Automatic generation of LEGO from the polygonal data

Sumiaki Ono
ARTDINKS
Tokyo, Japan

Youngha Chang
Knowledge Engineering
Tokyo City University
Tokyo, Japan

Andre Alexis
SONY CSL
Tokyo, Japan

Masayuki Nakajima
School of Game design, Technology and Learning Process
Gotland University
masayuki.nakaima@hgo.se

Figure 1 The algorithm overview. (left) inputted 3D polygon model, (middle left) voxel representation, (middle right) brick representation and (right) assembly manual.

Abstract—In this work, we propose a method that converts a 3D polygonal model into a corresponding LEGO brick assembly. For this, we first convert the polygonal model into the voxel model, and then convert it to the brick representation. The difficulty lies in the connection between bricks should be guaranteed. To achieve this, we define replacement priority, and the conversion from voxel to brick representation is done according to this priority. We show some experimental results, which show that our method can keep the connection, and achieve a robust and optimized method for assembling LEGO building bricks.

Keywords—assembly; lego; model conversion

I. INTRODUCTION

The block toy represented by LEGO can assemble a variety of shapes by combining blocks of simple form. While it is an attractive toy and stimulates users’ creativity, it is difficult for beginners to make a complicated 3D work without any references. Therefore, it sells as a kit including a building manual. In this paper, we propose a technique for generating 3D block artwork automatically from a 3D polygon model.

There has been previous research conducted investigating the subdivision of 3D models into corresponding parts and the representation of the 3D geometry by using parts. Mitani et al. proposed a method to generate unfolded paper craft patterns from 3D mesh data [1]. They subdivide the 3D model segment each part into zonal regions, and grouping triangles having similar topological distances from the part boundary. Finally, they can generate triangle strips by simplifying the mesh while retaining the borders of the zonal regions and additional cut-lines. The pattern is then created by unfolding the set of strips. This research is similar to our goal in terms of assembling subparts to create 3D model, but the paper craft do not need to consider the connections between parts, and no limitation on the shape of the subparts.

Igarashi et al. proposed a practical application to generate close-fitting customized covers for a given 3D model [2]. They first cluster vertices, and then generate multiple convex hulls. Next, it outputs a cover geometry to set union operation of these hulls, and the resulting intersection curves are set as seam lines. This method is also similar to our goal, however, it
does not need to consider the connectivity and has no shape limitation. Xin et al. propose a method to design and model burr puzzles from 3D models [3]. When the user interactively embed a network of knots into the 3D shape, their system optimizes and arranges the orientation of each knot, and modifies pieces of adjacent knots with an appropriate connection type. Then, the 3D model is partitioned by splitting the solid while respecting the assembly motion of embedded pieces. Their method also has a freedom in the shape of pieces, so we cannot directly apply their method to LEGO buildings. Lo et al. propose a method to realize a 3D polyomino puzzle, which fills the surface of 3D models by using polyomino-like shape pieces [4]. It is the most similar one to our research, but their goal does not need to consider the connectivity between pieces, since they can add the blank and tabs between the pieces freely. Silva et al. proposed a method to convert the 3D model into LEGO block representation [5]. They first voxelize the 3D model and polygonize with color coherence to produce LEGO bricks. Their method is implemented on the GPU so that it can achieve real time process. However, their method does not correspond to multiple brick types and also, it cannot guarantee the connectivity between bricks.

II. ALGORITHM OVERVIEW

In Figure 2, we show an illustration of typical LEGO brick. It has potches on the topside that are used to connect other bricks. The LEGO brick is named according to the number of potch. For example, the LEGO brick shown in Figure 2 is called as “2x4 brick”, that means that the brick has 2 x 4 potches on the topside. The 1x1 brick is called as a unit brick. In this work, we assume that the 3D model is constructed with 11 types of LEGO bricks, as shown in Figure 3. These LEGO bricks are 1x1, 1x2, 1x3, 1x4, 1x6, 1x8, 2x2, 2x3, 2x4, 2x6, and 2x8 bricks. All of these are contained in the LEGO basic sets, and all have the same brick height. The potches of the LEGO bricks are placed on the top of the brick. That means, the bricks can be connected each other only toward the vertical direction.

Our algorithm overview is as follows: The input of this work is the 3D polygonal model and the desired level of detail. The system then converts the 3D polygonal model to the voxel representation according to the level of detail, and then converts it to the brick representation. Finally, it generates an assembly manual for the input model.

III. Polygon Voxelization

We apply the conventional scan line method to convert the polygonal data to voxel representation. We then delete the internal voxels. We define a voxel as internal if it has 26 adjacent voxels. After deleting the internal voxels, there might be a case that some of the voxels do not have its 6-neighborhoods (see Figure 5) as shown in Figure 4 left. In this case, the red voxel cannot connect with other, and fall off the upper one. To avoid this, we insert voxels inside of the model (Figure 4 right, black voxels).

IV. Voxel to Block Representation

Now we convert the voxel representation to the block representation. Each voxel can be regarded as the unit brick. Therefore, the initial brick representation is already acquired. Now, we will merge the unit brick, and replace it with bigger bricks. All the blocks have the same height. Therefore, we will slice the 3D model into block height thick slices, and then replace each voxel (= smallest block) with larger blocks slice-by-slice. In this work, we will denote these slices as layer. The most important thing during the replacement is that the connection between bricks should be guaranteed. See Figure 5. Please observe the red voxel in the layer i. If it is not connected with one of 6-neighbor voxels (in green), it will then fall off. Therefore, one of 6-neighborhood connection becomes important in this process.

Figure 2 The shape of LEGO brick

Figure 3 Brick types used in this paper

Figure 4 Insert some of the internal voxels to ensure the connectivity between the voxels

Figure 5 Voxel neighborhoods
For this, we define a tree structure that shows which voxel should be connected. We refer to this structure as a tree structure as legograph, and in the legograph, we save three types of “link” between voxels: priority1 link, priority2 link, and endnote link.

The example of priority1 class link is shown in Figure 6 left (red line). Priority1 link illustrates cases in which there exists one voxel that sticks out sideways, and it should be merged with its neighbor (otherwise, it will be fall off).

Figure 6 Three types of links

In Figure 6 middle, we show an example of priority2 link (green line). In this example there is a voxel that sticks out sideways in the second layer, but it has neighborhood voxel in third layer. Therefore, even if it is not merged with its neighbor in the second layer, it can still connect with other voxels if its upper side voxel is connected to other parts.

An example of the endnote link is shown in Figure 6 right (blue line). From priority2 link to endnote link, connection should be achieved.

An example of legograph is shown in Figure 7.

(a) Voxel arrangement

Figure 7 Voxel representation and its legograph

With this legograph, we will convert the voxel representation to the block representation.

The basic process is that, choose a voxel, and place there one of a brick type shown in Figure 3. This replacement is done layer-by-layer, from bottom to top.

The block replacement in a layer \( y = y_i \) is done with a greedy method. For each position \((x, y_i, z)\), we try to put one of brick type \( t \), and to calculate the score. The position and brick type \( t \) with the highest score will then be chosen and be replaced.

The basic strategy for the scoring is as follows:

1. If a brick covers more under-layer bricks, the score becomes higher. This is because the assembly becomes stable if there are more connections with other bricks.
2. Bigger bricks get higher scores. This is for time efficiency and for saving the amount of bricks.
3. Link coverage will get higher score.

Based on this strategy, the score \( S \) can be defined as Equation 1.

\[
S(x, y_i, z, t) = w_1 N_u + w_2 N_s + w_3 N_a + w_4 N_{p1} + w_5 N_{p2} + w_6 N_{pe}
\] (1)

Where, \((x, y_i, z, t)\) means currently checked block is type \( t \) and its left-upper corner is placed at \((x, y_i, z)\). \( w_1, w_6 \) are weights, \( N_u \) is the number of lower layer blocks connected with currently checked block. \( N_t \) is the size of \( t \). \( N_a \) is the number of voxels connected with \((x, y_i, z, t)\) in \( y = y_i-1 \) layer. \( N_{p1}, N_{p2} \), and \( N_{pe} \) represent the number of priority1 links, priority2 links, and endnote links respectively, covered by \((x, y_i, z, t)\).

Finally, we adjust the legograph. Let us assume that there is a case that the priority 2 link is not connected in current layer.

Then, the link is carried over and upward as shown in Figure 8 Carry over of priority3 link.

Figure 8 Carry over of priority3 link

If the carry over occurs to the endnote link as shown in Figure 9 Priority change, then it will be changed to a priority1 link.
V. Division of Voxel model

The division process is done layer-by-layer, from bottom-to-top.

First, we attach labels for voxels in the current layer. We regard a layer as a 2D grid, and a grid cell is set to 1 if there exists a voxel, or 0 otherwise. Next, our system scans the grid from the top-left until it finds a cell labeled 1. Then we repeatedly check and tag the same label to its 8-neighboring grid cells if it is 1, like the region growing method. An example of labeling is shown in Figure 10 Labeling a layer. The set of voxels having the same label will be denoted as a region. In this layer, we have three regions.

Next, we update the label of each region by considering the connection with regions of its lower layer. See Figure 11. If the 1-to-1 connection occurs, then the label will be updated (re-label to the same label) as shown in Figure 11(a). Or, if 1-to-multiple or multiple-to-1 connection occurs, then the label will not change.

VI. RESULTS

A result is shown in Figure 13 and its assembly manual is shown in Figure 15. The input is the 3D model of the Stanford bunny. The colors of the blocks represent the division described in Section V. This result shows that the blocks are well connected with each other. Moreover, we can save the time and the number of blocks because the inner area of the 3D model is hollow. Furthermore, we realize parts-by-part building procedure.
VII. EVALUATION

We conducted user evaluation to verify the effectiveness of our system. We asked users to build a LEGO model (1) with voxel representation, (2) with assembly manual without part division, and (3) with assembly manual with part division. We employed the voxel representation in (1) instead of the smooth 3D rendering, because we want to direct the level of detail. The comparison is done with three points:

a) The amount of time to build models
b) The visual quality of assembled artwork
c) Hearing from the participants

All the users were beginners for building bricks. Table 1 shows the amount of time required. The time cost with voxel representation was the highest. With and without division into model parts does not affect the time cost.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>User 1</td>
<td>26:25</td>
<td>11:11</td>
<td>8:16</td>
</tr>
<tr>
<td>User 2</td>
<td>41:11</td>
<td>19:38</td>
<td>20:39</td>
</tr>
<tr>
<td>User 3</td>
<td>49:24</td>
<td>14:58</td>
<td>14:36</td>
</tr>
<tr>
<td>User 4</td>
<td>14:00</td>
<td>11:33</td>
<td>8:29</td>
</tr>
</tbody>
</table>

Next, we compare the quality of the assembled models. Figure 14 shows the assembled result of Torus without (left) and with (right) assembly manual. The left model cannot capture the smooth curvature of Torus, while the right one can represent the curve correctly.

Finally, we show some comments from the interviews with the users.

For the voxel reference, they commented:
- Cannot figure out how to achieve the smooth curvature
- Cannot ensure the connectivity between the blocks

For the assembly manual without part division,
- Can understand how to represent the smooth curvature
- It was a great chance to learn how to make the curved objects

For the assembly manual with the part division,
- It was much easier to construct a model not by bottom-up, but with small division of the model
- Can imagine where I am now constructing

VIII. CONCLUSION

In this paper, we propose a technique to generate the block toy building procedure automatically from a 3D polygonal model. However, there are still cases that the connection is not guaranteed. In the future, we would like to consider the improvement of the connections between bricks, and also would like to improve calculation time.

ACKNOWLEDGEMENT

This work is partially supported by Hayao Nakayama Foundation for Science & Technology and Culture, Japan.

Figure 15 Assembly manual for the bunny model shown in Figure 13