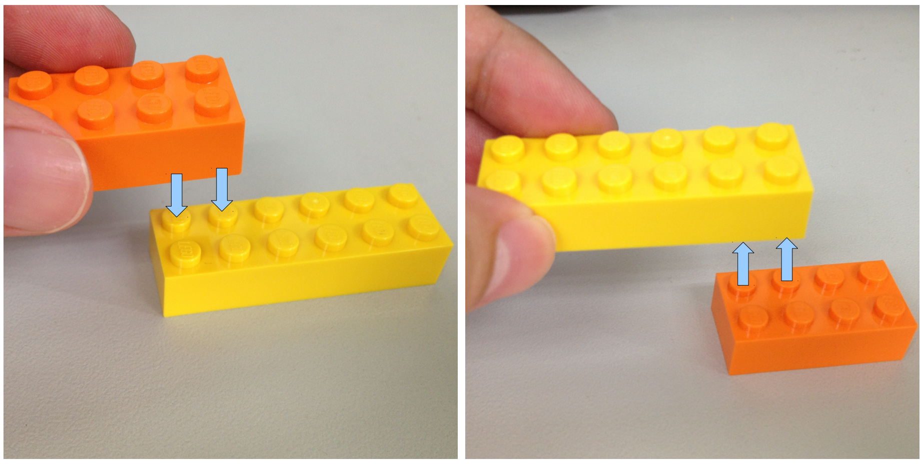

(a) Connect from above (b) Connect from below

Figure 3.15: LEGO brick connectivity.

will briefly discuss how to obtain a better building order.

Like we can devise a connected component algorithm based on depth first search on voxels, we can also design a similar algorithm, **BRICK_CONN_COMP**, to compute the connectivity of entire legolized 3D structure. But instead of using 3D matrix $SetId$, we use a hash table for this purpose. This is because our legolized representation data structure LM is designed by a composition of 3D pointer matrix Pm and a linked list $brickList$ storing information for all actual LEGO bricks. Each $Pm(x, y, z)$ is actually a pointer to one of items in $brickList$. In this way, we can conveniently reference the LEGO brick information stored in the $brickList$ by using $LM(x, y, z) = Pm(x, y, z)$. In addition, pointers provide unique item referencing, which is suitable for being as hash keys. This explained the reason why we use hash table to store $SetId$.

As mentioned in the Introduction chapter, this NP-hard problem is not easy to solve. After our related work survey, it is typical to formulate the problem as an placement optimization problem, and solving it using optimization techniques. We also follow this common approach as our final stage of legolizer. Among all optimization techniques, we follow the "Cellular Automaton" method described in the paper [5] and implemented our own version with some modifications.

Basically, they formulate the LEGO placement optimization problem in a "Cellular Automaton" way. Then the optimization process is just the same as the evolution process of a cellular automaton simulation. To formulate a problem solvable by cellular automaton simulation, it typically requires to define two elements.

1. What is a cell and its states?

2. How a cell to interact with nearby cells?

For optimization problems, we require two more elements, and thus become a optimization algorithmic framework called “cellular automaton optimization”:

3. How “healthy” is a cell?

4. When to stop optimization?

To formulate, they map the concept of a cell to a LEGO brick. The operations for interacting neighborhood cells are called “merging” and “splitting”. And finally, they have to define a local heuristic energy function to represent the fitness of a cell. In this way, they can evaluate how good is the placement of a LEGO brick.

We later found that the properties of cellular automaton are very suitable for LEGO construction problem. These properties are:

- Parallelism. Each cell is independent and isolated, it provides the opportunity of data parallel computing.

- Locality. Each cell only probes and interacts with its neighboring cells. This is especially important because eliminate the needs to check if a placement is blocked by other LEGO bricks. Instead, we only need to check whether inter-cellular interaction is successful of not. But this property may be also a drawback because we can only compute local cost function. This may lead to local optimal results.

- Homogeneousity. Cuboid LEGO bricks have homogeneous geometric properties, which is just the same as cells in cellular automaton.

We begin to formally define what is a cell for cellular automaton optimization to evolve a 2D layout of LEGO brick placement. This is the main object which we often manipulate in the algorithm. After the definition, we will use “cell” and “LEGO brick” interchangeably.

Definition 3. A cell in cellular automaton (see Figure 3.16)

A cell represents a cuboid LEGO brick. It is defined by a dimension of $(width \times height \times length)$ and positioned at (x, y, z) using the lowest voxel part of LEGO brick as the pivot voxel. The neighborhood of a cell is divided into vertical part and horizontal part. The definition of vertical neighbors are the same as the LEGO brick connectivity definition (Def. 2). The horizontal neighbors are defined by four side of the LEGO brick. That is, it can be defined by a voxel grid set $S = \{(x_r, y_r, z)\} \cup \{(x_r, y_r, z + length - 1)\} \cup \{(x, y_r, z_r)\} \cup \{(x + width - 1, y_r, z_r)\}$, where $x \leq x_r < x + width$, $y \leq y_r < y + height$, and $z \leq z_r < z + length$.

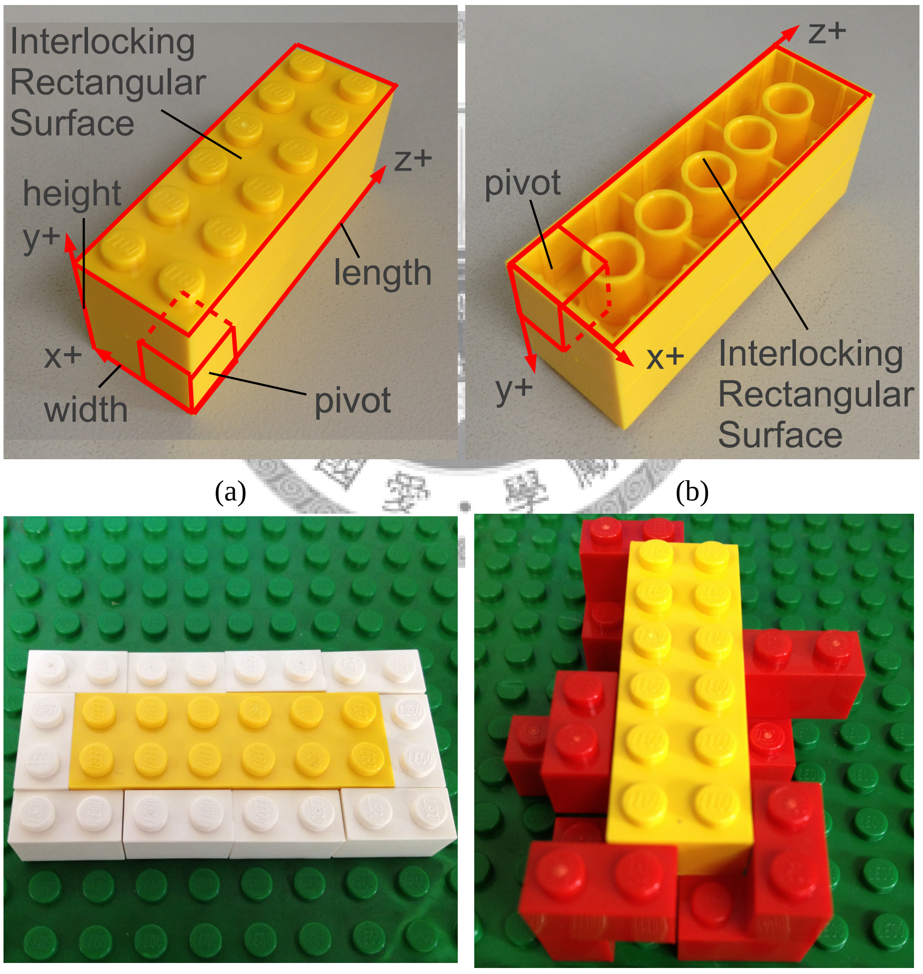

Interlocking Rectangular Surface
height
y+
z+
length
x+
width
pivot
(a)
z+
pivot
x+
y+
Interlocking Rectangular Surface
(b)
(c)
(d)

Figure 3.16: A cell in cellular automaton. The 2x2x6 LEGO brick is synthesized by two 2x1x6 LEGO bricks for illustration. (c) and (d) shows the white and red horizontal neighboring LEGO bricks.

Their initial attempt [5] to encode the problem by a standard cellular automaton is unsuccessful due to LEGO bricks are not homogeneously occupying only one grid but multiple grids. Therefore, they use the concept of “cluster of cells” to futher abstract a LEGO brick. But this abstraction may further complicate the algorithm for readers to understand, we would like to use only “cells” to abstract LEGO bricks.

The LEGO brick family set that we used in our legolizer can be found in Figure 3.17. Note that the dimension notation in the Definition 3 is different from the labels in Figure 3.17. And to ease the implementation, we handle the rotation of bricks by mirroring the dimension. For example, 1x1x2 cell and 2x1x1 cell represent the same LEGO 1x2 brick. In addition, Instead of adding so called L-shape LEGO brick as they [5] did. We strictly do not include the L-shape brick because they are rare in the LEGO basic brick set.

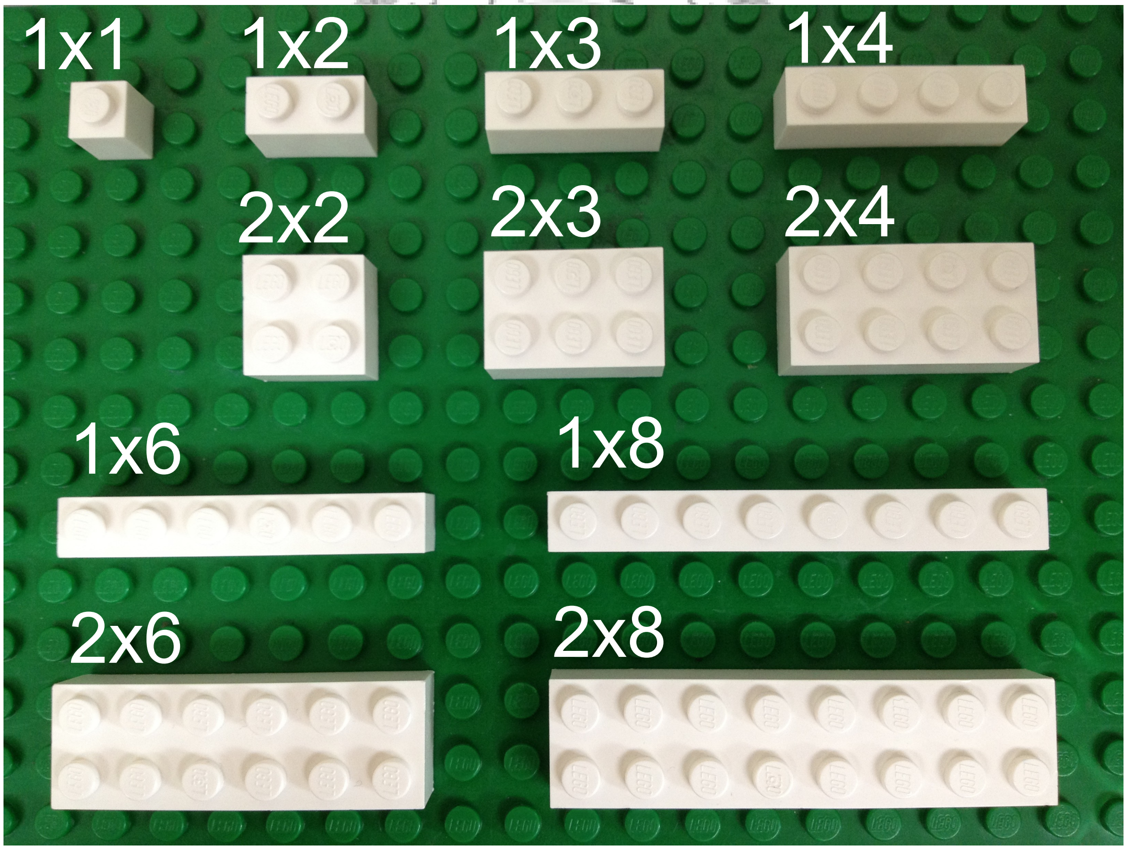


Figure 3.17: LEGO brick family set.

Similar to previous attempts, cellular automaton also uses layer-by-layer decomposition approach to solve the LEGO construction problem. Because most often the *height* of a cell is 1. As this can be seen by the input LEGO brick set, which is only consist of height-one LEGO bricks. This divide and conquer technique is often used to solve complicated optimization problems.

We now describe the procedure of cell automaton optimization first. We defer the explanation of the remaining three elements to the later part of this section. Initially, for all occupied voxels in the 3D voxelized model, we convert them into 1x1 generic LEGO bricks (i.e. $\forall\ Vm(x, y, z) = 1$, we replace it with a corresponding 1x1 generic LEGO brick). Then, for each layer, the algorithm iteratively optimize this layer until a given time step limit is reached. Each time step iteration performs the following 3 phases.

1. For each cell in the layer, probe horizontal neighboring cells to find best mergeable direction $\in \{+x, -x, +z, -z\}$ The "best" term is defined by element 3 - energy function. If there are more than one directions that evaluates the same minimum cost, we randomly choose one of them.

2. Compute all weakly connected components for this layer by viewing each cell's best horizontal mergeable direction as a directed edge pointing to a horizontally neighboring cell. These weakly connected components are called potential mergeable clusters.

3. For each potential mergeable cluster, merge all cells within the cluster. For any cell within the cluster, if the number of failed merge attempts for the cell is exceed a constant, we perform probabilistic split on it.

More precisely, the cellular automaton optimization algorithm can be described using the pseudocode 6. The element 4, termination criteria, is very simple. The variable $\mathbf{MAX\_ITERATION}$ is the time step limit for each layer.

We begin to explain the element 2 - cell interating operations. There are two operations: merge and split. The merging of two cells is defined as following:

Definition 4. Merge operation of two cells (two LEGO bricks) (see Figure 3.18)

Given two cells A and B according to the Definition 3, a merge attempt is successful if A and B satisify one of the following three conditions:

Algorithm 6 Cellular automaton optimization.

```
Require: A single entirely connected voxelized 3D model $Vm$.
                                  ▷ Variables with $LM$ prefix are legolized 3D volume.

  $\forall\ Vm(x, y, z) = 1,\ LM(x, y, z) \leftarrow$ 1x1x1 cell (LEGO brick).
  for $0 \leq y < LM.height$ do
      $LM_{layer} \leftarrow LM.layer(y)$
      $LM_{bestLayer} \leftarrow LM_{layer}$
      $bestLayerCost \leftarrow$ compute layer cost for each cell $c \in LM_{bestLayer}$
      using local heriustic energy function $\mathbf{BRICK\_ENERGY}(c)$.      ▷ (element 3)

      for $0 \leq timeStep <$ $\mathbf{MAX\_ITERATION}$ do
          (phase 1), parallelizable for each $c$:
          for all cell $c \in LM_{layer}$ do
              find $c.bestMergeDirection$ by probing horizontal neighboring cells by
              evaluate after merging energy using $\mathbf{BRICK\_ENERGY}(merged)$.
          end for

          (phase 2):
          compute the set of potential mergeable clusters $S$ by $c.bestMergeDirection$.
                    ▷ This can be solved by weakly connected component algorithm.

          (phase 3), parallelizable for each $s$:
          for all potential mergeable cluster $s \in S$ do
              for all cell $c \in s$ do
                  $mergedNewCell \leftarrow$ merge($c$, getCell($LM_{layer}$, $c.bestMergeDirection$)).
                                                                        ▷ (element 2)
                  if $mergedNewCell \neq$ empty void voxel (merge success) then
                      alter $LM_{layer}$ to reflect the merge .
                  else
                      $++c.mergeFailCount$
                  end if

                  if $c.mergeFailCount >$ $\mathbf{MERGE\_FAIL\_LIMIT}$
                  and pass $c.splitProbability$ test
                       split $c$ and alter $Lm_{layer}$ to reflect the split.      ▷ (element 2)
              end for
          end for

          $currentLayerCost \leftarrow$ compute layer cost for each cell $c \in LM_{layer}$
          using $\mathbf{BRICK\_ENERGY}(c)$.
          if $currentLayerCost < bestLayerCost$ then
              $bestLayerCost \leftarrow currentLayerCost$.
              $LM_{bestLayer} \leftarrow LM_{layer}$.
          end if
      end for
      $LM.layer(y) \leftarrow LM_{bestLayer}$
  end for
```

1. They are completely overlap in YZ plane, and there is such a LEGO brick type with the merged dimension $(A.width + B.width, A.height, B.length)$. The merged cell M has the merged dimension and the pivot coordinate of $(\min(A.x, B.x), A.y, B.z)$

2. They are completely overlap in XY plane, and there is such a LEGO brick type with the merged dimension $(A.width, B.height, A.length + B.length)$. The merged cell M has the merged dimension and the pivot coordinate of $(A.x, B.y, \min(A.z, B.z))$

3. They are completely overlap in XZ plane, and there is such a LEGO brick type with the merged dimension $(A.width, A.height + B.height, B.length)$. The merged cell M has the merged dimension and the pivot coordinate of $(A.x, \min(A.y, B.y), B.z)$

For the extension of color constraint mentioned in the reference work [5], The merge operation must additionally check if two cells are of the same color, or one of cell has the so-called "wildcard" color. Cells with wildcard color are used to relax the constraint, they often reside in the interior of the voxelized 3D model. The cell with non-wildcard color will override the wildcard color after merging.

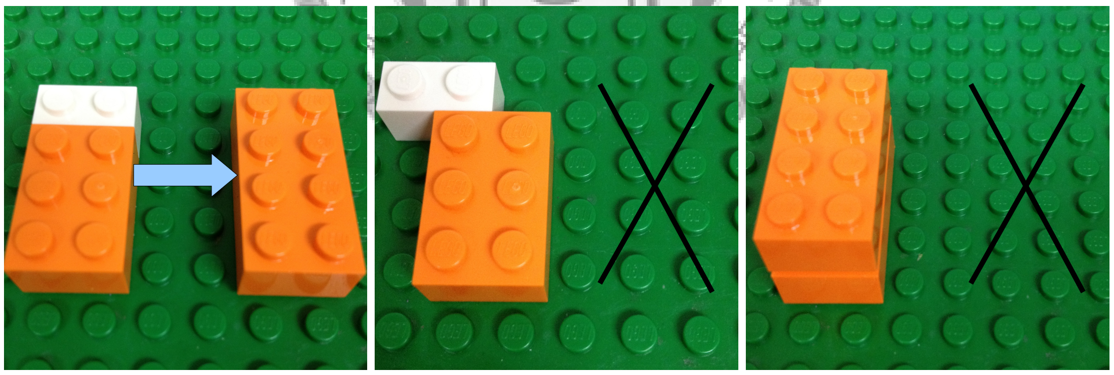

(a) Succeed (b) Failed by mismatch (c) Failed by no such type

Figure 3.18: Merge operation of two neighboring cells. (a) is a successful merge operation, provided that white color is the "wildcard" color. (b) is a failure merge operation because position mismatch leads to not completely overlap. (c) is a failure merge operation, provided that the input LEGO brick set is the same as Figure 3.17.

There are two problems when implementing the merge operation and related algorithm phases (phase 1 and 3). First, how does a cell know the properties of its horizontal neighboring cells? Second, how to smoothly merge cells in a potential mergeable cluster into a real large cell without the aid of L-shaped LEGO brick? (L-shaped brick will make merge operations within a potential mergeable cluster smoothly executed along the path formed by sequence of best mergeable direction).

The first problem is solved by the design of legolized representation data structure LM. As previously mentioned, we use the 3D matrix of pointers Pm to reference cells. We can reference a cell by a volume of pointers $ptrs$. Specifically, given a LEGO brick A stored in LM, $ptrs = \{Pm(x_r, y_r, z_r)\}$ where $A.x \leq x_r < A.x + A.width$, $A.y \leq y_r < A.y + A.height$, and $A.z \leq z_r < A.z + A.length$. Figure 3.19 (a) illustrate the above description. In addition, using this design can simplify the finding of horizontal mergable cells. We can only probe four voxels instead of iterating all horizontal neighboring voxels by Definition 3. Because it can be proven that given two mergeable cells, B and C, C must occupy Pm in one of these four voxel positions: $(B.x - 1, B.y, B.z), (B.x, B.y, B.z - 1), (B.x + B.width, B.y, B.z)$, and $(B.x, B.y, B.z + B.length)$.

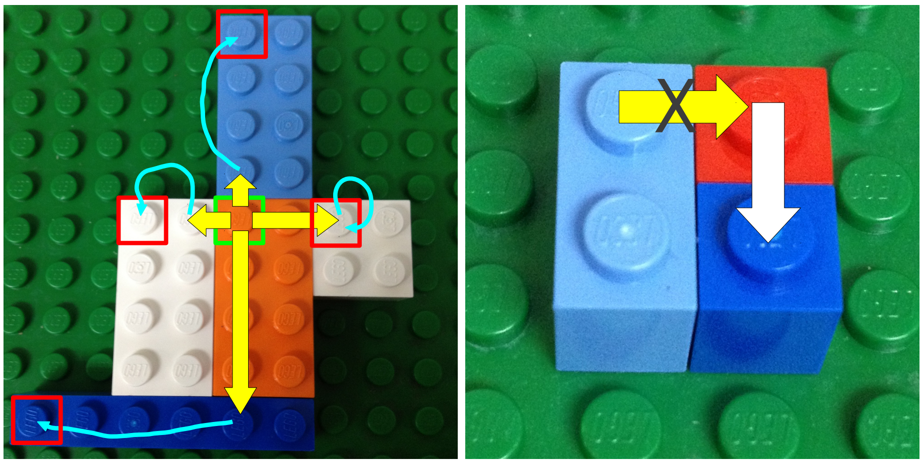
The second problem is basically the current cell cannot merge all of its neighboring cells due to dimension mismatching. We solve it by deferring the merge of current cell. More specifically, we perform merge operation on the neighboring cell first in attempt to get a cell whose dimension is matched for a successful merge. Then we return to merge the original cell whose merging is deferred. We use arrows in Figure 3.19 (b) to explain.

The splitting of a single cell is very simple. It is defined as following:

Definition 5. Split operation of a single cell

A split operation on a single cell is to break it into 1x1 cells. More clearly, Given any LEGO brick A, the split operation turn every voxel (x, y, z) into 1x1 generic standard LEGO brick, where $A.x \leq x < A.x + A.width$, $A.y \leq y < A.y + A.height$, and $A.z \leq z < A.z + A.length$.

As pseudocode 6 mentioned, the split operation is controlled by two parameters, $\mathbf{MERGE\_FAIL\_LIMIT}$ and $cell.splitProbability$. $\mathbf{MERGE\_FAIL\_LIMIT}$ should not be too small, as it may cause splitting too often. The parameter $cell.splitProbability$ is related to the brick size. Usually, the larger the brick, the smaller the probability to trigger a successful split.



(a) (b)

Figure 3.19: Notes for implementing merge operation. In (a), the center orange LEGO probes the four horizontal neighboring cells using the yellow arrows shot from pivot located by green rectangle. The voxels pointed by the yellow arrows then reference to the actual LEGO bricks, like the cyan arrows pointing to their pivots located by red rectangle. The probing result is that the center LEGO can only merge the light blue cell. In (b), we defer the merge led by the yellow arrow, and try the merge led by the white arrow first. After the merge of white arrow is completed, the yellow arrow is now pointing newly merged 1x2 LEGO, which becomes mergeable by the yellow arrow holder.

After we defined the element of neighborhood interaction routines, we begin introducing the local heuristic energy function, the element 3 of cellular automaton optimization. Basically, it is a weighted sum of different energy terms.

Definition 6. Local heuristic energy function ($\mathbf{BRICK\_ENERGY}(cell)$)

Given a LEGO brick $cell$ placed in the legolized 3D volume LM, the cost of this placement, E_{cell}, is defined by the following weighted sum equation [5, 10, 11]:

$$\begin{aligned} E_{cell} = {} & W_{numBricks} \times E_{numBricks} \\ & + W_{dirAltern} \times E_{dirAltern} \\ & + W_{numDistinctColor} \times E_{numDistinctColor} \\ & + W_{numConnBricks} \times E_{numConnBricks} \\ & + W_{areaConnBricks} \times E_{areaConnBricks} \\ & + W_{sameVedge} \times E_{sameVedge} \end{aligned}$$

If any of weight equals to zero, then it effectively disable the corresponding energy term. The energy terms often probe the connecting cells from above and below. The “above” and “below” are following the brick connectivity definition (Def. 2). And the counting of connecting cells are all distinct LEGO bricks, that is, if a vertical connecting LEGO brick covers the current evaluating LEGO brick, $cell$, more than one grids, it should be counted as one rather than the number of overlapping grids. We begin to explain every single energy terms.

The energy term $E_{numBricks}$ is simply computed by counting how many LEGO bricks in the legolized sculpture volume LM. If the energy term is used in the cellular automaton for local cost function, then it refers to the cost of $cell$ itself. We use uniform cost for each kind of LEGO bricks as the main reference work [5] did. In other word, $E_{numBricks} = 1$ for every LEGO brick in LM. We should set $W_{numBricks} > 0$ to minimize this energy term. This term can even be ignored when used in cellular automaton because it does not discriminate the merged LEGO bricks, and merging operation itself implicitly and effectively reduce the consumption of LEGO bricks.

The energy term $E_{dirAltern}$ means the above and below bricks connecting to $cell$ should result in alternative directionality. That is, if the longest dimension of $cell$ is along x-axis, then the above and below bricks connecting to it should have longest dimension along z-axis, and vice versa. Figure 3.20 demonstrate this energy term. The directionality alternation will make the LEGO structure more stable and connected since it follows the principle of masonry. In practice, given the input LEGO brick set, we can compute the longest dimension axis for each kind of LEGO beforehand. $E_{dirAltern}$ is actually equals to the number of LEGO that differs from $cell$ in longest dimension axis. For LEGO bricks with no longest dimension axis (e.g., squared LEGO bricks like 1x1 and 2x2), this term is ignored. We can infer from the description that we should set $W_{dirAltern} < 0$ to maximize this energy term.

The energy term $E_{sameVedge}$ means how many overlapping boundaries for connected LEGO bricks from above and below. Figure 3.21 illustrates this term. The boundaries of LEGO bricks are the reasons why they cannot be horizontally connected. If two connected

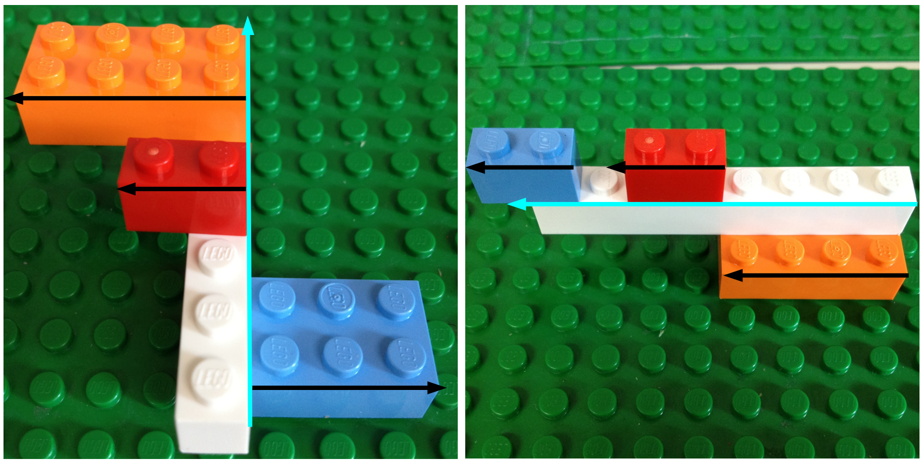
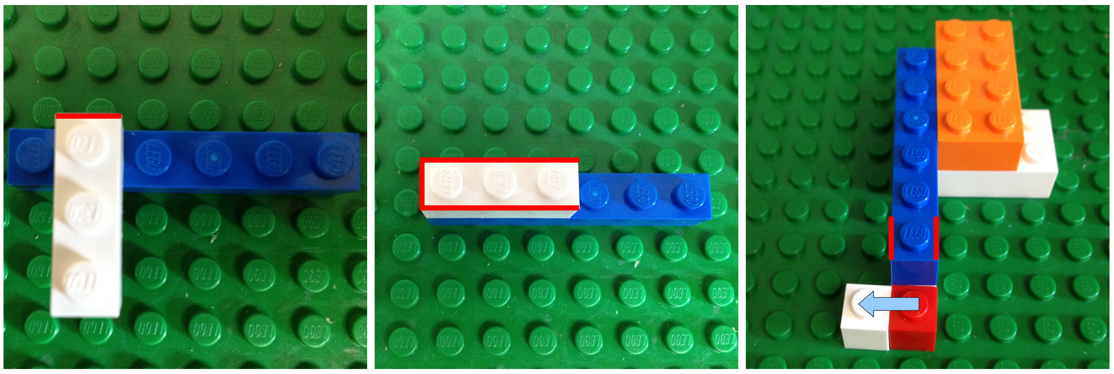

(a) Good placement, $E_{dirAltern} = 3$ (b) Bad placement, $E_{dirAltern} = 0$

Figure 3.20: The energy term $E_{dirAltern}$. The cyan arrows represent the longest dimension axis of $cell$. The black arrows represent longest dimension axes of bricks connecting to $cell$.

LEGO brick has more overlapping vertical cutting edges, it implies that they are covering themselves. As a result, they are tend to less helpful in increasing connectivity. We should set $W_{sameVedge} > 0$ to minimize this energy term. We originally implemented $E_{sameVedge}$ by counting the length of total overlapping edges. However, we discovered this energy term will have the side effect of penalizing $cell$ to cover square-shaped LEGO bricks. Because squared bricks have considerable area covering capability. When $cell$ has to choose a direction from L-shaped choices, this term will cause $cell$ to choose mergeable bricks without covering bricks from above and below. Therefore, we disabled this term in the experiment result by setting $W_{sameVedge} = 0$

The energy term $E_{numConnBricks}$ is computed by total number of bricks connecting to $cell$ from above and below. This term directly reflects the connectivity contribution of $cell$. We should set $W_{numConnBricks} < 0$ to maximize this energy term.

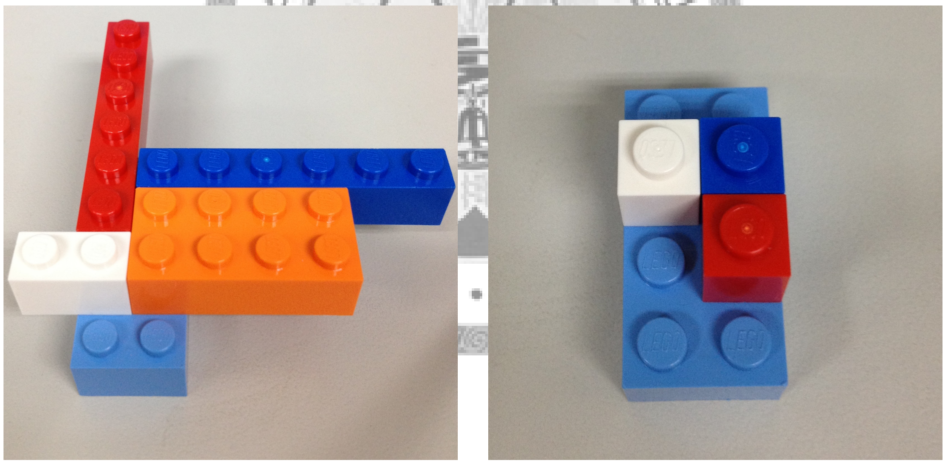
The energy term $E_{areaConnBricks}$ is computed similar to the energy term $E_{numConnBricks}$, but instead of just adding one for each distinct LEGO brick connecting to $cell$, it adds the area of vertical connecting LEGO brick. The area is computed as $width \times length$ It evaluates the connectivity contribution more precisely. We should set $W_{areaConnBricks} < 0$ to maximize this energy term.



(a) Good placement. $E_{sameVedge} = 1$ (b) Bad placement. $E_{sameVedge} = 3 + 1 + 3 = 7$ (c) Side effect. Bad merge direction.

Figure 3.21: The energy term $E_{sameVedge}$. In (a) and (b), *cell* is the blue LEGO brick. The red lines show the overlapping edges that should be penalized for bad placement. However, as (c) shows, this term may make the red LEGO brick to merge the white orphan brick pointed by the arrow, thus increase the risk of disconnection.

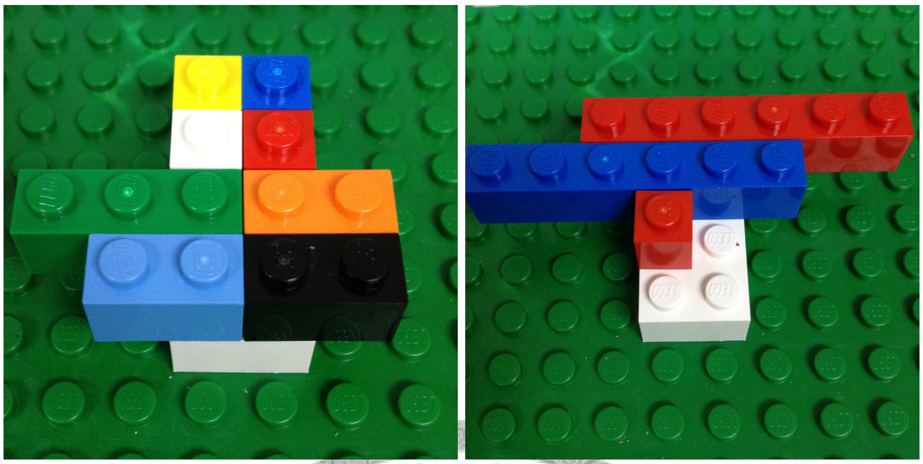
The $E_{numConnBricks}$ and $E_{areaConnBricks}$ terms will drive the cell actively to merge horizontally neighboring cells. Figure 3.22 illustrates the above two energy terms.



(a) Good placement. $E_{numConnBricks} = 4$ $E_{areaConnBricks} = 6 + 6 + 2 + 8 = 22$ (b) Bad placement. $E_{numConnBricks} = 3$ $E_{areaConnBricks} = 1 + 1 + 1 = 3$

Figure 3.22: The energy term $E_{numConnBricks}$ and $E_{areaConnBricks}$. *cell* is the light blue brick on the table.

The energy term $E_{numDistinctColor}$ means to make *cell* cover as many distinct color of LEGO bricks as possible. It is corresponding to the color extension of the reference work [5]. This is a reasonable energy term because color constraint makes the merging operation more difficult to successfully performed, and thus compromise the connectivity. Without this term, it is likely that the entire legolized model will split along line partition by different colors. Note that the “wildcard” color is also treated as one kind of color. We should set $W_{numDistinctColor} < 0$ to maximize this energy term. Figure 3.23 shows the example of this energy term.



(a) Good placement. $E_{numDistinctColor} = 8$

(b) Bad placement. $E_{numDistinctColor} = 2$

Figure 3.23: The energy term $E_{numDistinctColor}$. The white color here represents the wildcard color. $cell$ is the wildcard color sticked to the green plate.

In the original reference work, there was an energy term called $E_{uncovered}$, which counts the interlocking surface area coverage of $cell$ from above and below. That is, if $cell$ has fewer vertical connecting bricks, the more area is uncovered by LEGO bricks and resulting in larger $E_{uncovered}$. However, we feel that this energy term is much less effective, and it may prevent interior cells merge toward boundary. Because boundary bricks often uncovered by other cells to provide shape visual similarity, this term will penalize the outward merging LEGO brick, and thus produce hanging bricks.

In our experiment, our parameters for placement optimization are: $\mathbf{MAX\_ITERATION} = 98$, $\mathbf{MERGE\_FAIL\_LIMIT} = 11$, $W_{numBricks} = +7$, $W_{dirAltern} = -2$, $W_{sameVedge} = 0$ (disabled), $W_{numConnBricks} = +4$, $W_{areaConnBricks} = -1$, and $W_{numDistinctColor} = -2$. Table 3.1 shows the LEGO family set we used in the experiment and its splitting probability paramters for each kind of LEGO brick. The quantity distribution of the LEGO family set is equivalent to two boxes of LEGO 5623 basic brick set. Our table only considers the dimension, all bricks that has the same dimension but different colors will be aggregated into the total quantity of that dimension. The dimension notation is according to the Definition 3, and we do not mirror the type in the table to reflect the actual number. Figure 3.24 shows our cellular automaton optimization in action. It also illustrates our graphical user interface for this computer-aided LEGO sculpture assembly tool.

Type	Quantity	Splitting Probability	Type	Quantity	Splitting Probability
1x1x1	128	0			
2x1x1	280	0.7	2x1x2	212	0.7
3x1x1	40	0.7	3x1x2	76	0.7
4x1x1	32	0.6	4x1x2	76	0.6
6x1x1	20	0.5	6x1x2	14	0.5
8x1x1	14	0.4	8x1x2	8	0.4

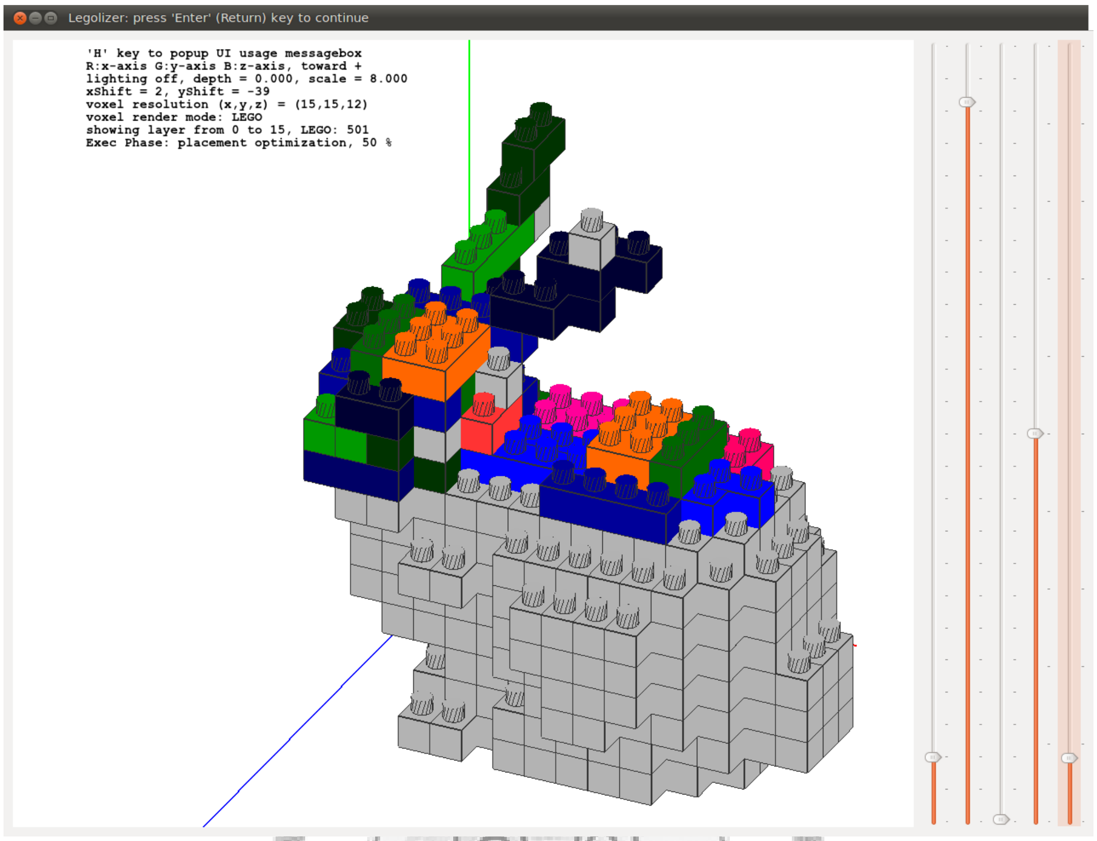
Table 3.1: The LEGO brick family set used in the experiment.

The optimized LEGO placement achieved nearly connected results. However, to further increase the chance of successfully constructing a LEGO sculpture, we implemented following post-optimization refinement procedures:

1. Repair disconnected hanging LEGO bricks on the surface.
2. Simple L-shaped conflict resolution.
3. Re-merge all bricks generated by above two procedures.
4. Adjust LEGO quantity distribution to relax the pressure from rare kinds of LEGO.

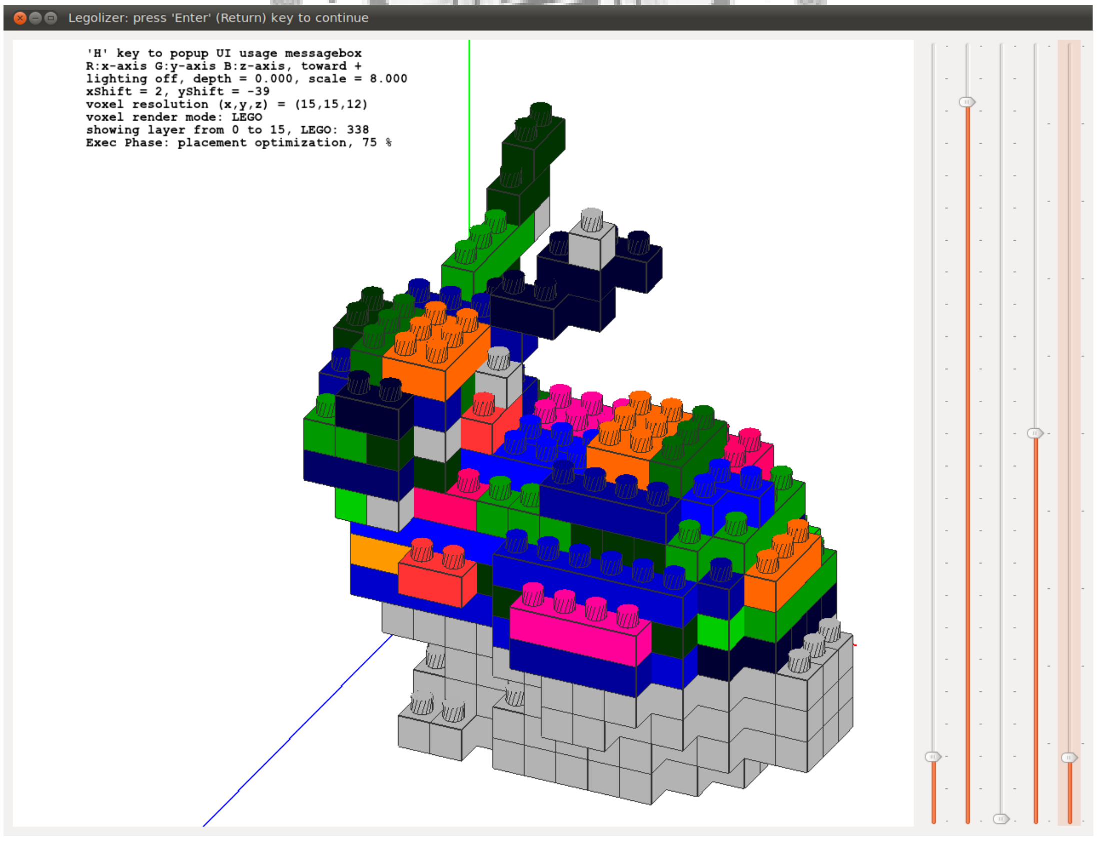
The first procedure is implemented because the optimized placement often consist of fragmented hanging LEGO bricks on the surface, resulting in disconnected legolized model. As shown in Figure 3.27, this kind of artifact is rather easy to repair. We use **BRICK_CONN_COMP** to find disconnected components containing only one LEGO brick. Then, for each of such disconnected LEGO brick, we split it and its nearby horizontal neighboring LEGO brick. The split operation is defined in Def. 5 and the horizontal neighborhood is defined in Def. 3. After splitting isolated LEGO bricks, we will re-merge them by third procedure rather than reusing the cellular automaton optimization. Because we want to completely avoid the artifact produced by it.

The second procedure is an attempt to repair connectivity lost by L-shaped conflicts, as the bunny example in Figure 3.27 (a). The L-shaped conflict example can be found


Legolizer: press 'Enter' (Return) key to continue

```
'H' key to popup UI usage messagebox
R:x-axis G:y-axis B:z-axis, toward +
lighting off, depth = 0.000, scale = 8.000
xShift = 2, yShift = -39
voxel resolution (x,y,z) = (15,15,12)
voxel render mode: LEGO
showing layer from 0 to 15, LEGO: 501
Exec Phase: placement optimization, 50 %
```

(a) layer 8 to 14 optimized


Legolizer: press 'Enter' (Return) key to continue

```
'H' key to popup UI usage messagebox
R:x-axis G:y-axis B:z-axis, toward +
lighting off, depth = 0.000, scale = 8.000
xShift = 2, yShift = -39
voxel resolution (x,y,z) = (15,15,12)
voxel render mode: LEGO
showing layer from 0 to 15, LEGO: 338
Exec Phase: placement optimization, 75 %
```

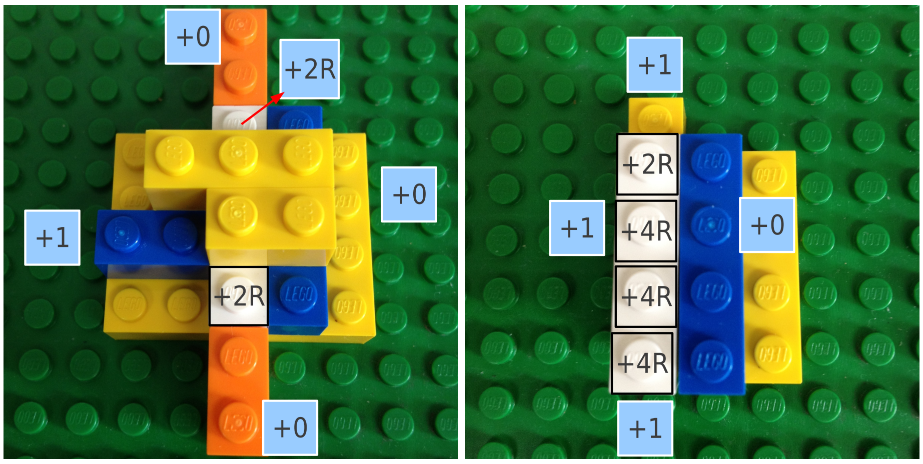
(b) layer 4 to 14 optimized

Figure 3.24: Cellular automaton optimization in action.

in Figure 3.21 (c). What we called "L-shaped conflict" means various problems introduced by the lack of L-shaped LEGO bricks. L-shaped voxels introduce conflicts because any of merge results are unable to connect L-shaped voxels, and usually left a $(1 \times height \times m)$ LEGO brick disconnected. One of resolution methods is to supply a 1x1 brick horizontally that makes L-shaped voxels transformed into squared-shaped voxels. This is what we did in the second refinement procedure. More clearly, we again apply a $\mathbf{BRICK\_CONN\_COMP}$ to identify all connected components by integer $cell.setid$. Then, we scan entire legolized volume to find possible L-shaped conflicts. If an empty void voxel $e(x, y, z)$ is surrounded by disconnected L-shaped bricks (identified using $cell.setid$), then we add one 1x1 LEGO at that position.

The third procedure is the brick re-merging algorithm for the first and the second refinement procedure. In order to avoid disconnecting these hanging bricks again, we use a different approach to guide the merge operation. We define another energy called "risk of disconnection". Like previously mentioned energy terms in optimization, this term also computes cost by vertical connection condition. For empty void voxels vertically neighboring to the brick, we add R for each of such voxel. For 1x1 generic bricks vertically neighboring to the brick, we add $2R$ for each of such brick. $R = \max(\{brick.area \mid \forall\ brick \in \text{LEGO family set}\}) \div \text{current } brick.area$. Unlike previously mentioned energy terms, we also consider horizontal neighboring bricks. We add one for each horizontal neighboring direction that the current brick cannot successfully merge. Figure 3.25 demonstrates how to compute the risk of disconnection of a given LEGO bricks. Please also refer the Definition 3 for related concept.

Provided with the definition of risk above, the algorithm basically performs merging from higher-risk bricks to lower-risk bricks. And the merge must follow the order from higher risk to lower risk strictly (i.e., $cell_a.merge(cell_b)$ is success provided that $cell_a.risk > cell_b.risk$). We use a priority queue to dynamically maintain the risk ordering. Initially, the priority queue is filled with all LEGO bricks from the legolized model LM. Then, we perform the merge described above. Newly merged LEGO brick is inserted into the priority queue. After insertion, it updates the risk of bricks near the



(a) Low Risk, $risk = 4R + 1$ (b) High Risk, $risk = 14R + 3$

Figure 3.25: Risk of disconnection. The white 1x4 brick is the current brick. In (b), there is only one 1x2 yellow brick supporting the center white 1x4 brick. Therefore, $+4R$ is for the added risk for both empty void voxels from above and below.

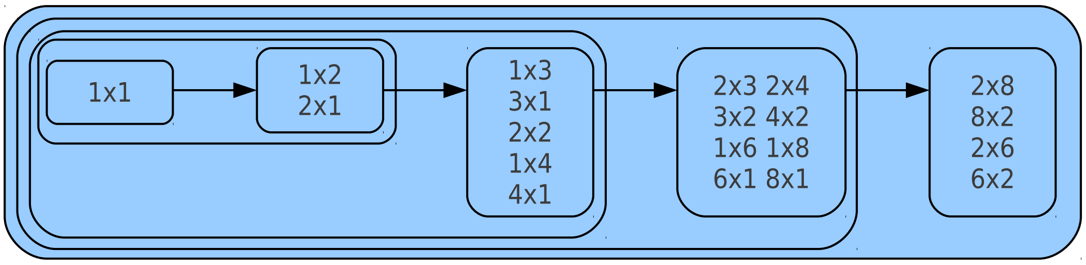
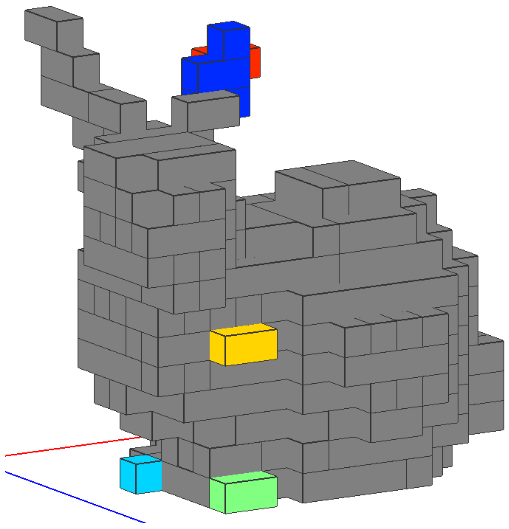


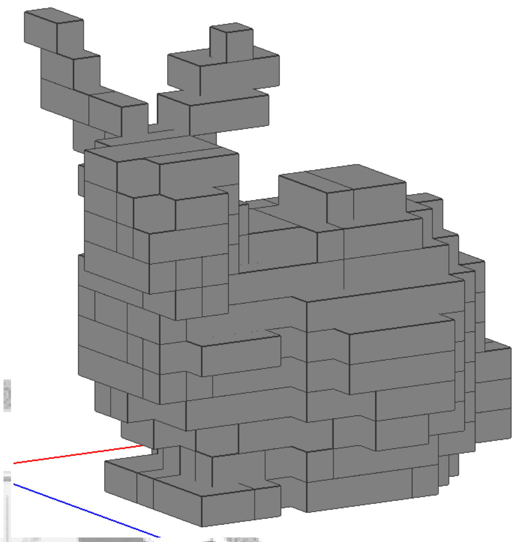
Figure 3.26: Merge closure, computed by the LEGO family set in Table 3.1.

newly merged brick. If the merge operation of a LEGO brick is failed, the brick is simply removed from the priority queue. This procedure is then terminated when the priority queue is empty. Fortunately, this procedure will also merge the smaller LEGO bricks left by previous optimization into larger LEGO bricks. In order to ensure all smaller LEGO bricks are merged into larger ones, we compute the number of merge operation from the set that containing only 1x1 LEGO bricks to the set that containing entire input LEGO family set, or the next set is the same as the current set. We call this “merge closure.” Figure 3.26 illustrates this computation. The result of our repairing attempt can be found in Figure 3.27.

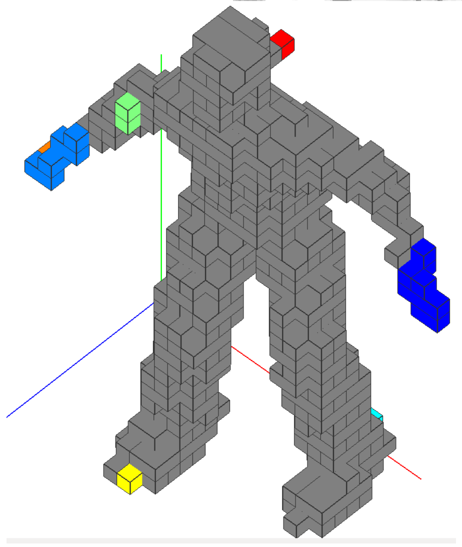
The fourth procedure is a heuristic greedy adjustment of LEGO brick quantity distribution. The idea is to split larger bricks of rare types into smaller bricks of rich types without losing connectivity. We continue splitting until the quantity limits of rare types



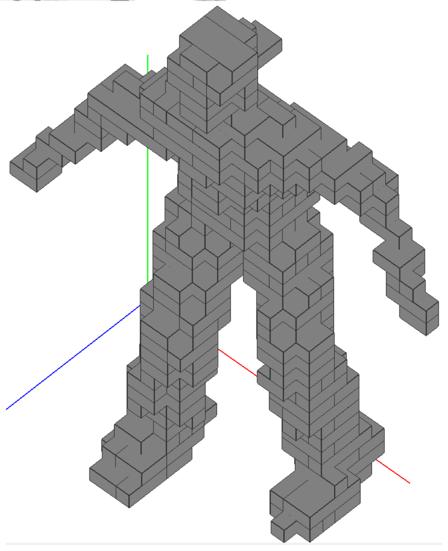
(a) Bunny, before repair
connected component = 6
number of brick = 191



(b) Bunny, after repair
entirely connected
number of brick = 165



(c) Child, before repair
connected component = 8
number of brick = 324



(d) Child, after repair
entirely connected
number of brick = 260

Figure 3.27: Post-optimization refinement - connectivity repair.

are satisified. We discovered that 1x3, 1x4, 1x6, 2x6 and 1x8 are rare types, 1x1, 1x2 and 2x2 are rich types. Therefore, we perform the following steps to achieve rare type quantity control.

- Split every 1x3 brick into 1x2 and 1x1.
- Merge every two 1x1 bricks into 1x2 bricks.
- Merge every two 1x2 bricks into 2x2 bricks (if mergeable).
- Split every 1x4 brick into two 1x2 bricks.
- Merge every two 1x2 bricks into 2x2 bricks (if mergeable).
- Split every 1x6 brick into three 1x2 bricks.
- Merge every two 1x2 bricks into 2x2 bricks (if mergeable).
- Split every 2x6 brick into three 2x2 bricks.
- Split every 1x8 brick into four 1x2 bricks.
- Merge every two 1x2 bricks into 2x2 bricks (if mergeable).

Except for the 2x6 split, there are always one or two merge after each split. This is because we want to prevent connectivity instability. However, this conservative sequences may also reduce the chance of successful quantity control.

Finally, to ease the actual construction, we begin to think if there are better building instruction order. We observed that, if we do not have the bottom plate, it is difficult for us to build LEGO structure layer-by-layer, which [5] did. We think a better building order should satisfy:

- Ensure always connected when adding a brick.
- We like to build LEGO structure bottom-up.

According to our observation, we implemented a bottom-up building order by simply a vertical priority breadth-first search of LEGO bricks using undirected connectivity defined in Def. 2.

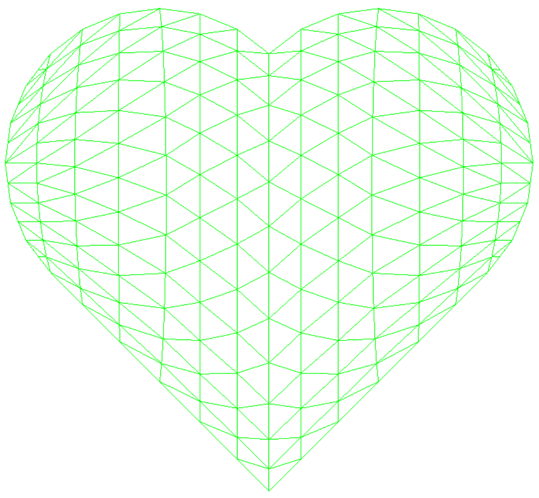
Chapter 4

Results and Evaluation

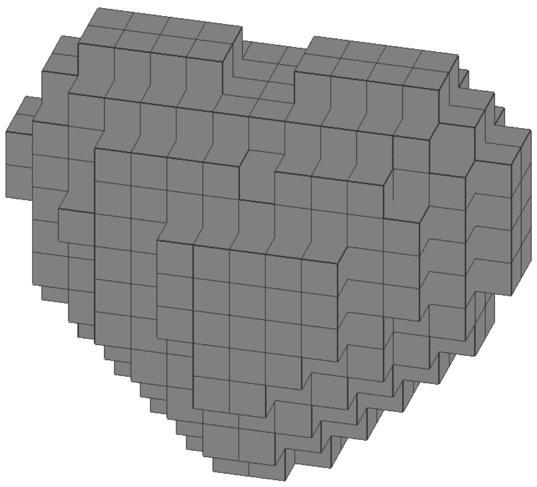
We have experimented and real constructed using the model listed in Table 4.1, each result will show four images and a table of statistics. The four images are wireframed 3D triangular mesh model, preprocessed voxelized model, computer generated legolized model, and actual constructed LEGO sculpture. The table of statistics for the given model will show the consumption of LEGO bricks using the format of Table 3.1, and optimized energy terms. The surface thickness of all voxelized model are 2 in all six sides, mentioned in the section 3.3. The optimization parameters are listed in the section 3.4. All results are entirely connected (i.e., the number of connected component is 1). Some results will have minor user intervention in order to cope with complicated disconnected situations. These interventing operation differences will also be listed in Table 4.1. Our legolized models can often completed within 30 minutes or one hour with the aid of graphical user interface. The builder often spends most of the time trying to find the exact LEGO brick instructed by the system.

Name of 3D Triangular Mesh Model	Applied User Intervention
Heart	Red surface and Constraint 5 rare types only for red
Standford Bunny (Simplified)	Red eye and Disable 5 rare types
Child	None

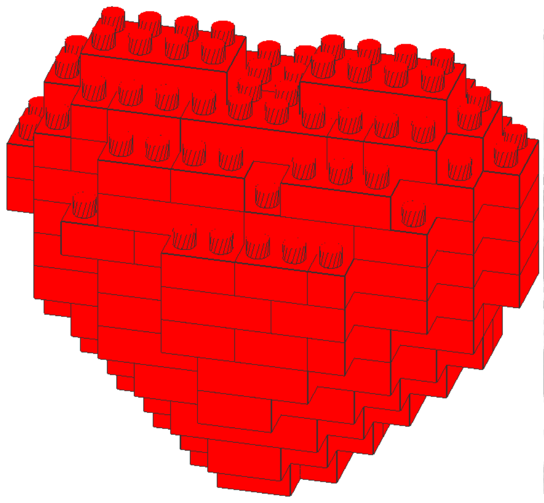
Table 4.1: Experimented 3D triangular mesh models.



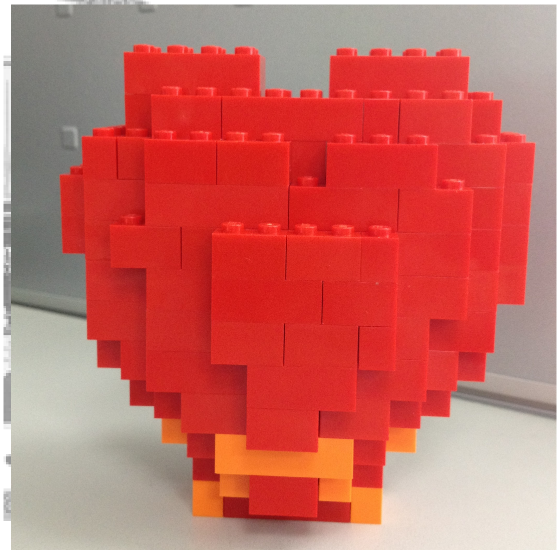
(a) Triangle count = 664



(b) Resolution = (x: 14, y: 13, z: 8)



(c) Total Energy = -3306

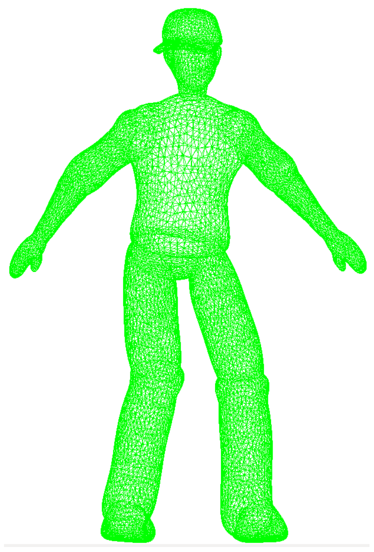


(d) Runtime = less than 1 second

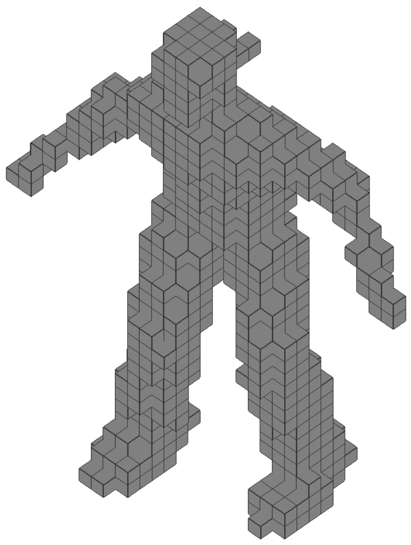
Figure 4.1: Experimented result - Heart.

Type	Quantity	Type	Quantity	Energy Terms	Cost
1x1x1	10			$E_{numBricks}$	+882
2x1x1	52	2x1x2	18	$E_{dirAltern}$	-244
3x1x1	8	3x1x2	12	$E_{sameVedge}$	0
4x1x1	6	4x1x2	14	$E_{numConnBricks}$	-1942
6x1x1	4	6x1x2	0	$E_{areaConnBricks}$	-1672
8x1x1	2	8x1x2	0	$E_{numDistinctColor}$	-330

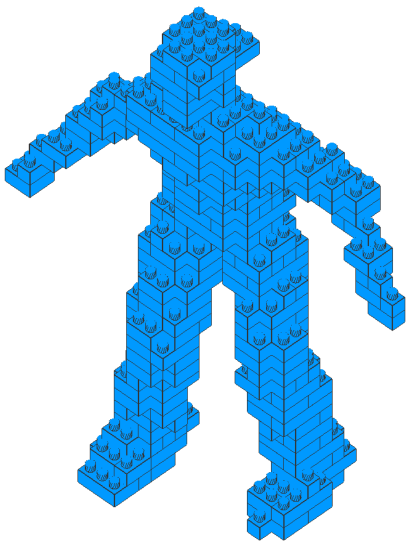
Table 4.2: Result Statistics - Heart.



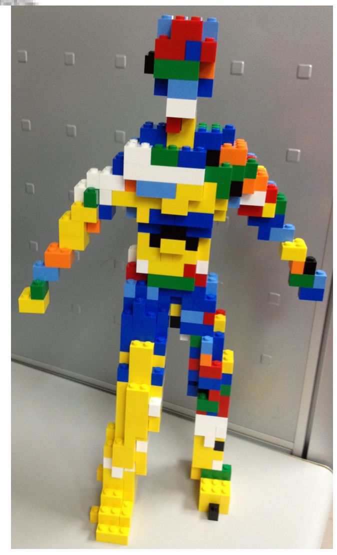
(a) Triangle count = 26668



(b) Resolution = (x: 25, y: 37, z: 12)



(c) Total Energy = -8961

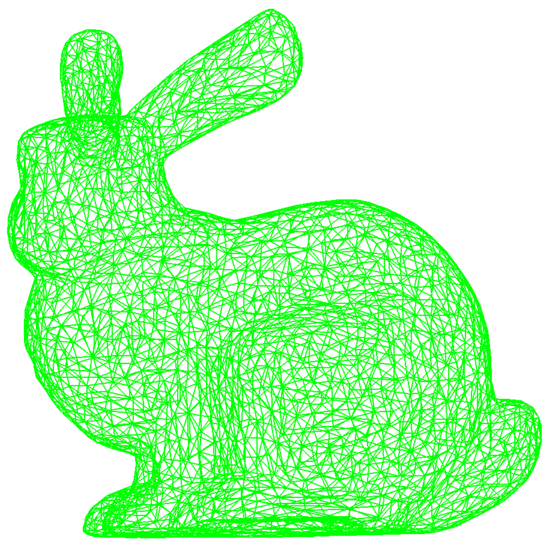


(d) Runtime = 5 seconds

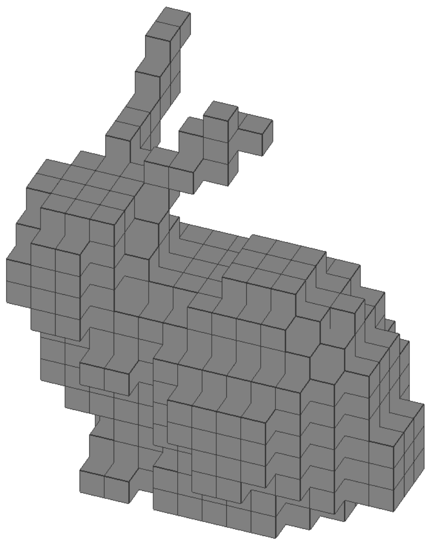
Figure 4.2: Experimented result - Child.

Type	Quantity	Type	Quantity	Energy Terms	Cost
1x1x1	46			$E_{numBricks}$	+1988
2x1x1	81	2x1x2	24	$E_{dirAltern}$	-1448
3x1x1	40	3x1x2	21	$E_{sameVedge}$	0
4x1x1	32	4x1x2	26	$E_{numConnBricks}$	-4627
6x1x1	9	6x1x2	2	$E_{areaConnBricks}$	-4184
8x1x1	3	8x1x2	0	$E_{numDistinctColor}$	-690

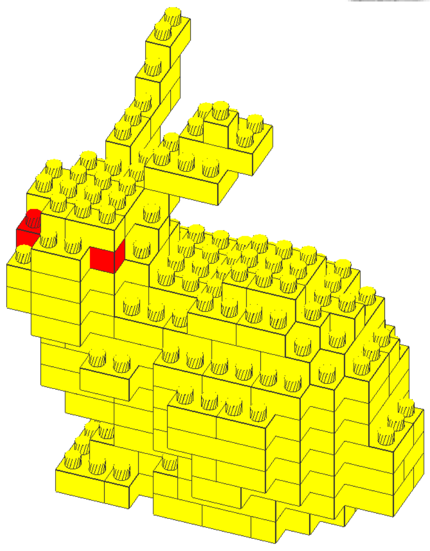
Table 4.3: Result Statistics - Child.



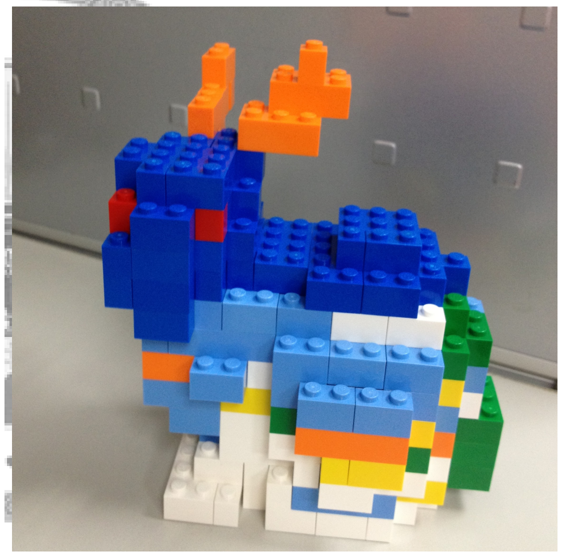
(a) Triangle count = 4968



(b) Resolution = (x: 15, y: 15, z: 12)



(c) Total Energy = -3904



(d) Runtime = 2 seconds

Figure 4.3: Experimented result - Bunny.

Type	Quantity	Type	Quantity	Energy Terms	Cost
1x1x1	42			$E_{numBricks}$	+1519
2x1x1	108	2x1x2	38	$E_{dirAltern}$	-284
3x1x1	0	3x1x2	17	$E_{sameVedge}$	0
4x1x1	0	4x1x2	12	$E_{numConnBricks}$	-2201
6x1x1	0	6x1x2	0	$E_{areaConnBricks}$	-2400
8x1x1	0	8x1x2	0	$E_{numDistinctColor}$	-538

Table 4.4: Result Statistics - Bunny.

There are still unsolvable cases appeared during our experiments, and our system can only show where LEGO bricks are disconnected due to the limitation of our methods. The limitations will be discussed in the next chapter. Figure 4.4 demonstrated this functionality.

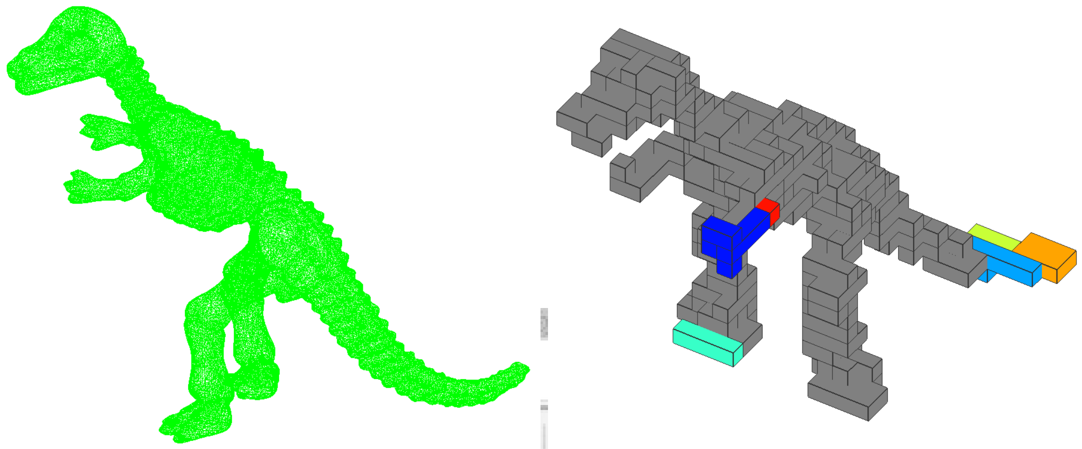

(a) (b) Connected component = 7

Figure 4.4: Failure detection - Dinosaur.

In addition to showing successful cases and failure cases, we did some simple evaluation of our methods according to the objectives of the LEGO construction problem. For connectivity objective, if we did not applying the post-optimization connectivity refinements, the overall chance of disconnecting is much higher, as shown in Table 4.5.

3D Model	# of disconnection without refinement	# of disconnection with refinement	# of connected component reduction with refinement
Bunny	20	14	20
Child	20	7	20
Heart	5	0	5
Dinosaur	20	20	18

Table 4.5: Evaluation - connectivity objective, test 20 runs.

For visual similarity objective, our informal user study shows that users favor our hybrid (thinner) voxelized result, it is more similar than box-triangle voxelization result. In addition, it brings fewer brick usage. Among 5 persons, only 1 person choose the box-triangle voxelized result once. The hybrid voxelized results are chosen 12 times.

For quantity constraint objective, we tested the extreme case that if we can obtain a fully connected LEGO model without using any of LEGO bricks from rare types (1x3, 1x4, 1x6, 1x8, and 2x8). We count that how many times does our method successfully decrease the usage of rare types larger than 10 bricks. Table 4.6 shows the results.

Rare types	Bunny	Child	Heart
1x3	5	4	5
1x4	5	3	5
1x6	5	2	5
1x8	3	5	5
2x6	5	5	3

Table 4.6: Evaluation - quantity constraint, test 5 runs.

For runtime limit objective, our method runs very quickly among all our experimented cases. We have also tested twice larger voxel resolution, the average runtime is around 30 to 60 seconds This is beacuse we firmly control the optimization maximum iteration count. We find that cellular automaton often converged within 100 iterations. And our method can even put to acceleration, since cellular automaton has part that can process in parallel. But this gain trades the possibility of finding an exact solution off. This is the limitation of our methods and will discussed in the following chapter.

Chapter 5

Discussion and Limitation

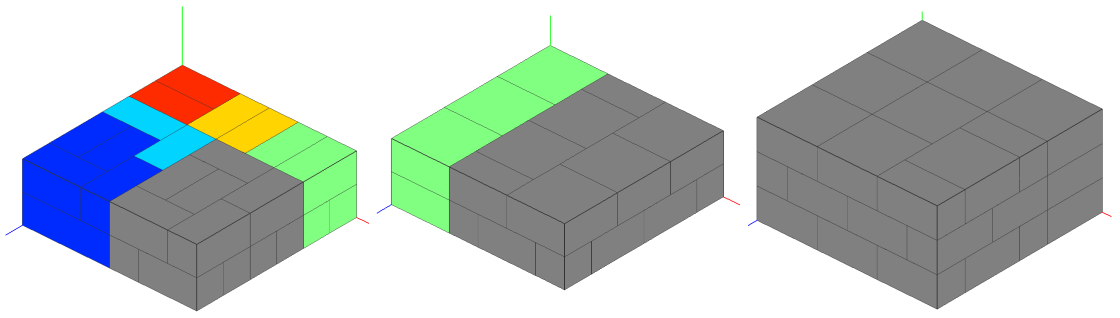
Although we can produce high quality results in nearly fully automated fashion, there are limitations within our system. First of all, our placement optimization algorithms, such as cellular automaton optimization and post-optimization connectivity refinement, have the following limitations due to the nature of heuristic design.

- Using only fixed amount of iterations will often lead to local optimal results.
- It mixes the element of randomness, every run of optimization may obtain different results providing the same input.
- No guarantee of finding the global optimal or a connectivity one solution.

These limitations will cause our “legolizer” program unable to deterministically answer whether it can solvable or not (and it may even left solvable parts unsolved). However, one can claim that it is the inherited limitation of heuristic algorithms for NP-hard problems, if we really want a true answer, we may contradict the timing constraint objective.

To further demonstrate these limitations, we made a pressure test on the cellular automaton algorithm. We limit the LEGO brick set that contains only 1x1 and 1x2 bricks. And the input voxelized volume is a 6x2x6 cube, which is shown in Figure 1.2 as an example of a problem instance. Our test results are demonstrated in Figure 5.1 and Table

5.1. Most of the time, the algorithm failed to produce fully connected result under very difficult cases.



(a) Table 5.1, row 2 (b) Table 5.1, row 4 (c) Table 5.1, row 8

Figure 5.1: The failure cases.

Cube	Brick Set	Iteration	# of Connected Component
6x2x6	1x1,1x2	98	9
6x2x6	1x1,1x2	198	6
6x2x6	1x1,1x2,2x2	98	1
6x2x6	1x1,1x2,2x2	198	2
6x3x6	1x1,1x2	98	4
6x3x6	1x1,1x2	198	8
6x3x6	1x1,1x2,2x2	98	1
6x3x6	1x1,1x2,2x2	198	1

Table 5.1: Pressure test of 6x2x6 and 6x3x6 cubes.

After the experiments on extreme cases, we begin to wonder that what may increase the difficulty of the LEGO construction problem. We discovered two following reasons that might force the connectivity optimization algorithm to search more placement solutions.

- The size factor of LEGO brick set. If the set contains only small bricks, then naturally we have to try more combination of placement to obtain a connected result.

- The height of voxelized model (i.e., number of layers). A disconnected component may get connected by using additional layers of voxels. As we have experimented, the 6x3x6 case is much easier to achieve one connected structure than 6x2x6 case.

The second limitation is that our hybrid voxelization is still unable to accurately capture the symmetry feature of input triangular 3D mesh model. The reason may lie in the

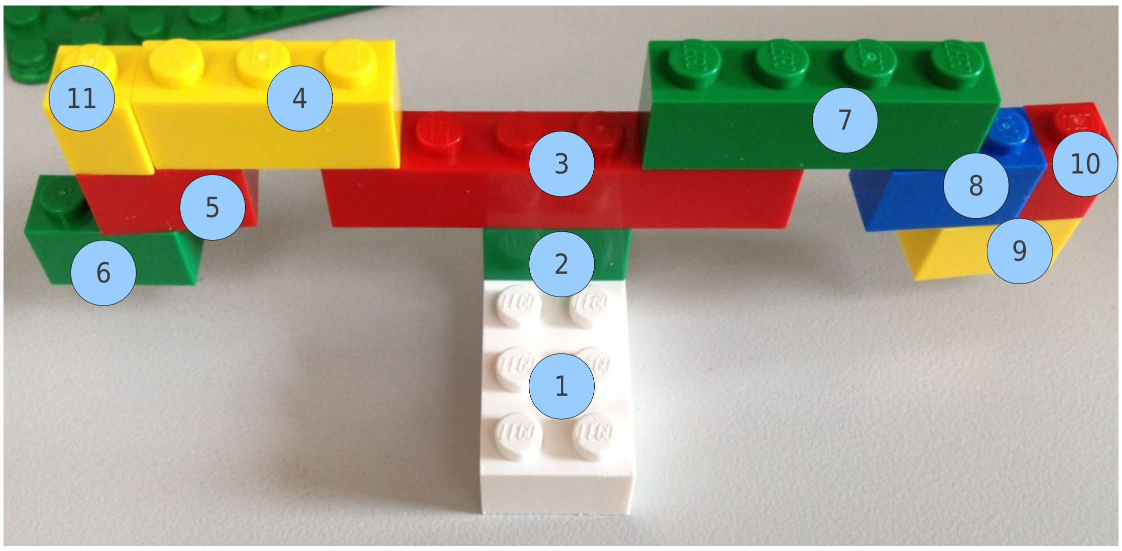


Figure 5.2: Building instruction order example.

ray-triangle voxelization, which is relatively less robust than box-triangle voxelization. Readers can reference the “Heart” results in Figure 4.3.

The third limitation is the building instruction order generation. Normally, if the 3D model changes its height smoothly, the generated building order is fairly easy for human to build. Figure 5.2 provides an example. However, when the 3D model consist of “tall legs”, shown in Figure 5.3, the generated building order will firstly touch down to the lowest, then will climb up. This result is rather strange for human to understand.

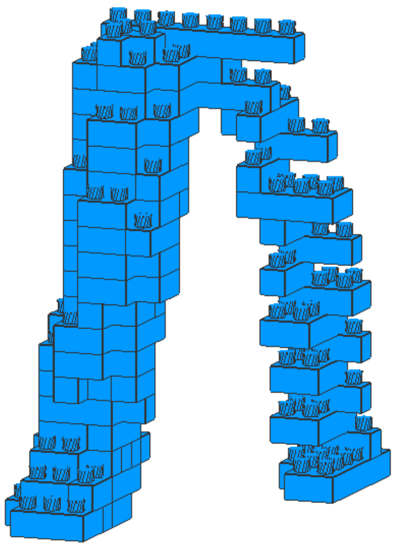


Figure 5.3: Building instruction order artifact.

Chapter 6

Conclusion and Future Work

We presented a constructible 3D model legolizer system that is nearly fully automated and runs within reasonable time. Because of our hybrid voxelization approach, our input 3D geometry can be nearly arbitrary and can use less LEGO brick while maintain visual similarity. We also increased the robustness of connectivity optimization algorithms with a noticeable amount by proposed post-processing algorithms. To further increase the chance of realization, we also provide a mean to adjust brick usage between rare types and rich types. Finally, our computed building orders are visually easy for LEGO players to finish their LEGO building project.

For future work, although our placement optimization considers color constraint, it is rarely tested because we do not automatically paint colors on LEGO sculpture by the texture of input 3D model. And our building order generation should consider part-by-part relationship since it is more nature for us to consider semantic components such as legs and torsos. Furthermore, there are always rooms for voxelization and optimization algorithms to improve.

Bibliography

[1] S. Xin, C.-F. Lai, C.-W. Fu, T.-T. Wong, Y. He, and D. Cohen-Or, “Making burr puzzles from 3D models,” in *ACM SIGGRAPH 2011 papers*, SIGGRAPH ’11, (New York, NY, USA), pp. 97:1–97:8, ACM, 2011.

[2] J. Vormann, “The dispatchwork project,” 2007. image from `http://www.janvormann.com/testbild/dispatchwork/`.

[3] N. Sawaya, “`http://brickartist.com/`,” 2012.

[4] “Model Mom Mary’s blog,” 2012. image from `http://news.legoland.com/post/Model-Mom-Marys-Five-Favorite-Models.aspx`.

[5] L. van Zijl and E. Smal, “Cellular Automata with Cell Clustering,” in *Proceedings of AUTOMATA2008 Workshop*, (Bristol, UK), pp. 425–440, June 2008.

[6] O. Timcenko, “LEGO: How to build with LEGO,” in *32nd European Study Group with Industry*, p. xix–xxi, 1998. `http://www2.mat.dtu.dk/ESGI/32/Report/ESGI32.ps`.

[7] S. Izadi, D. Kim, O. Hilliges, D. Molyneaux, R. Newcombe, P. Kohli, J. Shotton, S. Hodges, D. Freeman, A. Davison, and A. Fitzgibbon, “KinectFusion: real-time 3D reconstruction and interaction using a moving depth camera,” in *Proceedings of the 24th annual ACM symposium on User interface software and technology*, UIST ’11, (New York, NY, USA), pp. 559–568, ACM, 2011.

[8] C. Holz and A. Wilson, “Data miming: inferring spatial object descriptions from human gesture,” in *Proceedings of the 2011 annual conference on Human factors in computing systems*, CHI ’11, (New York, NY, USA), pp. 811–820, ACM, 2011.

[9] M. Fisher and P. Hanrahan, “Context-based search for 3D models,” in *ACM SIGGRAPH Asia 2010 papers*, SIGGRAPH ASIA ’10, (New York, NY, USA), pp. 182:1–182:10, ACM, 2010.

[10] P. Petrovic, “Solving the LEGO brick layout problem using evolutionary algorithms,” 2001. `http://www.nik.no/2001/08-petrovic.pdf`.

[11] R. A. H. Gower, A. E. Heydtmann, and H. G. Petersen, “LEGO: Automated Model Construction,” in *32nd European Study Group with Industry*, p. 81–94, 32nd European Study Group with Industry, 1998. `http://www2.mat.dtu.dk/ESGI/32/Report/lego.ps`.

[12] J. V. Neumann, *Theory of Self-Reproducing Automata*. Champaign, IL, USA: University of Illinois Press, 1966.

[13] S. Na, *Optimization for Layout Problem*. Syddansk Universitet. Mærsk Mc-Kinney Møller Instituttet for Produktionsteknologi, 2002.

[14] B. Lambrecht, “Voxelization of boundary representations using oriented LEGO plates,” 2006. `http://code.google.com/p/lsculpt/`, last accessed: July, 2012.

[15] L. Silva, V. Pamplona, and J. Comba, “Legolizer: A Real-Time System for Modeling and Rendering LEGO Representations of Boundary Models,” in *Computer Graphics and Image Processing (SIBGRAPI), 2009 XXII Brazilian Symposium on*, pp. 17 –23, oct. 2009.

[16] “LEGO Digital Designer.” By LEGO Group, `http://ldd.lego.com`.

[17] “Build with Chrome.” By LEGO Group and Google, `http://www.buildwithchrome.com/static/map`.

[18] J. Jessiman, “LDraw, LEGO CAD software package.” `http://beta.ldraw.org/`.

[19] L. C. Hassing, “L3P: an LDraw to POV-Ray conversion utility.” `http://www.hassings.dk/l3/l3p.html`.

[20] “POV-Ray - The Persistence of Vision Raytracer.” `www.povray.org`.

[21] K. Clague, M. Agullo, and L. Hassing, *LEGO Software Power Tools: With LDraw, MLCad, and LPub*. Syngress, 2003.

[22] T. Courtney, S. Bliss, and A. Herrera, *Virtual Lego: The Official Ldraw.Org Guide to Ldraw Tools for Windows*. No Starch Press Series, No Starch Press, 2003.

[23] A. Glassner, “Recreational computer graphics,” in *ACM SIGGRAPH 2006 Courses*, SIGGRAPH ’06, (New York, NY, USA), ACM, 2006.

[24] K.-Y. Lo, C.-W. Fu, and H. Li, “3D polyomino puzzle,” in *ACM SIGGRAPH Asia 2009 papers*, SIGGRAPH Asia ’09, (New York, NY, USA), pp. 157:1–157:8, ACM, 2009.

[25] T. Shigeo, H.-Y. Wu, S. H. Saw, C.-C. Lin, and H.-C. Yen, “Optimized Topological Surgery for Unfolding 3D Meshes,” in *Computer Graphics Forum (Pacific Graphics 2011)*, vol. 30, 2011.

[26] K. Hildebrand, B. Bickel, and M. Alexa, “crdbrd: Shape Fabrication by Sliding Planar Slices,” in *Computer Graphics Forum (Eurographics 2012)*, vol. 31, 2012.

[27] K. Xu, H. Zheng, H. Zhang, D. Cohen-Or, L. Liu, and Y. Xiong, “Photo-inspired model-driven 3D object modeling,” in *ACM SIGGRAPH 2011 papers*, SIGGRAPH ’11, (New York, NY, USA), pp. 80:1–80:10, ACM, 2011.

[28] L.-T. Shen, S.-J. Luo, C.-K. Huang, and B.-Y. Chen, “SD Models: Super-Deformed Character Models,” *Computer Graphics Forum*, 2012. (Pacific Graphics 2012 Conference Proceedings).

[29] F. Nooruddin and G. Turk, “Simplification and repair of polygonal models using volumetric techniques,” vol. 9, pp. 191 – 205, april-june 2003.

[30] T. Akenine-Möller, “Fast 3D triangle-box overlap testing,” *J. Graph. Tools*, vol. 6, pp. 29–33, Jan. 2002.