Lego Mosaics

Techniques and Algorithms

Deb Banerji

Last updated: 2021-9-16

Contents

- Intro
- Color Distance + Variations
- The Remix Problem
- 3D Mosaics
- Pixelization
- Code Links
- Q & A

Intro

- Who am I
- Who is this presentation for
- Disclaimer
  - Not all concepts here have publicly available code
  - I’ve linked the code for some more polished techniques

What is a lego mosaic

- Also known as ‘Lego Art’
- Brief history

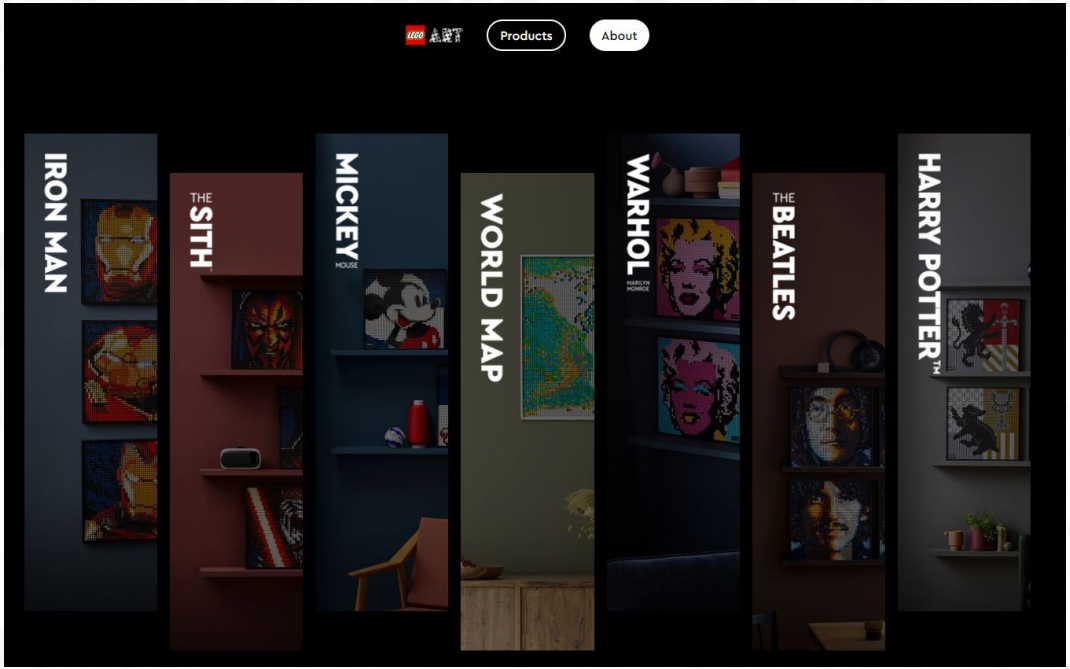
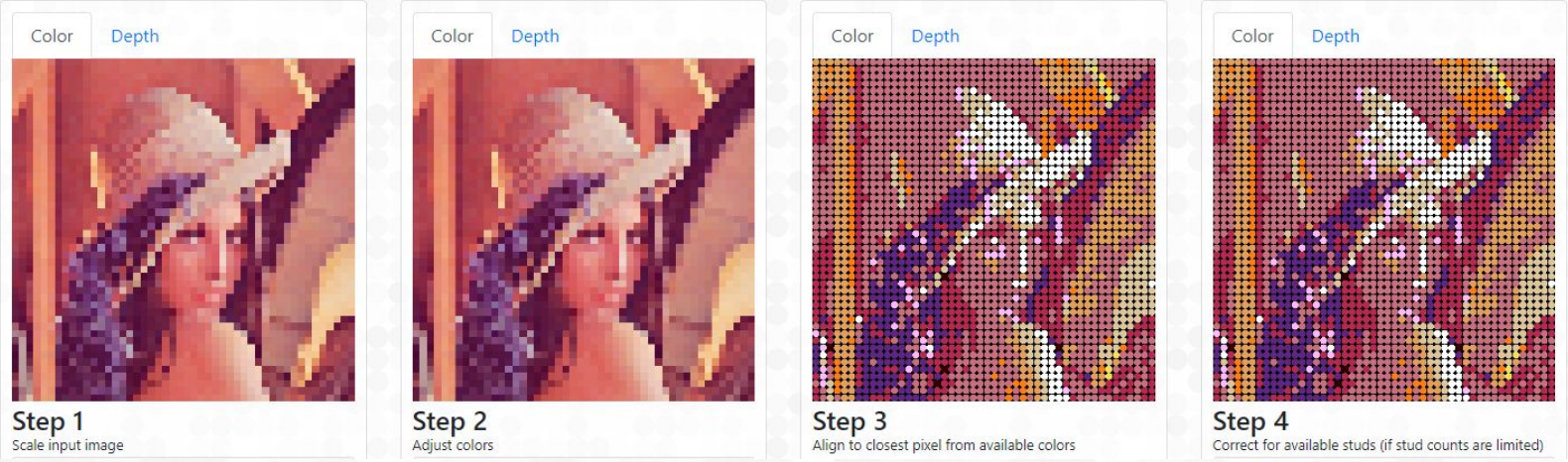


Image source: Lego.com

Lego mosaic software

- Has also existed for a while
- Some even created by Lego themselves!
- Designed to assist humans
- What we're going to explore today



What I won’t speak about

- Project architecture
- Deployment
- Security
- Performance optimizations
- Memory management
- HTML canvas usage
- Interactive pixel editing
- PDF generation
- Neural network deployment + details
- Other software engineering stuff

Don’t panic!

- Don’t worry if you don’t understand the details of the Math
- It’s way more useful to focus on understanding the ideas the Math is trying to capture

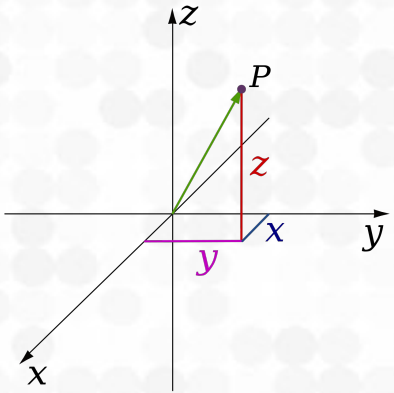
Color distance

- Computer representation of colors (RGB, Hex Codes)
- What is color distance?

colorDist(#FC33FF, #8033FF) < colorDist(#FC33FF, #176C14)

An example color distance function

- Euclidean RGB is relatively simple to code + interpret



Source: https://en.wikipedia.org/wiki/Three-dimensional_space

- More sophisticated functions include CIE94, CIEDE2000, Euclidean LAB, etc.

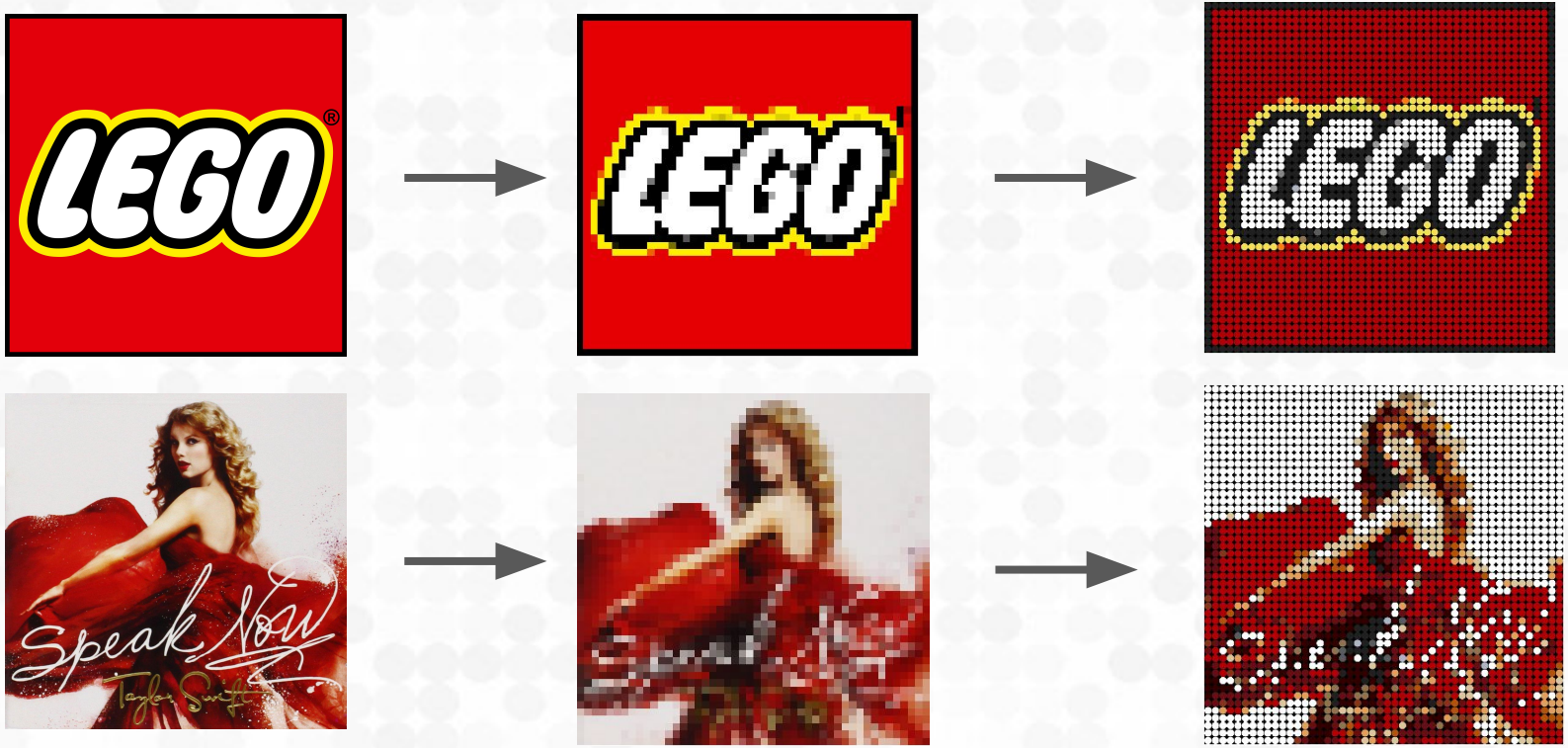
Mosaic creation

- Lego mosaic creation as minimizing a color distance function
- Scaling down + matching available Lego colors to pixels
  - For every pixel, find the ‘closest’ Lego color
- Mathematically, we’re minimizing the sum of the color distances between corresponding pixels

$$\sum \text{colorDist}(\text{original}_{i,j}, \text{replacement}_{i,j})$$

i = 1 to width, j = 1 to height

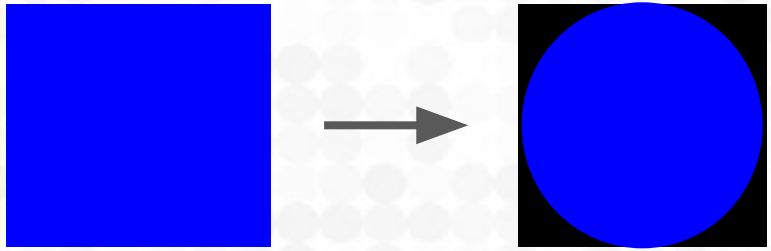
Examples



Source: ‘Speak Now’ Album Art, Taylor Swift, 2010

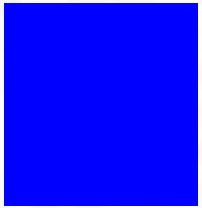
Bleedthrough

- Pictures with round studs as pixels may be darker than expected
- Solution: multiply brightness of studs by $\pi/4$ before alignment



Transparency

- Image formats such as .PNG contain transparency image for each pixel
- We can account for this by adding a transparency factor to the result of each color distance check
  - colorDist(original$_{i,j}$,replacement$_{i,j}$) + transparencyFactor
- ‘transparencyFactor’ is lower the closer the original is to the replacement in transparency
- Also need to change backing plate!



Brick image source: Bricklink.com

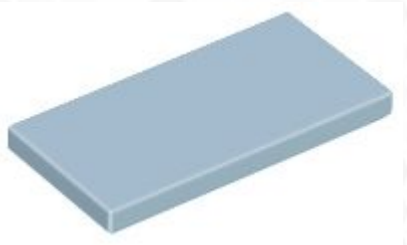
Metallicity

- Much trickier than transparency
- Isn’t stored as numbers in images
- Predict for each pixel with convolutional neural net
- Use same transparency trick afterwards
  - colorDist(original$_{i,j}$,replacement$_{i,j}$) + metallicityFactor

Brick image source: Bricklink.com

Complex Pixels

- What if pixels aren’t points?
- Asymmetrical pixels
- Pixel pieces with prints
- Multi pixel pieces (see ‘efficient tiling’)
- Raised pixels (see ‘3D mosaics’)



Brick image source: Bricklink.com

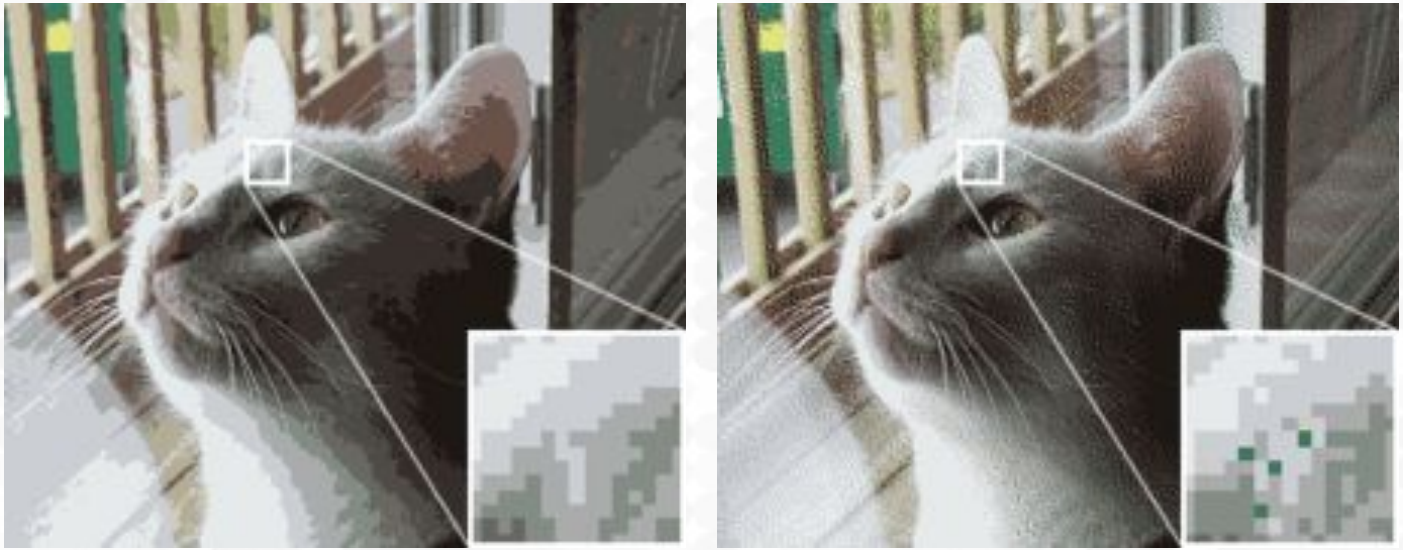
Brick image source: Lego.com

Price optimization

- Some colors can be significantly more expensive
- May still be worth it!
- Quality vs. price tradeoff
- We can use the transparency trick again:
  - colorDist($original_{i,j}$,$replacement_{i,j}$) + k*price($replacement_{i,j}$)
- The higher ‘k’ is, the more we care about price
- This is a very efficient way to account for price

Dithering

- Useful for large outputs



Source: https://en.wikipedia.org/wiki/Dither

The remix problem

- What if we have limited numbers of studs for each color?
- Same goal, different constraints
  - $\sum$ colorDist($original_{i,j}$,$replacement_{i,j}$)
- Much harder to minimize this now

Source: ‘It Doesn't Have to Make Sense’ Album Art, Ingrid Michaelson, 2016

Use cases

- You want to design something with your existing pieces
- You want to design a MOC from an existing set
- You want to design something that’s easy to acquire
- You think it would be cool
- You work for Lego!

Assignment problem/Linear programming

Assignment problem

From Wikipedia, the free encyclopedia

The **assignment problem** is a fundamental combinatorial optimization problem. In its most general form, the problem is as follows:

> The problem instance has a number of *agents* and a number of *tasks*. Any agent can be assigned to perform any task, incurring some *cost* that may vary depending on the agent-task assignment. It is required to perform as many tasks as possible by assigning at most one agent to each task and at most one task to each agent, in such a way that the *total cost* of the assignment is minimized.

Alternatively, describing the problem using graph theory:

> The assignment problem consists of finding, in a weighted bipartite graph, a matching of a given size, in which the sum of weights of the edges is a minimum.

Solution by linear programming [edit]

The assignment problem can be solved by presenting it as a linear program. For convenience we will present the maximization problem. Each edge (i,j), where i is in A and j is in T, has a weight w_{ij}. For each edge (i,j) we have a variable x_{ij}. The variable is 1 if the edge is contained in the matching and 0 otherwise, so we set the domain constraints:

$$0 \leq x_{ij} \leq 1 \text{ for } i,j \in A,T,$$

$$x_{ij} \in \mathbb{Z} \text{ for } i,j \in A,T.$$

The total weight of the matching is: $\sum_{(i,j)\in A\times T} w_{ij}x_{ij}$. The goal is to find a maximum-weight perfect matching.

To guarantee that the variables indeed represent a perfect matching, we add constraints saying that each vertex is adjacent to exactly one edge in the matching, i.e,

$$\sum_{j\in T} x_{ij} = 1 \text{ for } i \in A, \quad \sum_{i\in A} x_{ij} = 1 \text{ for } j \in T,$$

.

All in all we have the following LP:

$$\text{maximize} \sum_{(i,j)\subset A\times T} w_{ij}x_{ij}$$

$$\text{subject to} \sum_{j\in T} x_{ij} = 1 \text{ for } i \in A, \quad \sum_{i\in A} x_{ij} = 1 \text{ for } j \in T$$

$$0 \leq x_{ij} \leq 1 \text{ for } i,j \in A,T,$$

$$x_{ij} \in \mathbb{Z} \text{ for } i,j \in A,T.$$

Solving the remix problem

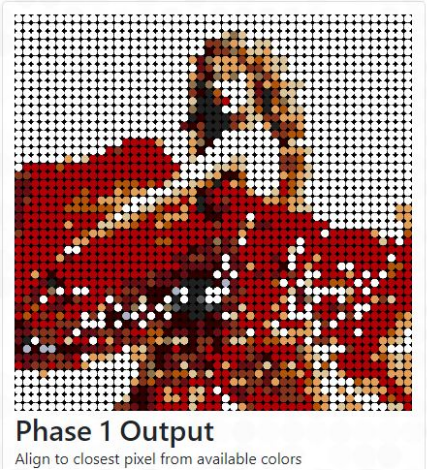
- The remix problem reduces to the assignment problem in polynomial time
  - Instead of agents and tasks, we have studs and pixels
- The assignment problem similarly reduces to linear programming
- Many libraries exist for solving linear programs
- The catch: it’s slow, and memory usage can be bad
  - Can get around this by reducing problem size, but this can be unreliable

Solving the remix problem (faster)

- To get around, this, we use a much faster 2 phase approximation algorithm
- The algorithm first fills in the ‘best matches’ for each available color as best as it can
- Then, it fills in the gaps using a similarly greedy approach
- It’s not perfect, but in practice, it works well under some reasonable assumptions:
  - Ex: If a user has at least one stud of a given color, then they can acquire more in the worst case (or remove this color entirely)

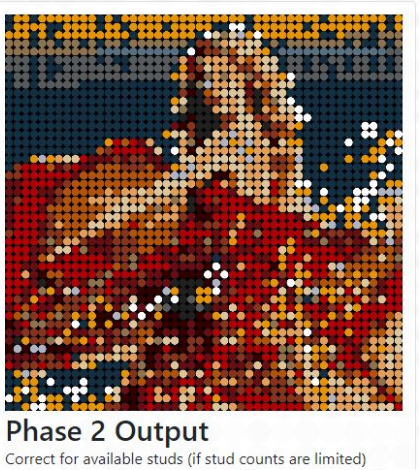
Easier debugging

- A 2 phase approach makes issues easier for humans to fix
- Ex: Trying to make our earlier example from the iron man set



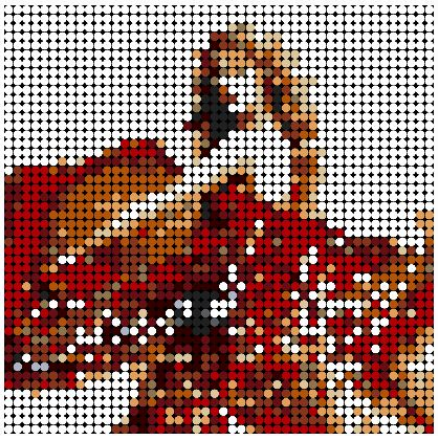
Phase 1 Output

Align to closest pixel from available colors



Phase 2 Output

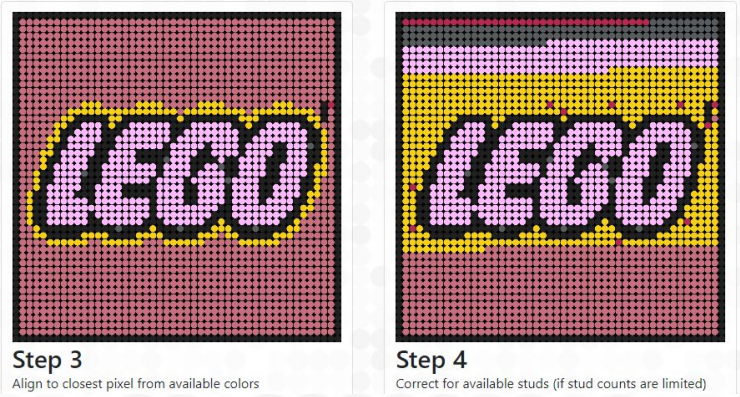
Correct for available studs (if stud counts are limited)



Easy to see that biggest problem is missing white studs

Segmented backgrounds

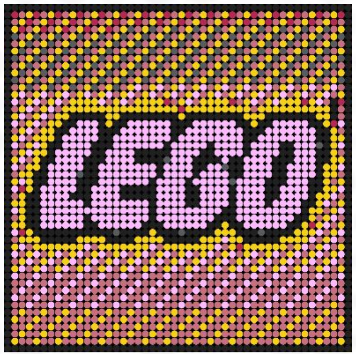
- For backgrounds and other monochromatic patches, we may fall short with color counts
  - Ex: previous slide
- May lead to weird/unpredictable segmentation



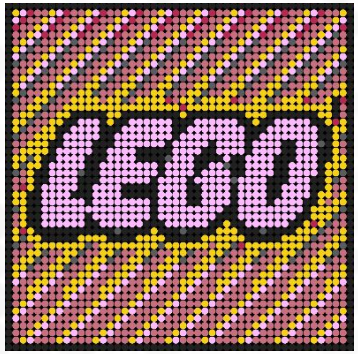
- We could detect, then manually process these
  - Can be difficult, may not be reliable

Pseudo ‘dithering’

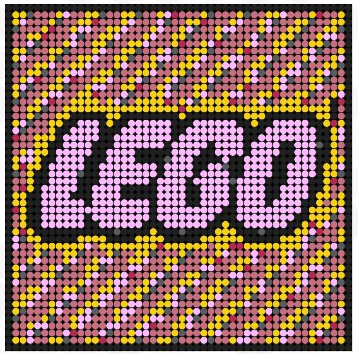
- Reuse our transparency color distance strategy
- This time, the extra factor is very small
  - Tiebreaking only
- colorDist(original$_{i,j}$,replacement$_{i,j}$) + ϵ((i+j) mod k)
  - Where ϵ and k are small, and k is an integer
  - Can also use cascading mod
  - Can also add even smaller noise after



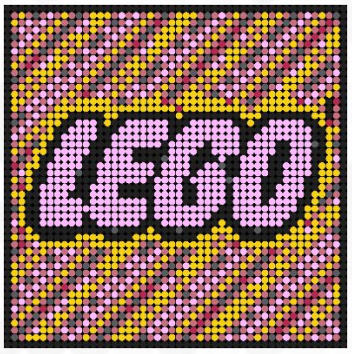
Mod 3



Mod 5



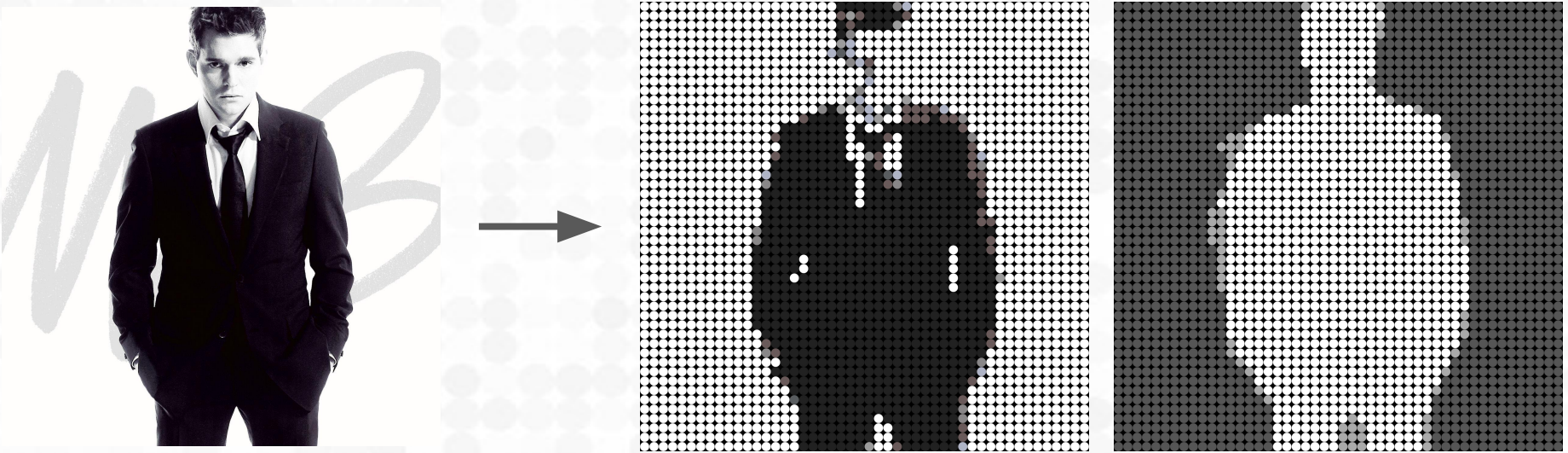
Noisy Mod 5



Cascading Noisy Mod

3D mosaics

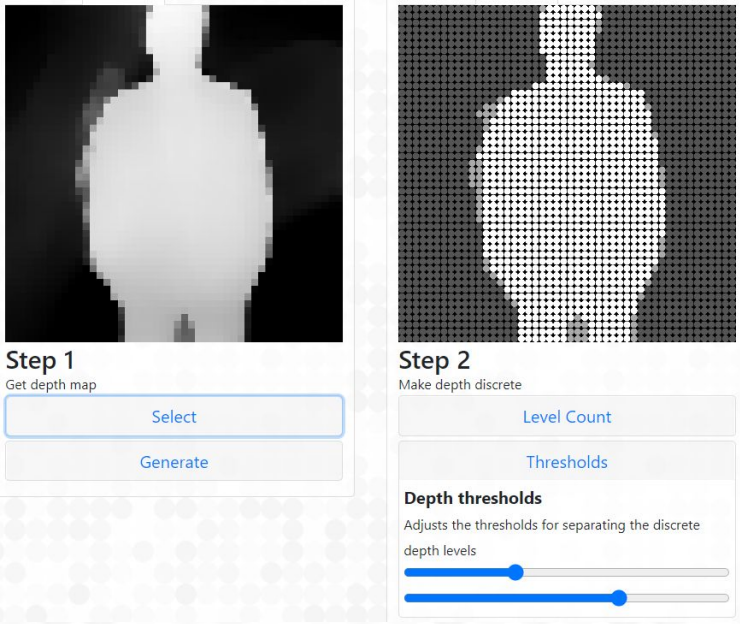
- Designing mosaics with depth
- Adding plates under certain pixels to raise them up



Source: `It's Time` Album Art,
Michael Bublé, 2005

Depth maps

- Usually used for portrait mode, etc.
- Usually contain a depth value from 0-255 per pixel
- Reduced from 256 to 2-5 levels for 3D Lego mosaics

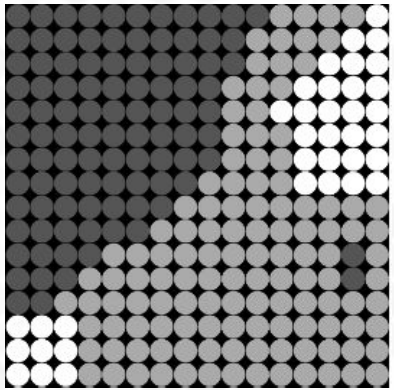


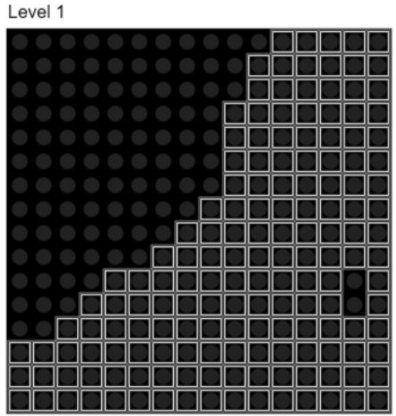
Inferring depth maps

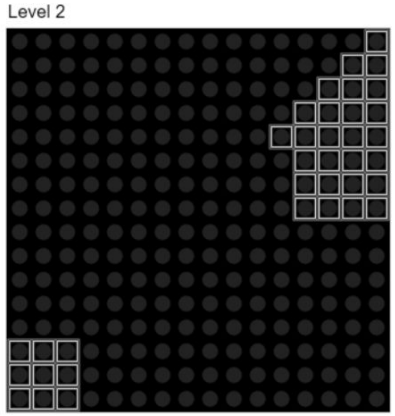
- What if a depth map for the image isn’t provided?
- We can infer one using monocular depth estimation
- Many algorithms for this exist
  - A bunch of these are neural network based
  - The one I use in production is MiDaS
    - Ranftl, René, Katrin Lasinger, David Hafner, Konrad Schindler, and Vladlen Koltun. "Towards robust monocular depth estimation: Mixing datasets for zero-shot cross-dataset transfer." (2020). *IEEE Transactions on Pattern Analysis and Machine Intelligence*

Backing plates

- Using plates in order to raise up the pixels to create depth
- Generally one layer of plates per layer of depth
- Can also fill in more space with bricks (for >3 layers)
  - Can be annoying in many cases, affect reusability
- Best to avoid plates with large areas
  - Difficult to remove

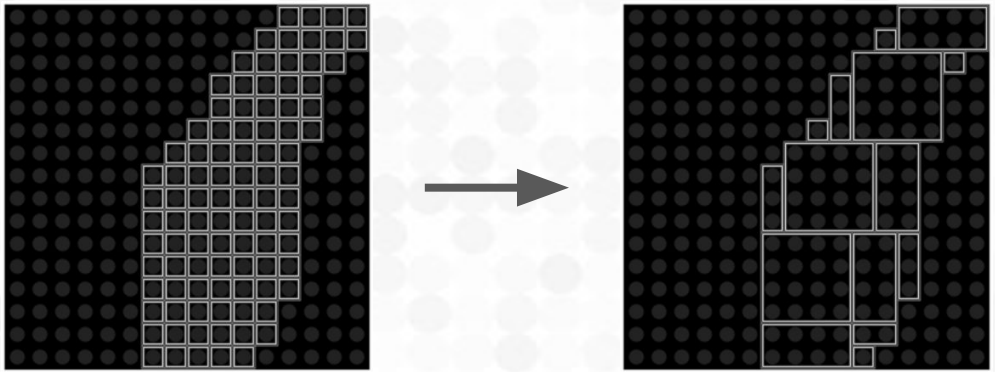






Efficient Tiling

- Want to minimize the number of plates used
- Given a set of possible plate dimensions
  - Might want to exclude large plates
- Solution: Sort by area, then fill from edges (‘stones + sand’)
  - Not perfect, but optimal in many cases
  - Guarantees adjacent small plates can’t be ‘merged’
  - Very fast



Pixelization

- Pixelization vs. pixelation vs. image retargeting
- Low resolution displays vs. pixel art and Lego mosaics
- ‘Quality’ can be subjective
- Could help deal with blurred edges in Lego mosaics
  - ‘anti-anti-aliasing’
- Not as relevant for larger resolutions

Source: ‘Red’ Album Art, Taylor Swift, 2012

Voting Majority

- Relatively fast method for fixing blurred edges
- May mess up non edge pixels
- Requires threshold tuning (for majority definition)
- Requires grouping before voting (remove small variations)
  - Might make things expensive
- Example: (30% threshold)

40% ‘Pink’
15% ‘Green’
20% ‘Purple’
25% ‘White’

Choose Pink

25% ‘Pink’
25% ‘Green’
25% ‘Purple’
25% ‘White’

No winner;
Just average colors, as before

Note: each color here is a group around a hex code

Superpixels

- Gerstner, Timothy, et al. "Pixelated image abstraction." *NPAR@ Expressive*. 2012.
- Designed for pixel art, so good for Lego mosaics
- Can be expensive (many iterations usually used)

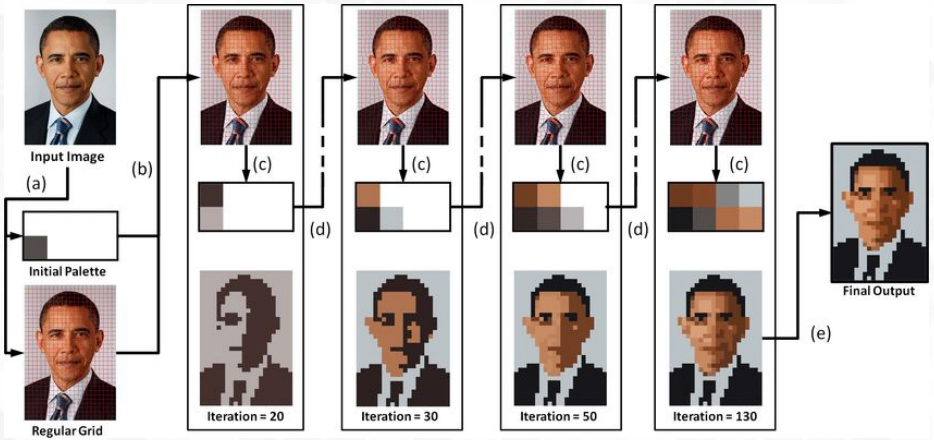


Image source: Same as citation

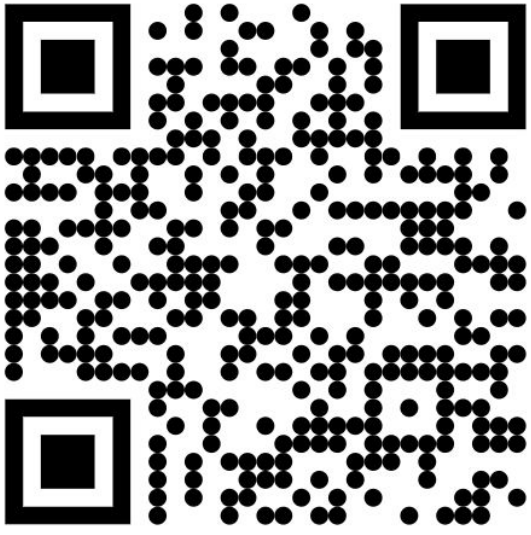
Seam Carving

- Avidan, Shai, and Ariel Shamir. "Seam carving for content-aware image resizing." *ACM SIGGRAPH 2007 papers*. 2007. 10-es.
- Can be expensive depending on input size
- Likely need to use a variation to target the remaining edges
- Could help deal with segmented backgrounds

Image source: Same as citation

Code Links

- github.com/debkbanerji/lego-art-remix
- See citations for non Lego art specific algorithm code
- See lego-art-remix.debkbanerji.com if you don’t care about code and just want to use a Lego mosaic tool



Q & A

- If you have a feature request for Lego Art Remix specifically, you can also open an issue on github.com/debkbanerji/lego-art-remix/issues