Abstract
LEGO® is the leading toy for children and adults alike to build their own three-dimensional models of objects in their day-to-day lives. Despite the endless possibilities of LEGO®, creating a model purely from imagination or even a real-life scene is far too great a challenge for the average LEGO® enthusiast. LEGOify is a novel solution attempting to assist the average user by allowing them to simply take a picture of an object they would like to make in LEGO®, and getting detailed instructions on how to build the desired model.

LEGOify is a modern algorithmic solution to creating instructions for any object from just a picture. The approach consists of previously implemented concepts, such as photogrammetry, but also implements a novel approach of brick placing through a divide and conquer technique.

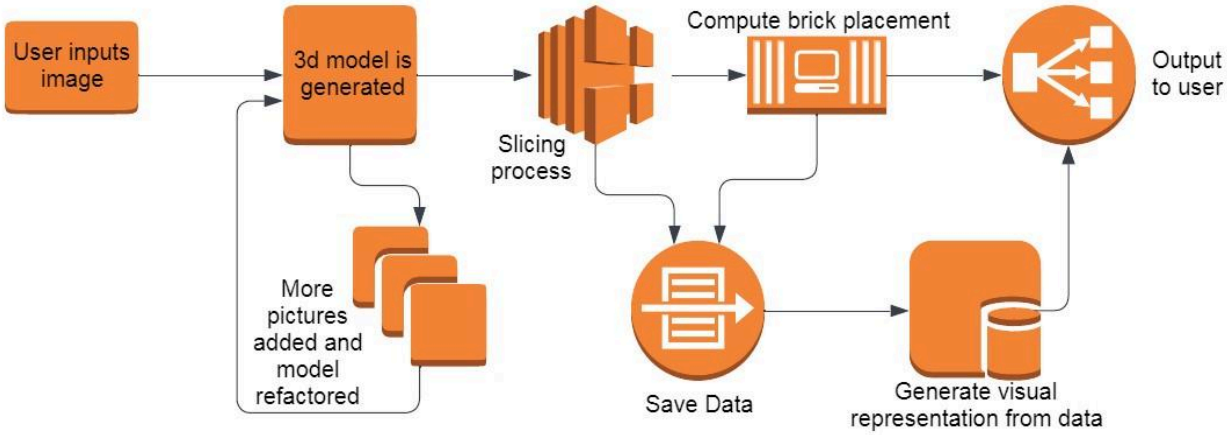
Problem
The problem it is attempting to solve is creating a usable LEGO® model from just a picture. Although this problem has been approached before, this approach is entirely original. The best attempt prior to this one is from MIT, in which an artificial intelligence (AI) model was used to create voxels which are then transformed into LEGO® bricks (voxels are 3 dimensional information points). While the scaling of this approach is better than most other approaches, voxels create a sense of instability. Voxels may create a non-smooth surface on say, a table, or a chair whereas an algorithmic approach would not. There are areas in the solution that capture “dead voxels” and may not be as structurally sound as intended. This is in part due to the AI model deciding whether a voxel is significant or not.

Other solutions include manually building instructions through a separate third party program. Although it is a very failsafe method, it requires extra downloads, and significant time and effort. Furthermore, it is not friendly to the average beginner who does not intend on making instructions for the larger population, rather simply make the model for themselves at any given moment.

Overall, the advantages of my code are: the quick model creation, allowing for users to instantly get the model they want, as well as having a relatively low difficulty level and a high level of reliability. Solutions with the voxels have been proven to create “dead” areas where there are randomly bricks that do not contribute to the overall shape. This algorithmic approach removes this error while maintaining the quick speed of returning an instruction file with ease.

Methods

The overall structure of the program is as follows:



The concepts listed above are explored in detail chronologically to provide more details.

Photogrammetry

Photogrammetry is the technique in which multiple 2d images are coordinated to create an approximate 3d model. This is a very commonly used technique and is made readily useable through the OpenCV library for both Python and C++. Artificial intelligence (AI) models today are capable of guessing the structure with just a single picture. If the user is unhappy with the model given, they are capable of prompting the computer with more pictures to create a clearer image.

Slicing

Utilising the OpenCV library again, we can simply cut the STL file into slices and take a bird's eye view screenshot. We can then grayscale the image in the same step, again utilising OpenCV. However, we realise that the overall detail of the image in its new strictly numerical form, where 1 represents white and 0 black, is far more than what is required. As a result, we can simply condense each 5 x 5 square in the picture prior to the calculation. This allows for a more condensed image. Finally, the data can be saved into an integer array, where each 1 represents an index that must be tiled by a LEGO® brick and a 0 is an open space.

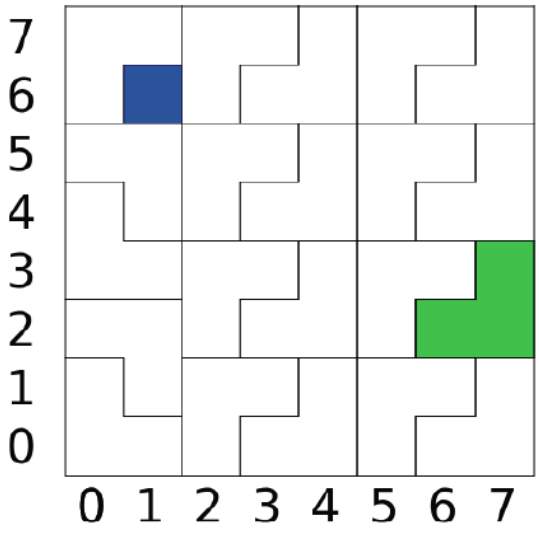
Tiling

To better understand the nature of tiling problems, existing complex problems can give us insight into the solution for our own LEGO® tiling problem. Tiling problems in general are problems which require placing predefined tiles over a given space in an efficient or minimal manner. The following problem is from the International Olympiad in Informatics in 2014 at Taipei, Taiwan.

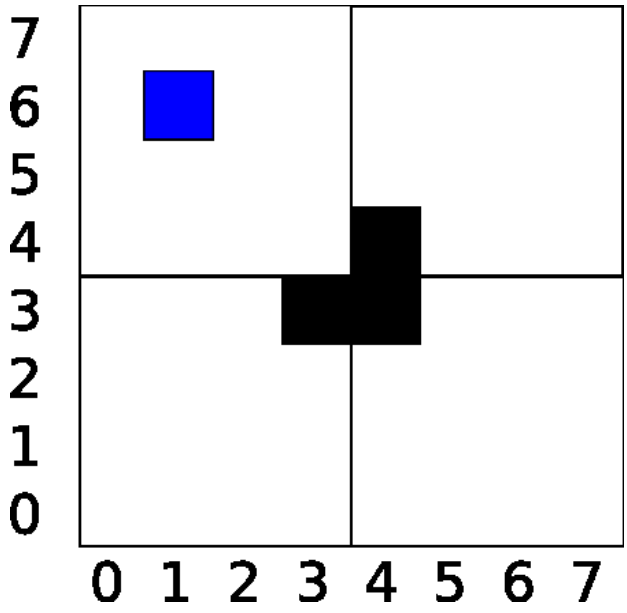
The problem essentially asks the user to tile a square space. The user is given a tromino (displayed below in the sample case) and the coordinate of a single 1 x 1 square within the grid. The user is asked to output any viable tromino tiling to fill the entire board without overlaps or extensions beyond the board. The size of the board is defined as 2^n, with n provided to the user.

We can determine an algorithmic approach through the provided sample case. It must be noted that a recursive brute force approach is not permissible for full marks due to the maximum bound of the tiling area being sized 2^8.

Let us first consider the sample case:



Before solving this case, we must understand that our base case is when $n = 1$ and a single tromino can solve the case as shown to the right.



From the base case, we can make a broader deduction to tile the entire board. It is always necessary to place a tromino on a board such that it “balances” the board. We can define balancing the board as a state in which each quadrant has the exact same number of “disturbances”; number of spaces covered by tiles. Since we are always working towards creating a square board, this would mean forming a square with the tromino. This principle is better displayed through the image shown below.

When we broaden our view to a board with just 4 squares, as displayed on the left, it can be noted that only the large square in the top

left contains the starting tile. In order to retain “balance” on the board, all the other quadrants must also have tiles placed within them. Thus, a tromino is placed at the center in the configuration shown.

Each quadrant can then be examined as its own larger square: that is to say, they can each be split into 4 smaller “sub-quadrants” and then balanced as described earlier. The process of creating progressively smaller, balanced “sub-quadrants” is repeated until a base case is reached (i.e. the shape of the remaining area corresponds exactly with that of a tromino). This strategy represents a form of recursion that is less computationally expensive than the brute force approach presented earlier.

It is important to note that the tromino tiles are each separately considered as 1 x 1 squares when found in a quadrant, allowing for the recursion cases to be similar to that of the starting case.

Observations about tiling problems can be drawn from this problem. It is necessary to have a base case and a reasonable transition state without a large complexity. Thus, we can start building the basis of our own tiling process for the LEGO® bricks.

Base Cases
The base cases are the simplest part of the algorithm. To increase usability and reduce the complexity of the program, a set of LEGO® bricks with the following dimensions was chosen to be the base case:
1 x 2
1 x 3
1 x 4
2 x 2
2 x 3
2 x 4

It must be noted that the 1 x 1 brick is not included in the base case as it somewhat ruins the structure. It is much harder to get connectivity across multiple levels when assembling, and can ruin the overall recursion structure. As well, the minor loss of a 1 x 1 square is often not relevant to the overall structure being made.

Divide and Conquer Transition
The first—rather obvious—observation is that bricks only need to be placed where there are non-zero indices in our previously recorded black and white digit array. Thus, a bounding box can be employed to ensure that blank indices are not searched recursively. The bounding box ensures that all marked squares are captured whilst minimizing the total area. Searching blank squares recursively wastes overall computation time as there is no further way to break down the chunk into LEGO® pieces. Thus, we can simply skip over the blank areas not within the bounding box.

The bounding box can be checked for base cases (i.e. shapes corresponding exactly to a singular LEGO® from the aforementioned set). If no base cases are found with the current dimensions, it suggests that we must continue to divide our area into further subsections. We can perform this in a greedy way by dividing the bounding box in half along its greatest dimension. This division creates two smaller bounding boxes, which can themselves be divided

Eventually, our method will result in one of the base cases, or at times, a 1 x 1 square. Since the single block is not one of the base cases, no additional processing is required. However, if a true base case is found, additional computation is required to determine whether a brick of the corresponding shape must be placed. The threshold value for placing a brick is 0.5: at least 50% of the proposed area of the brick needs to have been computed as a non-zero index in order for it to be placed. If a larger base case is found but does not meet the threshold value, recursion can continue as it may be possible to place smaller bricks that meet the threshold value.

Time Complexity Analysis
The codebase and its time complexity is better understood when it is examined as a series of functional modules: namely, precomputation, divide and conquer, and postcomputation.

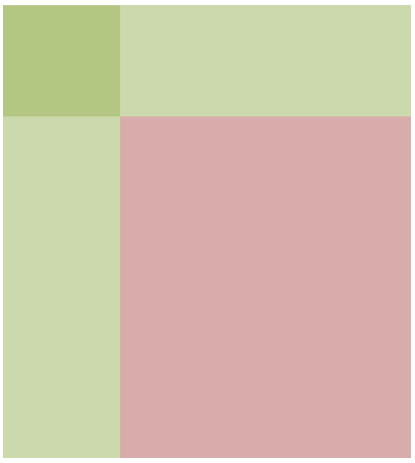
Precomputation
The precomputation consists of finding the original bounding box, and creating a 2D prefix sum array. Both are done simply through looping in $O(n^2)$ time. The design space can be understood as a Cartesian coordinate system. As such, the bounding box can be calculated by keeping track of the minimum and maximum x and y values with a non-zero value, and the 2D prefix sum is simply addition across all the rows and columns.

Divide and Conquer
Since we are continually dividing our area by 2, we can maintain that we will at some point reach our base case of a 1 x 1 square. In order to calculate how long this would take, let's let W be the width of the bounding box, and H the height. Thus, the maximum number of divisions to reach the lowest case of 1 x 1 is $\log(\max(W,\ H))$. Thus, our overall complexity for this step is $O(\log(\max(W,\ H)))$, meaning that our algorithm scales very well even with large data.

It is important to note that the 2D prefix sum array permits us to compute each step at $O(1)$, and it would not otherwise be possible for our data to scale as so. Since we require the total of a rectangle area in the array to determine whether it meets the threshold, we must calculate this quickly (repetitive calculation).

If we attempt to brute force the calculation every time, this will make each computation step require $O(n^2)$ complexity. Applying 2D prefix sums we can do the following:

When we add up our 2D prefix sum, we sum across each row and each column. This means that each index represents the sum of all indices where it is the bottom right index, and the top left index is at (0, 0). However, from this 2D prefix sum, we can find the sum of any area in $O(1)$ time.



If we visualise our 2D prefix sum on the left, we can attempt to find the sum in the red square. By taking the value of the bottom right corner of red box, we include the green areas.

However, we can easily remove these from the sum by taking the bottom right indices of the light green areas.

Once we remove these areas, we must note that the dark green area has been removed twice (when both the light green rectangles were removed). Thus, we have to add back the dark green square to ensure that it is not taken away twice.

Postcomputation

The main purpose of the post computation is to make the solution displayable in an efficient manner. The method used to save the block placements was simply to put the block number in the indices to which the block is placed. However, to once again do this efficiently, we can use an approach that is the reverse to the 2D prefix sum: a 2D difference array.

If we want to insert a block, we can use this difference array technique by adding the value ‘b’ for each of these indices.

When we add up all the indices in a similar manner to a 2D prefix sum array, the intended red box will pick up the values intended.

The benefit is that our complexity is now simply $O(n^2)$, as we have $O(1)$ updates and a single n^2 loop after all the updates are completed. THis is in comparison to simply looping through each block and updating them without this technique, resulting in an $O(n^3)$ complexity.

Depth-First Search (DFS) and Final Brick Placement
From our final array, we are given the information of which square belongs to which block, but not exactly where each block is placed. From here, we can simply loop through and perform DFS if we come across a non-zero value to find the full block by searching for adjacent squares that share this value. We can then place the block and continue looping through the array, while marking all indices belonging to the block as visited.

The overall complexity is $O(n^2)$, as we are simply going through every possible index to check for any LEGO® blocks.

Summary

The overall code scales well at an $O(n^2)$ level although the overall algorithm is much more efficient. However, our code does not need further optimization due to the sheer size it is capable of performing at currently. Theoretically speaking, $n \leq 5000$ is our upper bound on the data by the given time complexity. Through experimentation, we find that arrangements upwards of 300 LEGO® bricks per slice are formed at this upper bound value. Since the average recreational LEGO® builder is unlikely to use even 300 LEGO® bricks in a single design, the algorithm is sufficiently powerful for the intended users.

File Summary
slice.py: slices the STL file at the provided heights.

eachslice.py: iterates through each slice and converts them to a black and white file. From this black and white file, a 2D boolean array is generated and saved.

dividenconquer.cpp: iterate through the boolean files and apply the algorithm described. Save a set of 2D integer arrays depicting the placement of each brick on each slice

draw.py: apply DFS and iterate through the 2D integer arrays to generate the pictures. Generate the final graphics, essentially translating the 2D array of integers into a visual .png format. Scale each index of the array to become a 10 pixel x 10 pixel square in the final image.

Testing and Proof of Concept
Overall testing of the program was done with a ball shape. This is due to the concavity of the sphere and the different structural requirements. It is quite definitive to claim that if a sphere is to be created accurately, any model may be created accurately. By using a sphere, it is easy to verify the validity of each section of code simply by seeing the output visually. There are less components to confuse when checking visually, but also have enough complexity to expose fallacies in the code.

The final demo includes the design for *Toothless* from *How to Train a Dragon*. The correctness of this model suggests that our testing through the sphere is accurate in its entirety.

Limitations

In the current user interface, users are provided solely with the birds-eye view of each layer. Further improvements could include creating each slice as a 3D model, such that the user can rotate around and view the model freely. This provides a better perspective on where exactly to place the blocks.

The program works exceptionally well when scaled to a large degree. Further improvements could be to make an algorithm that adjusts the 2D slices such that when working with smaller models, key parts of the model do not go missing. This would make the overall program more usable, especially for people with fewer LEGO bricks at hand.

Future Development

The code is fully functional at a very high level; there is minimal interaction required from the user to result in the model. However, all the files are currently routed to run locally. Through a service like AWS, the service can be hosted online as a web service with ease. Continuing to update the application with regards to the limitations can guarantee a successful application with a high interest.

Citations

How to make your own LEGO set. (2021, November 15). Brickset.com. https://brickset.com/article/66046/how-to-make-your-own-lego-set

Lennon, K., Fransen, K., O'Brien, A., Cao, Y., Beveridge, M., Arefeen, Y., Singh, N., & Drori, I. (2021, August 18). Image2Lego: Customized LEGO Set Generation from Images. ArXiv.org. https://doi.org/10.48550/arXiv.2108.08477

(2023). Dmoj.ca. https://static.dmoj.ca/data/peg/ioi1403/ioi1403.png

(2023). Dmoj.ca. https://static.dmoj.ca/media/martor/fb4f102a-29bb-4328-aeca-7198b2c16c6a.png

IOI '14 Practice Task 3 - Tile. (n.d.). DMOJ: Modern Online Judge. Retrieved June 12, 2023, from https://dmoj.ca/problem/ioi14pp3

Blelloch, G. (n.d.). Prefix Sums and Their Applications. https://www.cs.cmu.edu/~guyb/papers/Ble93.pdf