COMP4423 – Assignment 1

HEO Sunghak (21097305D)

1. Introduction

With the rapid development of computer vision techniques, many developers have released visual transformation applications that can capture camera input and convert it into stylized outputs, such as cartoon filters, virtual avatars used by online streamers, and augmented reality effects on social media. These application shows how images can be transformed into another ways of representations while keeping the essential information. Like such applications, this project aims to convert an image into a complete LEGO-style image that consists of combinations of LEGO bricks. The project includes constraints such as limiting the number of LEGO bricks while preserving visual information of the original image.

Building a robust pipeline that operates under realistic constraints is significant. The implementation is in a modular structure consisting of image preprocessing, color quantization, grid construction, brick selection, and rendering output. For Task 2, the pipeline converts the image into grayscale and match it to three colors – black, gray, and white - using only 1x1 LEGO bricks. For Task 3, the program uses multiple brick shapes (e.g., 4x2, 2x4, 2x2, 1x2) and more colors. For Task 4, the whole functions are deployed in a real-time camera, testing robustness and effectiveness.

The goal of this assignment is not only to produce a high-quality image, but to build a well-justified pipeline and report, evaluate its robustness in practical scenarios, analyze its weaknesses and improvements.

2. Implementation

2.1 Overall System Design

The system is designed as a modular image processing pipeline making a LEGO-style image from an input within constraints. The entire architecture adheres to a sequential transformation process, in which each phase executes a specific task and forwards its result to the subsequent phase. This modular design enhances clarity, testability, and extensibility.

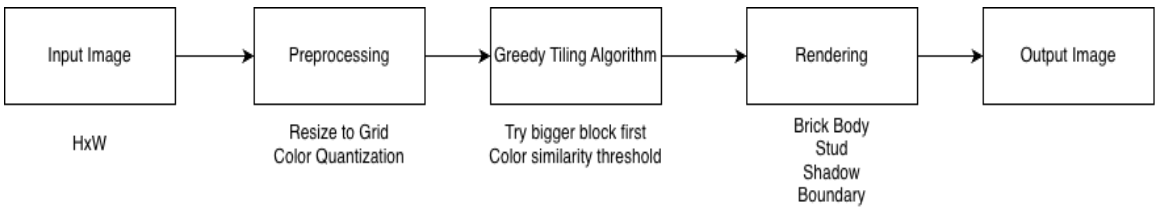

Figure 1. Overall Pipeline of the LEGO-style image generation system

The pipeline consists of the following main components:

1) Image Loading

The system provides the two input modes: static image and real-time camera capture. In static mode, an image file is loaded by PIL library. In real-time mode, OpenCV captures frames from a webcam. This setting shows that the algorithm works in both experimental setting and real time application.

2) Preprocessing and Grid Construction

A constraint of this assignment is there should be no more than 100x100 bricks for an image. To satisfy this, resizing an input into 100x100 grid was necessary.
A scaling factor is computed as:

$$scale = \min\left(\frac{\text{max\_brick}}{H}, \frac{\text{max\_brick}}{W}, 1.0\right)$$

Where H and W are the height and width of the input image.

Then, the height and width of a grid image is determined by:

$$grid_h = scale * H, grid_w = scale * W$$

The image is resized into a grid using nearest-neighbor interpolation which is fast enough and good to preserve LEGO-like characteristics. Each pixel of the grid represents one 1x1 lego's position and color.

3) Color Processing

Color processing is applied to handle noise and make the color of each LEGO consistent.

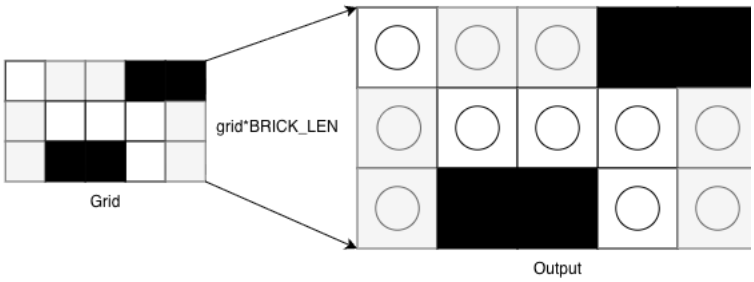

Figure 2. Color Quantization and Mapping in Task 2

For Task 2, a grid of an input image is mapped into three discrete colors (black, gray, white) after transforming into grayscale. As shown in the Figure 2 above, each pixel in the grid determines both the position and the color of the corresponding LEGO brick in the output image.
For Task 3, colors are handled in a different way. The original RGB values in the input image are not changed even after resizing to grid. The number of colors is implicitly reduced during the brick placement algorithm. This will be explained in the next step. When choosing which shape of a block to be placed, the algorithm checks whether all pixels within the candidate region are similar enough to the base pixel.

$$D = |R1 - R2| + |G1 - G2| + |B1 - B2|$$

If the Manhattan distance shown above equation between the base pixel and the compared pixels in the candidate region is less than the threshold t we initially defined, the color of the block is determined as the mean of RGB values of the pixels in the block region.

This algorithm results in several points:

- The overall color variation looks smooth.
- Detailed regions are described by 1x1 bricks to keep the original shape as much as possible.

The threshold t is very significant for smoothness and coherence of the image. According to Figure 3 and Table 1, when t=30, it keeps the color details very well while it also creates increasing number of small bricks. On the other hand, when t=100, the bricks will be bigger so the number of small bricks is reduced, but the color description is simplified.

Figure 3. Final Output When Threshold t = 30 and 100

	T=30	T=100
1x1	1730	348
1x2	1628	1350
2x1	325	128
2x2	241	142
2x4	387	741
4x2	38	25
Total	4349	2734

Table 1. The Number of Bricks When Threshold t = 30 and 100

4) Brick Placement Algorithm

For Task 3, the brick placement algorithm decides how multiple LEGO brick shapes are arranged while preserving the structure and features of the input image and reducing the total number of bricks.

Each pixel of a grid represents an RGB color value. Iterating from top-left to bottom-right, it attempts to place the largest possible brick from a predefined set of shapes (4x2, 2x4, 2x2, 2x1, 1x2, 1x1) to unoccupied position.

The algorithm checks two conditions to place a block to a particular position:

1. The brick must be within the boundary of the grid size.
2. All pixels in the candidate brick region must be of a similar color to the base pixel.

The second condition is already explained in the previous step.

When a brick is placed, the placed positions are marked as occupied so that already placed position will not be placed by other bricks again.

Additionally, the brick summary, a total number of bricks and count for each brick type, is recorded during the execution of this algorithm.

5) Rendering

Using the brick placement information from the previous step, the output image is rendered.

The height and width of a output image is defined by:

$$out_h = grid_h * BRICK\_LEN, out_w = grid_w * BRICK\_LEN$$

where BRICK_LEN is the length of a 1x1 brick in pixel.

The information of each brick is described by its top-left coordinate (x,y), width dx, height dy, and its color. This part performs the following step:

1. Brick Body Rendering:
   Using the brick information, the rectangular region of the brick is colored with an assigned RGB color. In this step, the boundary of each brick is also drawn as well.
2. Stud Rendering
   The color of the stud is different from its brick color to produce more visually appealing output. A factor is multiplied to the mean of the RGB value of the brick. If the mean is bright, the stud color becomes slightly darker. If the mean is dark, the stud color becomes lighter.
3. Shadow Rendering
   The color of shadow is a fixed value as it is shown to be more appealing after several experiment.

Drawing shadow was a bit tricky as it seems to require some mathematical operation to draw crescent shape. However, I found a simpler and more efficient way, which is overlapping two ellipses to create crescent.

6) Brick Summary

This step only requires to print out the recorded count from the previous step.

7) Real-Time Deployment

Real-time deployment was implemented using OpenCV to capture video frames from camera. Each frame was processed through the same pipeline as the previous tasks.

The main purpose of this step is to prove that this pipeline is efficient enough to compute real-time deployment. To keep a reasonable performance, the input frame is resized into 100x100 to reduce computational cost during brick placement and rendering, which are time-consuming operations.

3. Method

3.1 Program Design and Testing

The program was made to be simple and easy to understand. It basically follows modular structure, a structure that is divided into parts and each part of the pipeline was built and tested on its own before it was added to the whole system. The main parts of the program are image preprocessing, grid construction, color processing, brick placement, rendering and real-time deployment.

During implementation, each step and module was tested in a Jupyter Notebook. After implementing a specific module, its results are visualized to make sure it is working well. For example:

- Quantization was verified by checking each pixel is mapped to the correct discrete color.
- The brick placement was examined by printing each brick coordinate and checked no overlapping occurred.

In addition, the testing was performed with live camera input for the purpose of making real-time operations about the stability and response times for frame processing, and to make sure the pipeline was operating as intended under constant input.

Benchmarking testing was not performed with a formalized testing framework, however, each module was unceasingly subjected to continuous iterative testing of intermediate outputs and changes to parameters, to ensure correctness and stability before the final integration of all modules.

3.2 Robustness in Real-World Scenarios

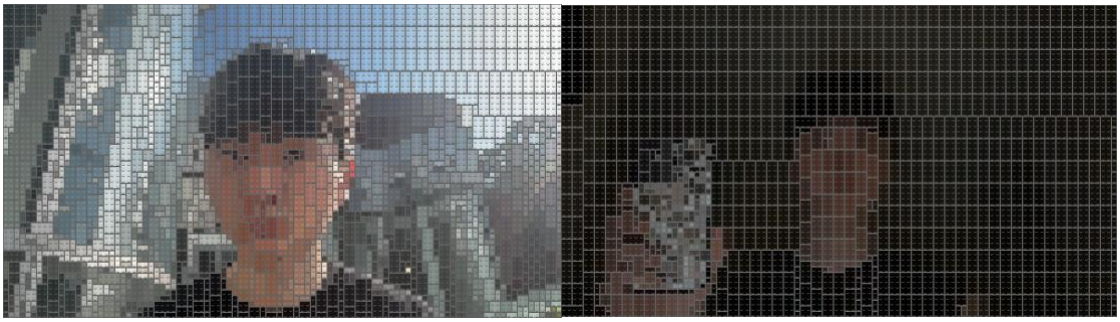

Figure 4. Testing Under Light and Dark Environment

Robustness is measured based on the situation where cameras are deployed in a real-time manner. Input frames will experience varying resolution, lighting conditions, and scene complexity. Therefore, the system is intended to deliver reliable performance across a wide range of input conditions.

Firstly, the computation cost of an algorithm does not rely directly on the original image resolution. Prior to the actual placement of bricks, the input image will be resized to fit a bounded size grid (e.g., not exceeding 100×100). This means that all subsequent processing (e.g., comparing color and actual placing of the brick) is performed on a fixed size grid, therefore we can reliably predict execution time no matter what the original resolution was.

The second consideration is changing the threshold value t determines how visually robust a result is. Changing the threshold value will not affect how much time is spent analyzing each pair of pixels, but changing the threshold value will indirectly affect how many bricks will be distributed throughout the render. As explained earlier, a lower threshold will result in a higher number of 1×1 bricks, resulting in the retention of finer detail in the rendering process, while a higher threshold allows for larger bricks to be produced when smoothing color changes between two pixels. This may affect rendering time due to the potential number of studs being drawn. Nevertheless, the overall time complexity of the algorithm will not be drastically affected due to either the threshold's effect or to the number of studs.

A greedy tiling algorithm uses deterministic iteration from top-left to bottom-right, which allows for non-recursive backtracking or global optimization and thus a predictable time per frame. Maintained times for multi-threaded operation are critical to sustained function in an on-the-fly implementation.

While the constraints of the grid size (e.g., rows and columns), deterministic operation, and bounded color comparisons lead to steady performance during realistic scenarios.

3.3 Problems and Solutions

There were many design decisions made throughout the development process that must be balanced between visual quality and computational efficiency. Not all the decisions

made will not be addressed in this section, as these are minor implementation issues. However, some of the more critical engineering considerations are discussed here along with the rationale behind the final design decisions.

1) Global Color Quantization vs Region-Based Color Merging

One major design question was whether Task 3 would implement explicit global color quantization prior to brick placement.

Implementing global quantization first will eliminate many possible colors from use which may reduce the number of colors available while making brick placement easier and creating a more "LEGO-like" output because of limited color variety. However, this would sacrifice detail and create artificial color boundaries prior to considering structural grouping.

Instead of performing global color quantization, this system used region-based color merging while placing bricks. When the system combined similar colored pixels into a larger brick, it compared the difference between their colors to a pre-defined threshold to determine if they should be grouped together. This method will allow colors to be simplified across the spatial area where they are found and allow adaptive simplification of colors.

The latter choice is good because:

- Details are preserved in high-frequency areas.
- Color choice and boundaries become more context aware and natural.

In this subjective matter, I selected the latter.

2) Brick Boundary Rendering Strategy

The next consideration was how to render boundaries between bricks.

The two considered options are:

1. Drawing explicit grid lines over the rendered image.

2. Making a small gap around each brick to implicitly create boundaries with background color.

The first option is straightforward, efficient, and easy to implement. On the other hand, using a gap and rendering each piece slightly smaller than the area it occupies, the second option, ensures that an appropriate amount of space is left around each piece for separation purposes and allows pieces to maintain a single unified look regardless of additional size. This implementation requires more careful debugging.

Although there were no significant performance differences between the two methods, just drawing lines over the rendered image was selected due to its simple implementation and easy understandability.

3) Shadow Rendering Strategy

To make it more LEGO-style, it was decided to also put a shadow effect on studs. At first, the idea of producing a crescent shape for the shadow mathematically was considered, but it would have required the use of geometric masking operations, removing circular areas that overlapped. Nonetheless, this method is too complex and potentially creates computation overhead for real time deployment.

Instead, a simpler method was used by drawing 2 ellipse shapes, each with a small positional offset from the other, one representing the shadow and the other the stud highlight. This layering method creates a substantial shadow effect visually while keeping the computation cost very low.

This engineering approach is a tradeoff between mathematical precision and practical performance. It has been shown that using the simpler method provides the same level of visual enhancement as the mathematically correct approach but does not impact on the performance of real-time hardware.

4) Use of GenAI

- Adding comments to the code
- Brainstorming greedy tiling approach
- Discussing color similarity metric (Manhattan distance, Euclidean Distance)
- Transforming code into modular structure
- Assisting in debugging reasoning
- Exploring alternative shadow rendering strategies

5) How GenAI Understand and Solve the Tasks?

GenAI seems to break down the assignment into small problems. The pipeline is decomposed into well-defined steps for grid representation, color comparison, brick placement and rendering. This made this program modular and maintainable.

GenAI also solves a problem based on well-defined solution on the Internet. Each task is mapped to existing algorithmic patterns. For example, the brick placement is similar to a two-dimensional tiling or covering problem, which I learned when I study competitive programming. My first approach was combination of BFS and greedy, but GenAI suggested providing only a greedy approach, focusing on lowering computational

complexity. This shows that GenAI often suggests commonly used strategies that it can learn from public resources.

However, when GenAI evaluates the visual quality of multiple output images in terms of LEGO-style and structural realism, slight differences were not always clearly spotted. Therefore, evaluation of the image quality depends on a human eye since the rendered output is visually complex and in not common style. GenAI is often better at reasoning about the structure of the algorithm than in assessing whether the final output resembles realistic styles of LEGO construction.

6) Conclusion: Limitations and Improvement in GenAI's solution?

While GenAI is becoming increasingly versatile and there have been many improvements to the existing systems, there are still a few limitations associated with GenAI.

As discussed earlier, GenAI has limited ability to judge visual quality. While it can understand various problems and recommend a solution, it does not yet have the ability to judge whether an output looks like real LEGO to a human. As a result, judgments regarding whether LEGO bricks look natural, realistic, or visually balanced still rely on human judgment. As a consequence, human still needs to properly prompt GenAI for visual refinements.

In addition, human still needs to optimize parameters. In case of tuning the threshold value for color similarity checking, the optimal value must be proven by empirical means rather than letting GenAI produce an optimal value using an image. One way to use GenAI in this case is to have it judge every results. You can ask it how to tune the parameter every time you send the result. However, this is time-consuming and not recommended. Various experimental data values must be handled manually to find the optimal value.

In conclusion, it can be implied that while GenAI can be used as a substantial resource for coding and reasoning, human judgment and ability to utilize GenAI is still significant in working on tasks that require human perception or the tuning of parameters.