ISSN 2464-4617 (print)
ISSN 2464-4625 (online)
Computer Science Research Notes - CSRN
http://www.wscg.eu
WSCG 2025 Proceedings

Automated Conversion of 3D Mesh Car Models into LEGO Brick Sets Using Voxel-Based Optimization

Jan Ljubič, and Ciril Bohak
University of Ljubljana, Faculty of Computer and Information Science
Večna pot 113
1000, Ljubljana, Slovenia
jl6426@student.uni-lj.si, ciril.bohak@fri.uni-lj.si

Abstract

The transformation of 3D mesh car models into LEGO-compatible designs requires optimizing structural integrity, brick placement, and computational efficiency. This paper presents an algorithm specifically designed to convert polygonal car meshes into structurally sound LEGO representations. The approach involves voxelization, heuristic-based component mapping, and optimization techniques to ensure connectivity and maintain key design features. Using Blender for pre-processing and voxel-based methods for brick placement, the algorithm prioritizes larger LEGO elements to enhance stability and reduce part count. The proposed method is tested on various car models, demonstrating its ability to generate LEGO-compatible structures while preserving essential visual and functional details. The findings highlight potential applications in automated LEGO set design.

Keywords

3D Mesh Processing; voxelization; procedural modeling; 3D model conversion; structural optimization.

1 INTRODUCTION

The transformation of digital 3D models into physical representations has gained significant attention in various domains, ranging from rapid prototyping to digital fabrication. Among these, converting 3D mesh models into LEGO brick-based structures presents unique challenges, particularly in terms of structural integrity, brick placement optimization, and aesthetic fidelity. LEGO models must not only visually resemble their digital counterparts but also maintain physical stability and logical connectivity when constructed.

Manually designing LEGO-based models is a time-consuming and intricate process, requiring extensive experience in both digital modeling and LEGO-compatible structural design. Traditional approaches involve using dedicated LEGO modeling software or manually assembling bricks in real-world prototyping. However, automating this process can significantly enhance efficiency, particularly in applications where rapid design iteration and structural feasibility are essential.

This paper focuses on the automated conversion of 3D mesh car models into LEGO-compatible representations using a voxel-based optimization approach. The proposed method takes a polygonal car model as input, voxelizes it to create a discrete representation, and maps the resulting voxel grid to an optimal LEGO brick arrangement. The multi-stage process prioritizes structural connectivity, optimal part selection, and part minimization while preserving key design elements such as wheels, windows, and body shape.

The primary motivation for this work is to support the development of novel LEGO car sets which is particularly challenging due to the need to balance realism, stability, and part efficiency. Current manual approaches require extensive human effort, making them inefficient for large-scale or iterative design workflows. The main contributions of this paper are:

- **Voxelization-Aware LEGO Model Generation** – A voxelization-based approach that preserves the proportions and structure of 3D car models by correcting for the cuboid nature of LEGO bricks.
- **Component-Aware Brick Placement** – A semantic segmentation system that maps

car components (e.g., wheels, windows, cabin, lights) to optimized LEGO brick selections using convolution- and correlation-based placement strategies.

- **Graph-Based Connectivity Optimization of LEGO models** – A depth-first search algorithm that detects and resolves disconnected LEGO subgraphs by strategically placing large bricks to ensure model connectivity and stability.

The rest of the paper is structured as follows. We first present related work in Section 2. Next, we present our approach in Section 3. We present results in Section 4, and discuss their quality in Section 5. Finally, we present the conclusions in Section 6.

2 RELATED WORK

The conversion of 3D models into LEGO-compatible representations has been an active research area, particularly in the domains of voxelization, structural optimization, and procedural generation of LEGO models. Prior studies have proposed various methodologies, ranging from genetic algorithms to deep learning-based approaches, to improve the accuracy and efficiency of LEGO model generation. This section reviews key contributions to this field, focusing on voxelization techniques, optimization strategies, and automated LEGO model reconstruction.

2.1 Voxelization and LEGO Model Representation

Voxelization is a crucial preprocessing step in the transformation of 3D mesh models into LEGO-compatible structures. Zhao *et al.* [2] proposed a real-time voxelization algorithm optimized for complex polygonal models, enabling rapid discretization of 3D meshes into a structured voxel grid suitable for further processing. Huang *et al.* [4] developed an accurate method for voxelizing polygon meshes that ensures topological consistency and geometric fidelity, laying the groundwork for subsequent LEGO model generation techniques.

Voxelization techniques tailored for LEGO reconstruction have been explored in several works. Min *et al.* [9] introduced a silhouette-fitted voxelization approach, which refines voxelization by preserving the outer shape of the model while adapting the internal structure for optimal LEGO representation. This method enhances visual fidelity while ensuring model stability. Similarly, Gower *et al.* [3] proposed an automated model construction approach that segments voxelized 3D objects into LEGO-compatible components, facilitating efficient brick placement strategies.

2.2 Optimization Strategies for LEGO Model Generation

Once a 3D mesh is voxelized, selecting an optimal brick layout is essential to ensure both stability and efficiency. Lee *et al.* [5] developed a genetic algorithm-based approach to determine the optimal arrangement of LEGO bricks for voxelized 3D models, significantly improving structural integrity and part efficiency. Their method demonstrated how evolutionary computation can enhance the automation of LEGO model design.

Beyond heuristic-based optimization, deep learning has also been explored. Thompson *et al.* [12] introduced a deep generative model trained on LEGO graphs to predict optimal brick placement, enabling automatic generation of stable LEGO structures. While such methods show promise, they require extensive training data and computational resources.

Alternative heuristic approaches have also been investigated. Cai *et al.* [1] proposed an intelligent system for constructing complex LEGO models using rule-based algorithms and heuristic search techniques, improving the feasibility of large-scale LEGO reconstructions. Ono *et al.* [10] introduced an algorithm for automatically generating LEGO assembly instructions from polygonal 3D models, streamlining the transition from digital designs to physical builds.

2.3 Automated LEGO Model Reconstruction

Several studies have focused on automating the reconstruction of LEGO models from 3D meshes and images. Luo *et al.* [8] presented "Legolization", a framework for transforming 3D models into structurally valid LEGO structures by prioritizing part connectivity and load distribution. Their approach ensures that generated models are not only visually accurate but also physically stable.

Image-based LEGO model generation has also been explored. Lennon *et al.* [6] proposed Image2LEGO, a system for generating customized LEGO sets from input images, demonstrating how computer vision techniques can facilitate LEGO design from real-world objects. Zhou *et al.* [14] extended this concept by developing computational techniques for generating LEGO sketch art, broadening the applicability of LEGO-based design automation.

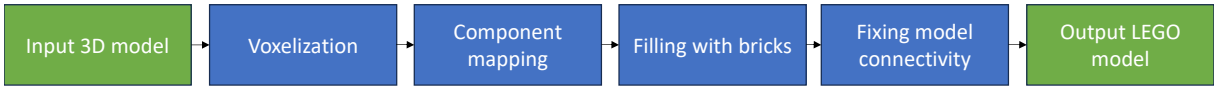

Figure 1: The overall system structure.

In addition to standard model reconstruction, Testuz *et al.* [11] proposed an algorithm for constructing large-scale LEGO sculptures, optimizing the use of brick sizes while ensuring mechanical stability. Zhou *et al.* [13] extended this work by introducing an automated system for generating realistic LEGO architectural sculptures, incorporating shape-preserving constraints to enhance fidelity.

The body of research on voxelization, optimization, and automated LEGO model generation provides a strong foundation for further advancements in the field. While previous approaches have addressed general-purpose LEGO model construction, there remains a gap in domain-specific applications, such as the conversion of 3D car models into LEGO-compatible designs. This paper builds upon these prior works by developing a voxel-based optimization framework specifically tailored for car models, ensuring structural integrity while preserving key vehicle features.

3 METHODS

To address the given problem, we developed a system that automatically converts a 3D mesh model of a car into a LEGO-compatible representation. Due to the complexity of the task, the system is structured into multiple phases, enabling modular processing where each phase is responsible for a specific sub-task. The overall system architecture is illustrated in Figure 1.

3.1 Voxelization

The voxelization process is carried out using the Binvox program, which accepts a 3D model in various formats, voxelizes it based on user-defined parameters, and outputs a voxelized 3D grid in the `.binvox` format. This output can be read and converted into a NumPy data array using the `binvox-rw` module in Python. The voxelized grid is a 3D array of integers, where 0 represents empty space, and 1 indicates occupied voxels within the model. The output of this stage can be seen in Figure 2.

To evaluate the voxelization output, we initially replace each occupied voxel with the smallest available LEGO brick, a $1 \times 1 \times 1$ unit brick. However, a direct one-to-one mapping results in gaps between horizontal layers due to the cuboid nature of LEGO bricks, which do not perfectly align with

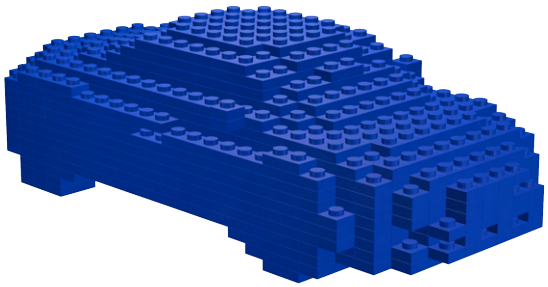

Figure 2: The output of the voxelization stage.

unit voxels. This misalignment arises because standard LEGO bricks are 40 % of the height of a unit cube. To correct this, we introduce a scaling factor of 2.5 along the vertical axis before voxelization, ensuring that the final LEGO model accurately preserves the original proportions of the input 3D car model.

3.2 Component Mapping

Cars typically consist of distinct functional components, such as (1) wheels, (2) the cabin, and (3) lights. The component mapping phase is designed to classify each voxel within the model based on its corresponding car component. This classification is essential for determining the appropriate LEGO bricks, their placement, and their respective properties, such as color and material. The component mapping stage is divided into two sub-phases: (1) Preprocessing the 3D model to define all necessary components and (2) Mapping each voxel to its corresponding car component. The components can be built using different LEGO bricks. Some of the bricks used in our approach can be seen in Figure 3.

3.2.1 Preprocessing the 3D Model

Initially, input 3D models are treated as single objects, representing the car as a whole. To facilitate component-based brick selection, users must manually define separate model components in Blender. This is achieved by creating new geometric objects that represent key elements, such as wheels, the cabin, and lights. Due to the discrete nature of the voxelization grid, a certain degree of approximation is required. Instead of manually segmenting the existing model, users can create separate bounding geometries that define the target components.

Once all components are defined, they are grouped under a single object within the Blender scene,

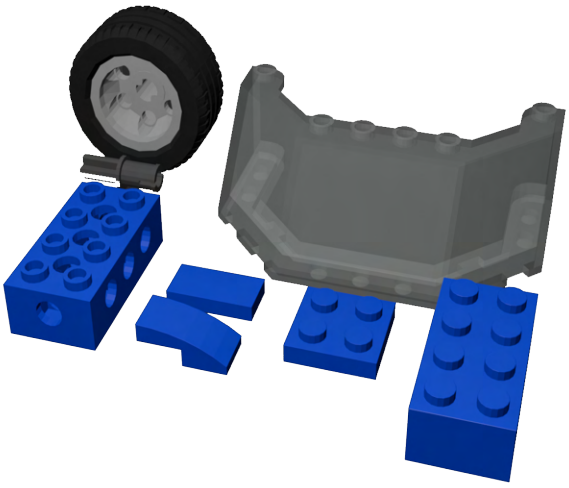

Figure 3: The output of the component mapping stage.

where each component retains its individual identity. This grouping mechanism simplifies hierarchical mapping, as each component can be referenced by its assigned name in subsequent processing steps.

3.2.2 Component Mapping via Ray Casting

Before proceeding with voxel-based mapping, the 3D model must be aligned and scaled to match the voxelization grid. This manual step in Blender allows users to adjust the desired scale for optimal brick resolution.

To classify each voxel, we introduce a component tracking grid that mirrors the dimensions of the voxelization grid. However, instead of binary occupancy values (0 or 1), this grid stores integer labels, each corresponding to a specific car component.

To assign voxel classifications, we employ a ray-casting technique, where each voxel is analyzed to determine its component affiliation. This process involves iterating through the voxel grid, identifying intersecting component volumes, and marking the voxel grid accordingly. The resulting component-aware voxel grid serves as the basis for structured LEGO brick placement, ensuring that each car component is represented with the appropriate LEGO elements. The output of this stage can be seen in Figure 4.

3.3 Filling with Bricks

The process of filling the model with LEGO bricks is the most computationally intensive stage of the entire workflow. It consists of several sub-stages executed in a specific order to ensure the correct and consistent placement of bricks. Bricks are placed in horizontal layers, starting from the bottom and

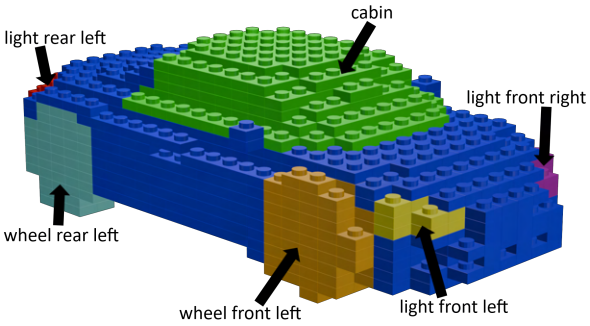

Figure 4: The output of the component mapping stage.

progressing to the top. The thickness of each layer depends on the height of the currently selected bricks, which can be classified as either thick or thin bricks.

Initially, only bricks corresponding to directly mapped voxels are placed in the model. However, to incorporate bricks of different dimensions, a method is required to determine all the available positions for each type of brick, which may span multiple voxels. We implement two key methods for this purpose:

- *Convolution*, used for bricks whose kernels are entirely filled, and
- *Correlation*, used for bricks whose kernels are only partially filled.

Convolution is applied to standard thick, thin, and smooth bricks, while correlation is used exclusively for sloped bricks. These methods ensure that each brick type can be correctly placed, provided that sufficient space is available in the voxel grid.

To handle the different orientations that bricks can take within the model, we define cardinal directions for brick placement. Each brick can have either no direction, two directions, or four directions, depending on the number of filled voxels within its kernel. For example:

- A $2 \times 2 \times 3$ brick has no directional constraints.
- A $4 \times 2 \times 3$ brick has two orientations (north-south and east-west).
- A $3 \times 1 \times 3$ sloped brick has four orientations (north, south, east, and west).

The number of available directions for a brick determines the number of possible orientations that must be considered when searching for valid placement positions.

Since the model consists of bricks of varying sizes, a good placement strategy is necessary. We implement a greedy brick selection method that calculates the surface area of each brick and prioritizes placing the largest bricks first, followed by progressively smaller bricks. This approach enhances

Figure 5: Sub-stages of the *Filling with Bricks* phase.

model connectivity, as larger bricks generally contribute to stronger structural integrity by connecting more bricks. Once all available placements for a given brick type are exhausted, the algorithm discards that brick and moves to the next largest available brick, continuing until all possible placements are completed.

To further improve brick placement and enhance the model's visual symmetry, we introduce a mirroring algorithm. This algorithm checks the mirrored voxel positions across the Y-axis (the axis along the car's length) for each brick placement. If empty space is available, it places a corresponding mirrored brick on the opposite side. The algorithm also prioritizes placements where mirrored positions are available before placing other bricks, contributing to a more balanced and aesthetically pleasing design.

The following subsections describe the sub-stages involved in filling individual components, as illustrated in Figure 5. The outputs of each sub-stage are visualized in Figure 6.

3.3.1 Wheels

The first components to be placed in the model are the wheels. The best-fitting LEGO wheels and tires are selected based on the diameter of the corresponding wheel component in the 3D model. Before placing the wheels, the connecting brick that holds them must be aligned with the actual 3D model's wheel center. This alignment is achieved by shifting the wheel's center to the closest calculated fitting hole in the connecting brick. Once the positioning is corrected, the wheels, their connecting pins, and the supporting bricks are placed into the model at the pre-mapped wheel component locations.

3.3.2 Cabin

The cabin consists of two main parts: (1) the windshield and (2) the remaining cabin structure, including the roof. The windshield is manually selected based on the cabin height, choosing between a lower windshield (6 LEGO units) and a higher windshield (9 LEGO units). Once selected, the windshield piece is placed at the front of the car, oriented backward. In cases where the cabin height exceeds the windshield height, adjustments are made by increasing the height of the voxelized and component grids. After this adjustment, the remainder of the cabin is filled using sloped, smooth, and standard LEGO bricks, with the final step being the addition of the roof bricks.

3.3.3 Sloped and Smooth Bricks

Sloped bricks are placed first, followed by smooth bricks. A key consideration for both of these brick types is that they have a smooth top surface (see Figure 3), preventing other bricks from being attached above them. Consequently, they can only be placed at the very top of the model. To ensure correct placement, the voxelization grid is analyzed to identify all voxels above the sloped and smooth bricks. If these voxels contain only empty space, the placement is approved, and the corresponding brick is inserted. This process continues iteratively until all available positions for sloped and smooth bricks are filled.

3.3.4 Thick and Thin Bricks

Thick and thin bricks form the majority of the bricks used in the generated model. The filling process begins with thick bricks, which have a height of 3 LEGO units, ensuring structural stability and efficient space coverage. Once all available positions for thick bricks are filled within the current layer, thin bricks (1 LEGO unit high) are placed to fill smaller gaps and add finer details to the model.

3.3.5 Fitting Lights

The front and rear lights are filled using the same method applied for thick and thin bricks, with an additional constraint that ensures only the specific voxels assigned to lights are populated. To differentiate the lights from other model components, distinct colors are applied: yellow for front lights and red for rear lights. All other components, except wheels, pins, and the windscreen, retain the default car body color.

3.4 Fixing Model Connectivity

Up to this stage, the LEGO models are generated without explicit connectivity considerations. If physically assembled, some bricks may lack proper connections, causing structural weaknesses. To address this, we implement a connectivity enforcement method that ensures all bricks remain securely attached.

The connectivity problem is similar to a graph connectivity problem, where each brick is treated as

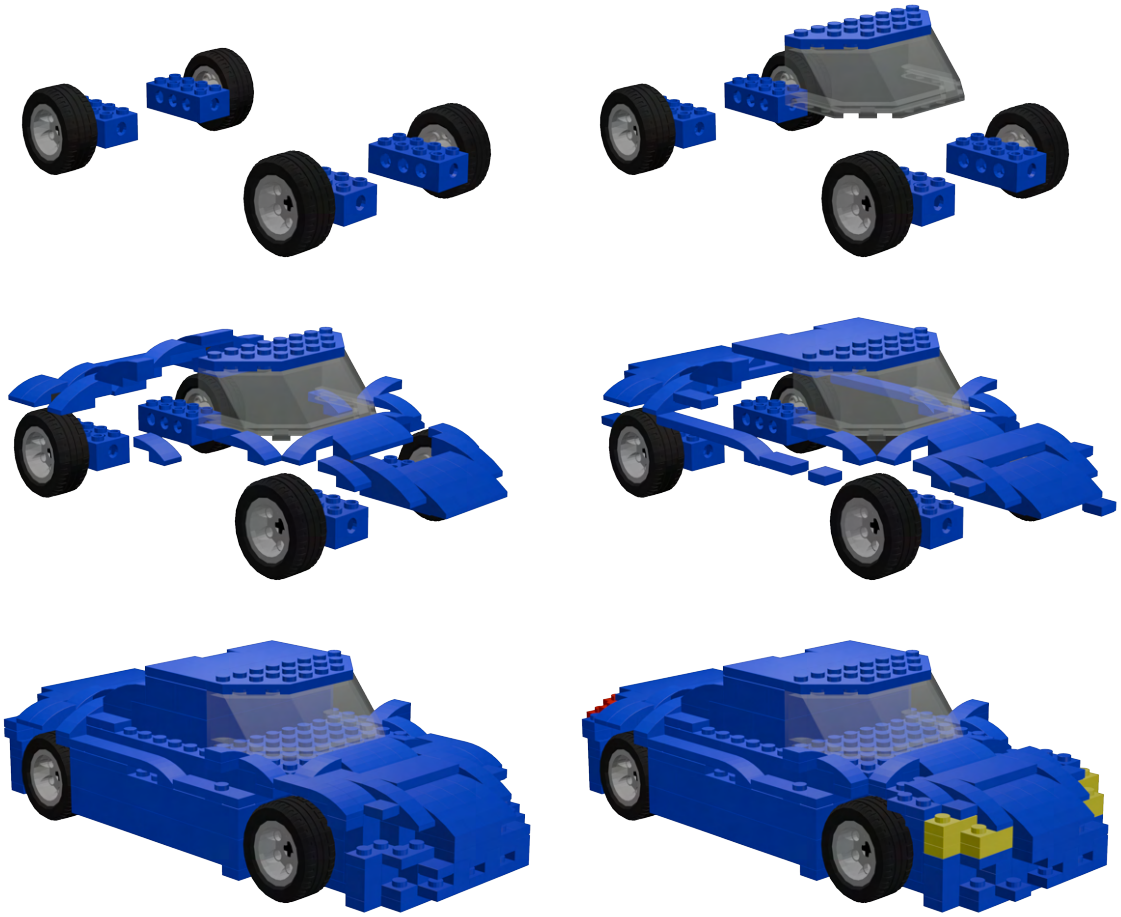

Figure 6: Outputs of individual sub-stages: Wheels (top-left), Cabin (top-right), Sloped and smooth bricks (middle row), Thick and thin bricks (bottom-left), and Fitting lights (bottom-right).

a graph node. A connected LEGO model should form a single connected graph; otherwise, disconnected brick clusters will lead to instability. To analyze this, we traverse all bricks using a depth-first search (DFS) algorithm, identifying all distinct subgraphs (disconnected regions). The goal is to merge these subgraphs into a single cohesive structure.

Each individual brick initially lacks information about its surroundings. To resolve this, we modify the brick data structure by introducing two additional sets:

- *Connected Bricks Set*, which stores direct connections between bricks.
- *Neighboring Bricks Set*, which identifies adjacent but unconnected bricks.

A grid-based representation is created to track the occupied voxels of each brick using a unique ID. This enables efficient processing of connections.

Once all subgraphs are identified, the algorithm proceeds with merging. The largest subgraph (containing the most bricks) is designated as the main structure, while smaller subgraphs must be connected to it. To achieve this, a random brick from a disconnected subgraph is selected, along with a neighboring brick from the main structure. The two selected bricks are removed, and the vacant space is analyzed to determine the largest possible brick that can bridge the gap between both subgraphs. The empty space is filled with the largest brick that fits, ensuring strong and stable connections.

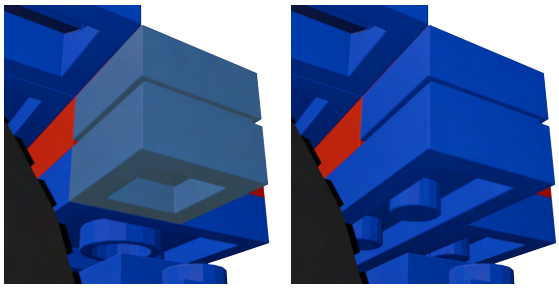

Figure 7: The output of the component mapping stage.

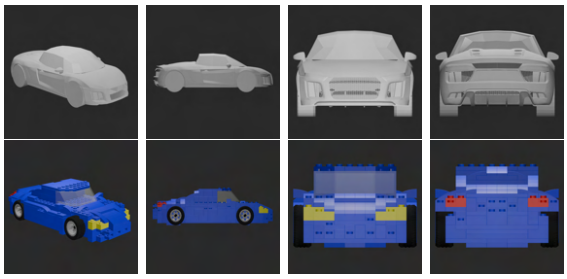

Figure 8: Comparison between input mesh model and LEGO model of sports car from different angles.

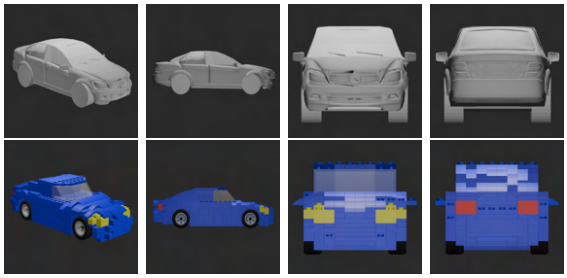

Figure 9: Comparison between input mesh model and LEGO model of sedan from different angles.

For instance, if an empty space of size $4 \times 2 \times 2$ is detected, it is filled using two $4 \times 2 \times 1$ bricks, effectively linking the two subgraphs. This approach is adaptable to different empty space dimensions and considers the material type of the affected voxel, ensuring the newly placed bricks retain the correct material properties (e.g., front lights, rear lights, or car body). The output of the this stage can be seen in Figure 7. In some instances, the process of connecting the subgraphs causes some bricks to be removed from the model altogether, as seen in the last row images of Figure 6.

3.4.1 Adjusting Brick Materials

Since the connectivity algorithm introduces new bricks to merge disconnected regions, some bricks may be placed across multiple components, leading to incorrect material assignments. To correct this, a material validation method is implemented. Each brick is checked to ensure that its assigned material corresponds to the component in which it is fully located. If a mismatch is detected, the material is updated to the correct one. This step is especially critical for front and rear lights, preventing unintended color inconsistencies in the final LEGO model.

4 RESULTS

We present the output results of our approach for various 3D car models. The method was applied to three distinct car types: a sports car (Figure 8), a sedan (Figure 9), and an SUV (Figure 10). These

Figure 10: Comparison between input mesh model and LEGO model of SUV from different angles.

models were chosen due to their differences in size, cabin height, and overall shape, providing a comprehensive context for analyzing the various stages of our approach.

Additionally, we performed a performance analysis by measuring the execution time for each stage of the approach across all three cases displayed in Table 1. The total time required to generate a LEGO model varies depending on factors such as the randomness in brick layout generation and the complexity of the connectivity function.

A notable issue observed during the model connectivity stage was the potential occurrence of infinite loops. This happens when a specific brick (referred to as a "static" brick) cannot be removed, and no alternative bricks are available for placement, causing the algorithm to become stuck. To prevent this, an exit condition was implemented to terminate the process if no viable solution is found. Consequently, the execution time for the model connectivity stage is not always consistent across multiple runs.

To evaluate the computational efficiency of the approach, we measured the time required for both the voxelization process and the complete LEGO model generation. Each test was executed 10 times, and the average results were recorded. The experiments were conducted on a system equipped with an AMD Ryzen 7 7800X3D processor, 32GB of DDR5 RAM, and an AMD RX 7900XTX graphics card.

The total time required for generating the LEGO models varies significantly depending on several factors, including the complexity of the input model, the placement of bricks, and the number of disconnected subgraphs that must be merged.

5 DISCUSSION

The performance analysis results (Table 1) indicate that the two most time-consuming stages in our system are voxelization and fixing connectivity stage. The time required for voxelization is directly influenced by the size and complexity of the input

Table 1: Average total processing times for the test models and average times for each stage.

Car	Triangle count	Voxelization	Component mapping	Wheels	Cabin	Sloped	Smooth	Thick	Thin	Lights	Fixing connectivity	Total generation
Sports car	579,027	44.092 s	14.805 s	0.245 s	0.038 s	2.887 s	1.372 s	0.248 s	0.551 s	0.251 s	10.953 s	**31.350 s**
Sedan	70,926	46.358 s	4.509 s	0.170 s	0.0378 s	3.305 s	1.486 s	0.263 s	0.752 s	0.256 s	7.645 s	**18.424 s**
SUV	7,812	1.368 s	2.636 s	0.171 s	0.037 s	2.245 s	1.221 s	0.233 s	0.446 s	0.315 s	10.542 s	**8.356 s**

mesh model—larger polygonal models result in significantly longer processing times.

On the other hand, the execution time of the connectivity fixing stage varies for a different reason: randomness. Due to variations in brick placement, the algorithm may need to connect a different number of subgraphs in each run. Additionally, certain bricks are classified as "static" bricks, meaning they cannot be removed once placed. This can lead to an issue where the algorithm continuously removes and replaces the same brick, failing to establish a necessary connection. If this process reaches a predefined threshold of unsuccessful attempts, the algorithm removes the problematic bricks altogether, leading to extended processing times.

By analyzing the input and output images in Figure 8-10, it is evident that the generated LEGO models closely resemble their corresponding input mesh models. This could be further validated through a user study in future work.

A key limitation of our method is that it is constrained to a predefined set of car components (wheels, cabin, and lights). Future extensions could introduce additional components to enhance model diversity and realism. Furthermore, the system is restricted to a fixed set of LEGO bricks, orientations, and placement rules. Expanding the available brick types and refining placement strategies could improve both structural stability and visual fidelity in future iterations of the approach.

6 CONCLUSION

In this paper, we explored the problem of generating LEGO car models from input 3D mesh models. We proposed, defined, and tested a solution designed to aid LEGO designers by automating and accelerating the early phases of LEGO model creation. Our method systematically converts 3D car models into LEGO-compatible representations through a structured pipeline that includes voxelization, component mapping, structured brick placement, and connectivity optimization.

The results demonstrate that our approach effectively translates input meshes into recognizable LEGO car models while ensuring structural connectivity. Future work could focus on several key enhancements. Machine learning techniques could be integrated to improve brick selection and placement strategies, incorporating more advanced building techniques used in official LEGO models. The 3D car model could be automatically segmented into individual components as presented in [7]. Refining the connectivity algorithm to handle complex edge cases without resorting to brick removal would improve the structural integrity of the generated models. Expanding the set of supported LEGO bricks, orientations, and components would allow for more intricate and diverse designs. Automating the generation of step-by-step assembly instructions would make the system more practical for real-world model reconstruction. Finally, conducting user studies to evaluate the perceptual similarity between the generated LEGO models and their original 3D counterparts would provide valuable insights for further refinement.

Overall, our system provides a foundation for automating LEGO car model generation from 3D mesh data, with potential applications in digital prototyping and interactive design workflows. Addressing the identified limitations and exploring future improvements could streamline LEGO car model design while enhancing both structural and aesthetic quality.

7 REFERENCES

[1] Hao Cai, Yanjia Chen, Lingling Xu, Khalid Elbaz, and Chengdian Zhang. Intelligent building system for 3d construction of complex brick models. *IEEE Access*, 8:182506–182516, 2020.

[2] Zhao Dong, Wei Chen, Hujun Bao, Hongxin Zhang, and Qunsheng Peng. Real-time voxelization for complex polygonal models. In *12th Pacific Conference on Computer Graphics and Applications, 2004. PG 2004. Proceedings.*, pages 43–50, 2004.

[3] Rebecca A. H. Gower, Agnes E. Heydtmann, and Henrik Gordon Petersen. Lego: Automated model construction. In *32nd European Study Group with Industry*, 1998.

[4] Jian Huang, Roni Yagel, Vassily Filippov, and Yair Kurzion. An accurate method for voxelizing polygon meshes. In *IEEE Symposium on Volume Visualization*, pages 119–126, 11 1998.

[5] Sangyeop Lee, Jinhyun Kim, Jae Woo Kim, and Byung-Ro Moon. Finding an Optimal LEGO® Brick Layout of Voxelized 3D Object Using a Genetic Algorithm. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO '15, page 1215–1222, New York, NY, USA, 2015. Association for Computing Machinery.

[6] K. Lennon, K. Fransen, A. O'Brien, Y. Cao, M. Beveridge, Y. Arefeen, N. Singh, and I. Drori. Image2LEGO®: Customized LEGO Set Generation from Images, 2021.

[7] Qing Liu, Adam Kortylewski, Zhishuai Zhang, Zizhang Li, Mengqi Guo, Qihao Liu, Xiaoding Yuan, Jiteng Mu, Weichao Qiu, and Alan Yuille. Learning part segmentation through unsupervised domain adaptation from synthetic vehicles. In *CVPR*, 2022.

[8] S. Luo, Y. Yue, C. Huang, Y. Chung, S. Imai, T. Nishita, and B. Chen. Legolization. *ACM Transactions on Graphics*, 34:1–12, 2015.

[9] K. Min, C. Park, H. Yang, G. Yun, and Y. Grim. Legorization from Silhouette-Fitted Voxelization. *KSII Transactions on Internet and Information Systems*, 12, 2018.

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

[11] Romain Pierre Testuz, Yuliy Schwartzburg, and Mark Pauly. Automatic generation of constructable brick sculptures. In *Eurographics 2013-Short Papers*, page 81–84, 2013.

[12] R. Thompson, E. Ghalebi, T. DeVries, and G. Taylor. Building LEGO Using Deep Generative Models of Graphs, 2020.

[13] J. Zhou, X. Chen, and Y. Xu. Automatic Generation of Vivid LEGO Architectural Sculptures. *Computer Graphics Forum*, 38:31–42, 2019.

[14] Mingjun Zhou, Jiahao Ge, Hao Xu, and Chi-Wing Fu. Computational design of lego® sketch art. *ACM Trans. Graph.*, 42(6), December 2023.