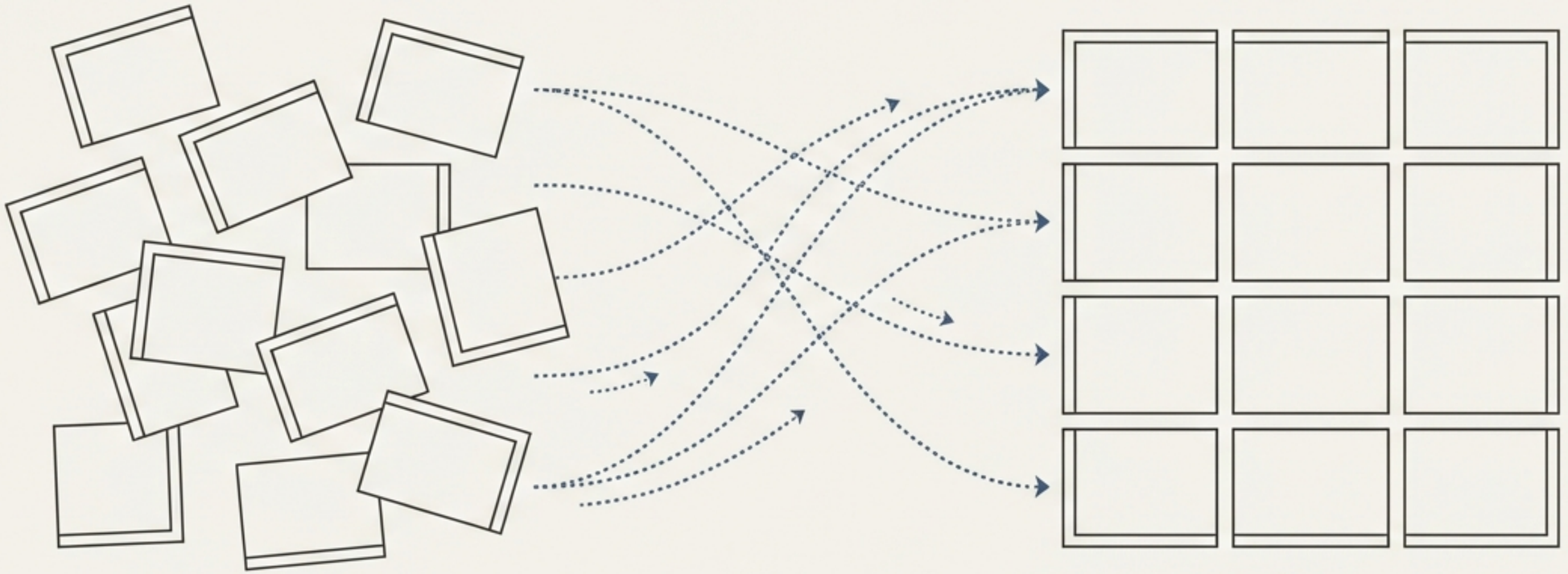
The Image Puzzle Solver Toolkit: A Technical Deep-Dive

An automated pipeline for reconstructing fragmented images.



This presentation details the architecture and algorithms behind a toolkit designed to automatically reassemble a set of rectangular puzzle tiles, outputting both a static reconstruction (PNG) and an animated assembly (MP4).

From Scattered Fragments to a Coherent Whole

INPUT

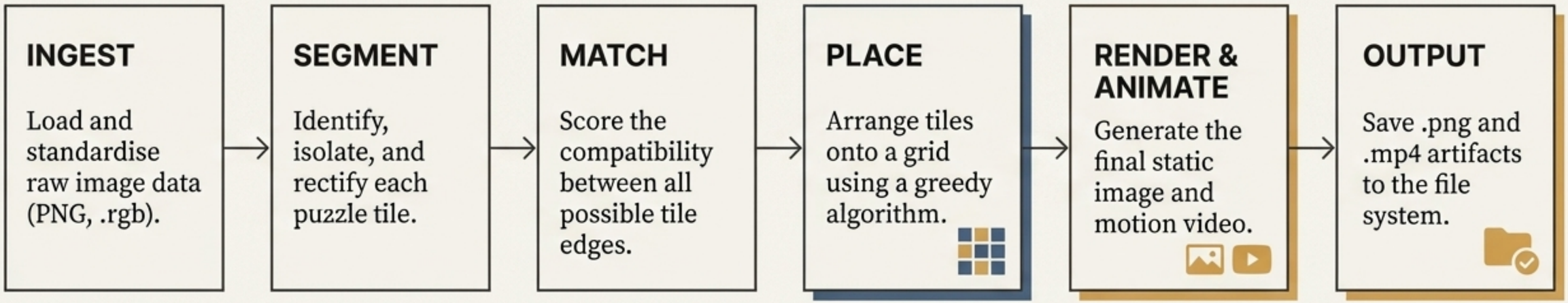
OUTPUTS

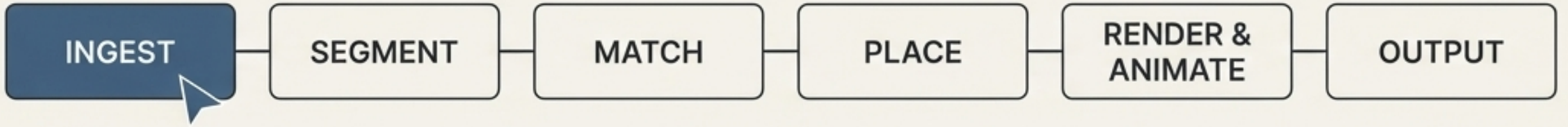
Solved PNG

Assembly MP4

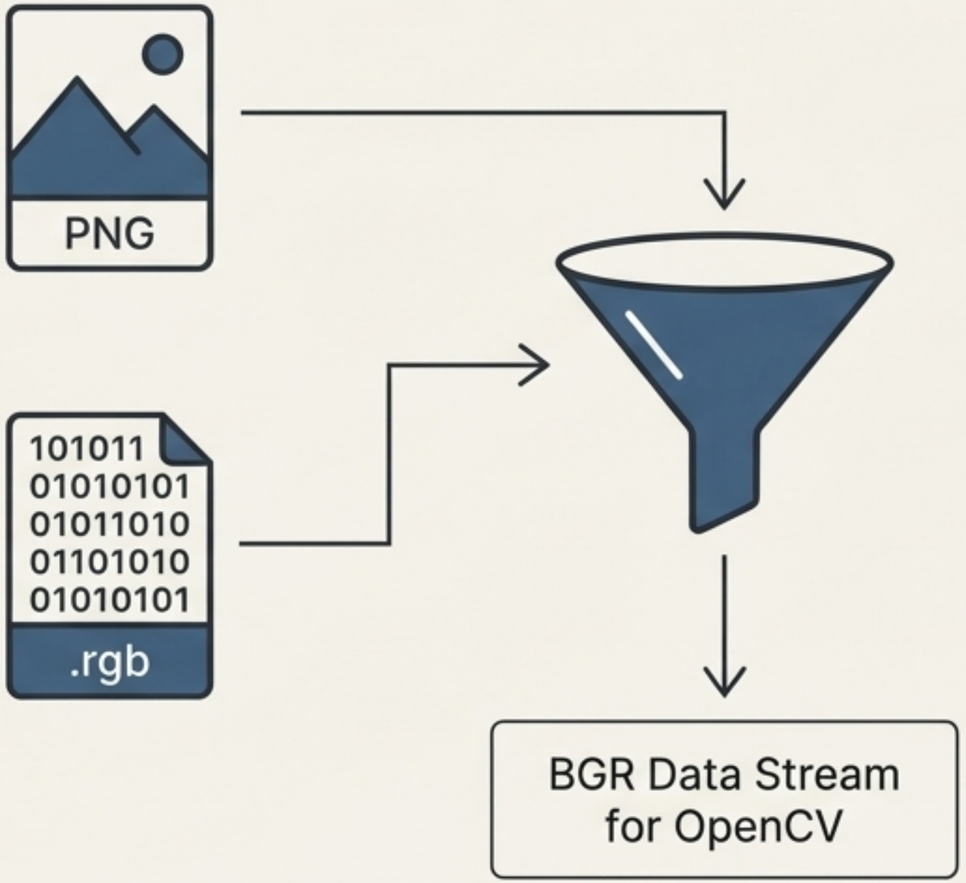
The system ingests a composite image of puzzle tiles and produces two key outputs: a solved PNG grid and a dynamic MP4 animation of the assembly process.

The Core Pipeline: A Step-by-Step Assembly Line





Stage 1: Ingesting and Standardising Input Frames



Function: `load_frame()` in `mosaic_engine/ingest.py`

```
load_frame() in mosaic_engine/ingest.py
```

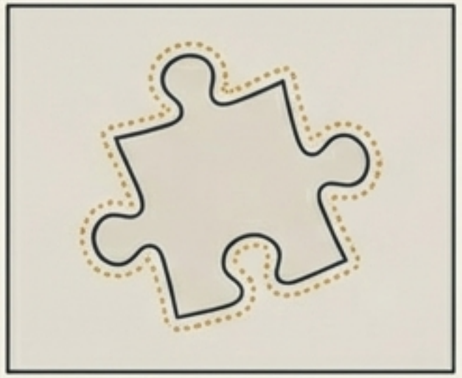
Supported Formats:

- **PNG:** Read directly via `cv2.imread`.
- **Raw .rgb:** Interpreted as a `3×N×N` linear byte payload. Dimension `N` is auto-detected via `_deduce_rgb_dimension`, with a fallback to 800.

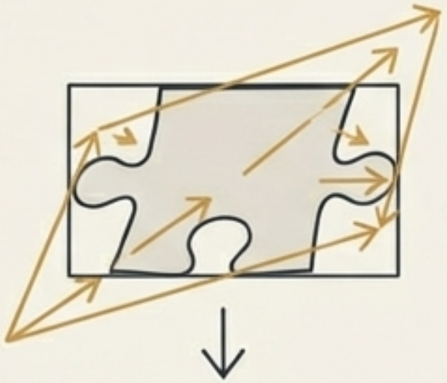
Data Standardisation: All inputs are converted from RGB to BGR for internal consistency with OpenCV.

Error Handling: The system raises exceptions on missing or corrupt input files, validated by `` `_iter_inputs` ``.

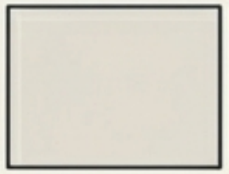
Stage 2: Segmenting and Rectifying Puzzle Tiles



1. Detect Contour



2. Apply Perspective Transform

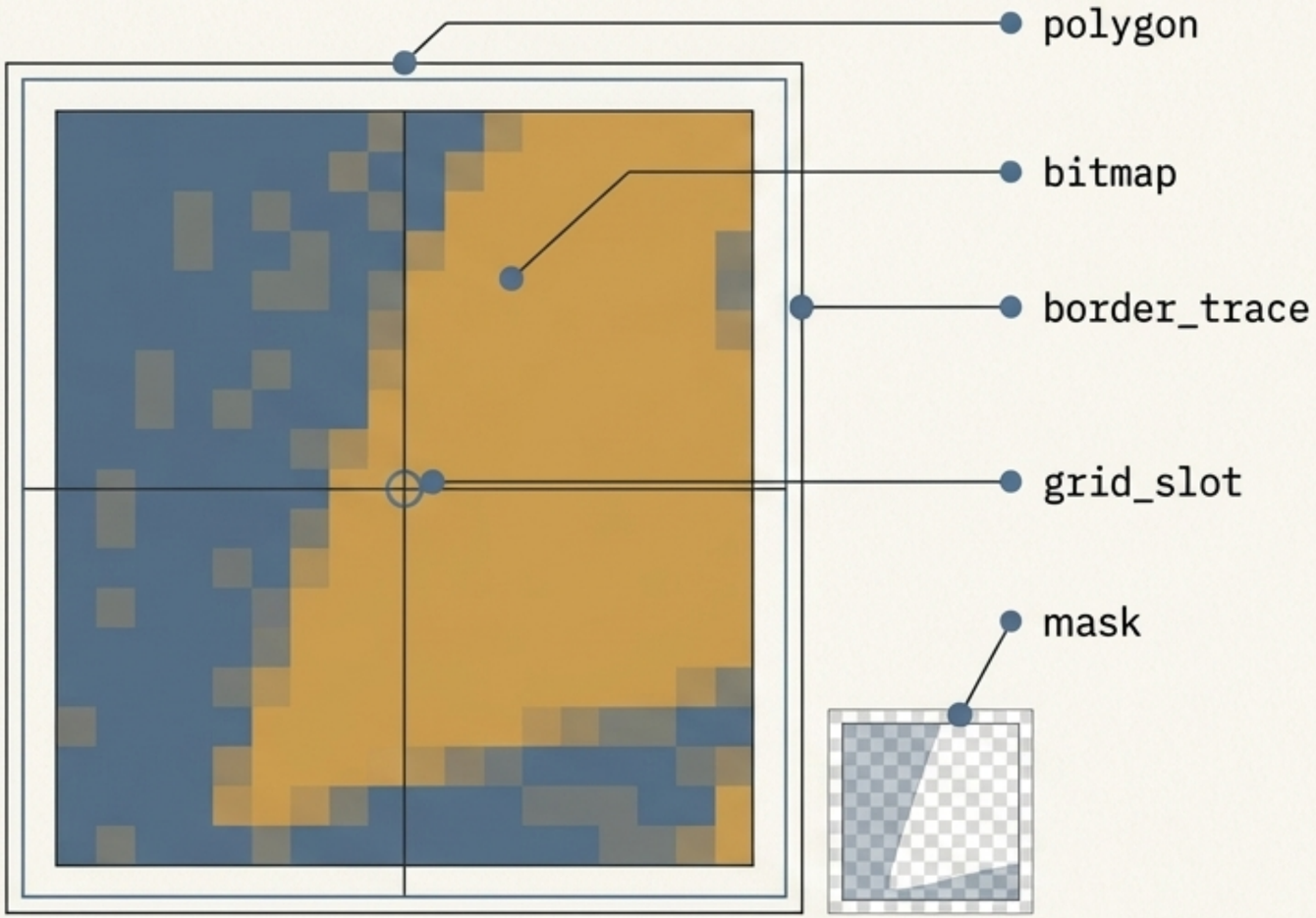


3. Isolate & Rectify

Segmentation Workflow (`extract_fragments()`):

1. **Locate Shapes (`_locate_shapes`):** Convert to grayscale, apply a binary threshold, and use `cv2.findContours` to detect external boundaries of all shapes.
2. **Isolate Quadrilaterals:** Filter contours to keep only four-cornered shapes.
3. **Rectify Orientation (`_realign_quadrilateral`):** For each quadrilateral, a perspective transform is applied to normalise its rotation, effectively 'straightening' the tile.
4. **Trace Edges (`_trace_outline`):** Capture the edge pixel data along the original contour, relative to the newly cropped tile.

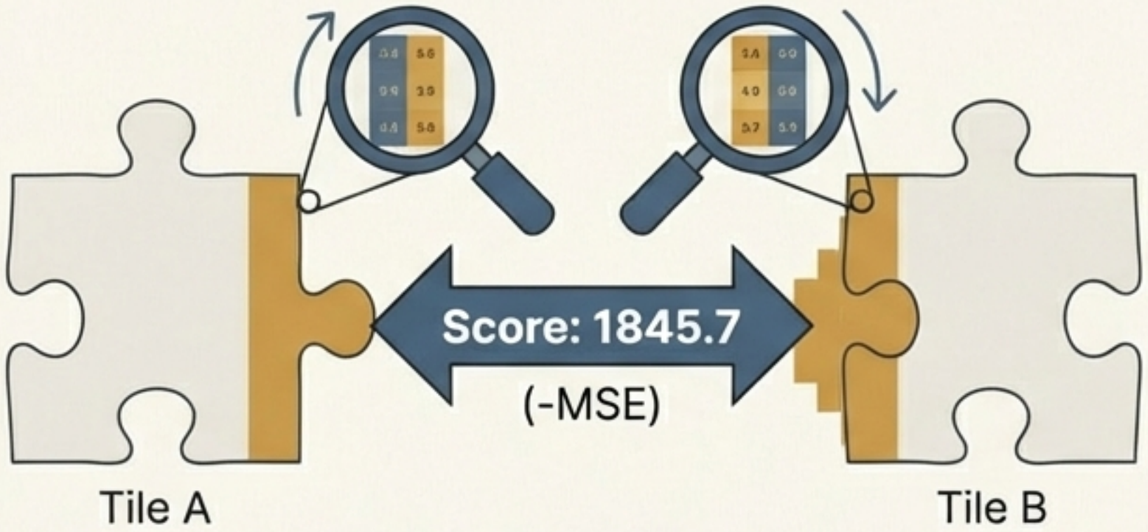
Anatomy of a Tile: The `piece_fragment` Data Structure

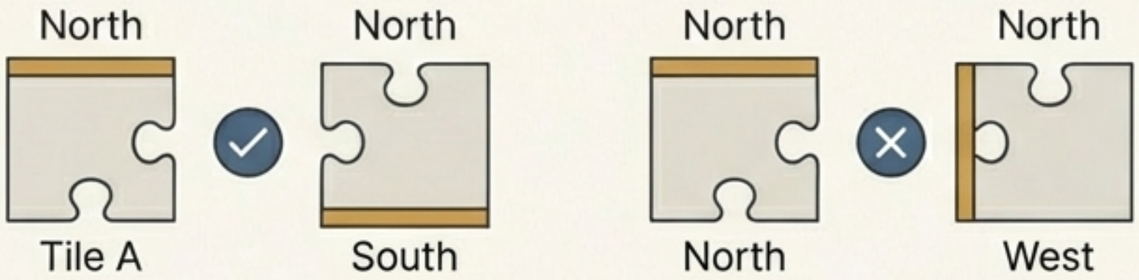


Key Attributes

- `polygon` : Normalised coordinates of the rectified tile.
- `border_trace` : Sampled edge pixel data used for matching.
- `bitmap` : The actual pixel data of the cropped tile.
- `spin_hint` : Initial rotation detected during segmentation.
- `target_box` : The final dimensions for the tile in the output grid.
- `grid_slot` : The (row, col) position in the final puzzle.
- `start/final anchors` : Positional data for animation.
- `mask` : A binary mask for clean compositing, created via `ensure_mask()` .

Stage 3: Scoring Edge Similarity with MSE

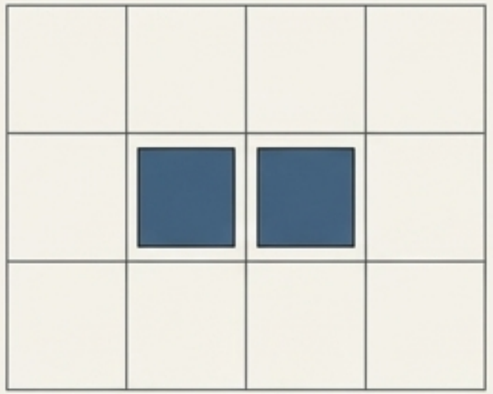




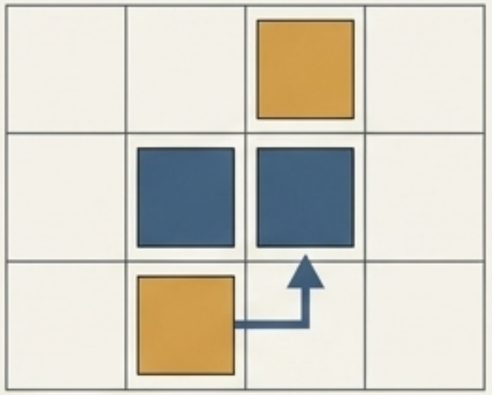
Matching Algorithm (`compute_pairings()`):

- **Goal:** For every possible pair of tiles, find the best complementary edge fit (e.g., the right edge of Tile A vs. the left edge of Tile B).
- **Mechanism (`score_edge_alignment`):**
  - Edge vectors (top, bottom, left, right) are extracted as `float32` arrays.
  - Similarity is calculated using **Mean Squared Error (MSE)** between the pixel values of two opposing edges.
  - The final score is a negative MSE, so higher values indicate a better match.
- **Key Constraint:** The algorithm only compares opposing edges (north/south, east/west). It does not perform a rotational search (e.g., testing a north edge against a west edge).

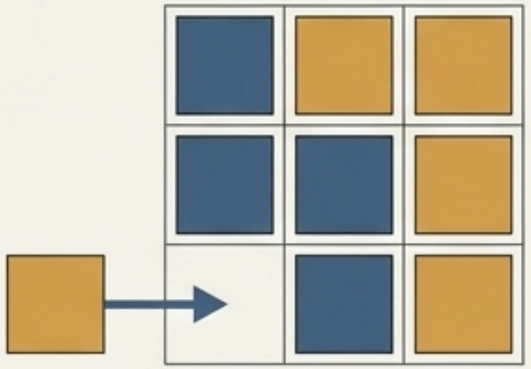
Stage 4: A Greedy Approach to Grid Placement



1. Seed with best-scoring pair.



2. Iteratively add best adjacent tile.

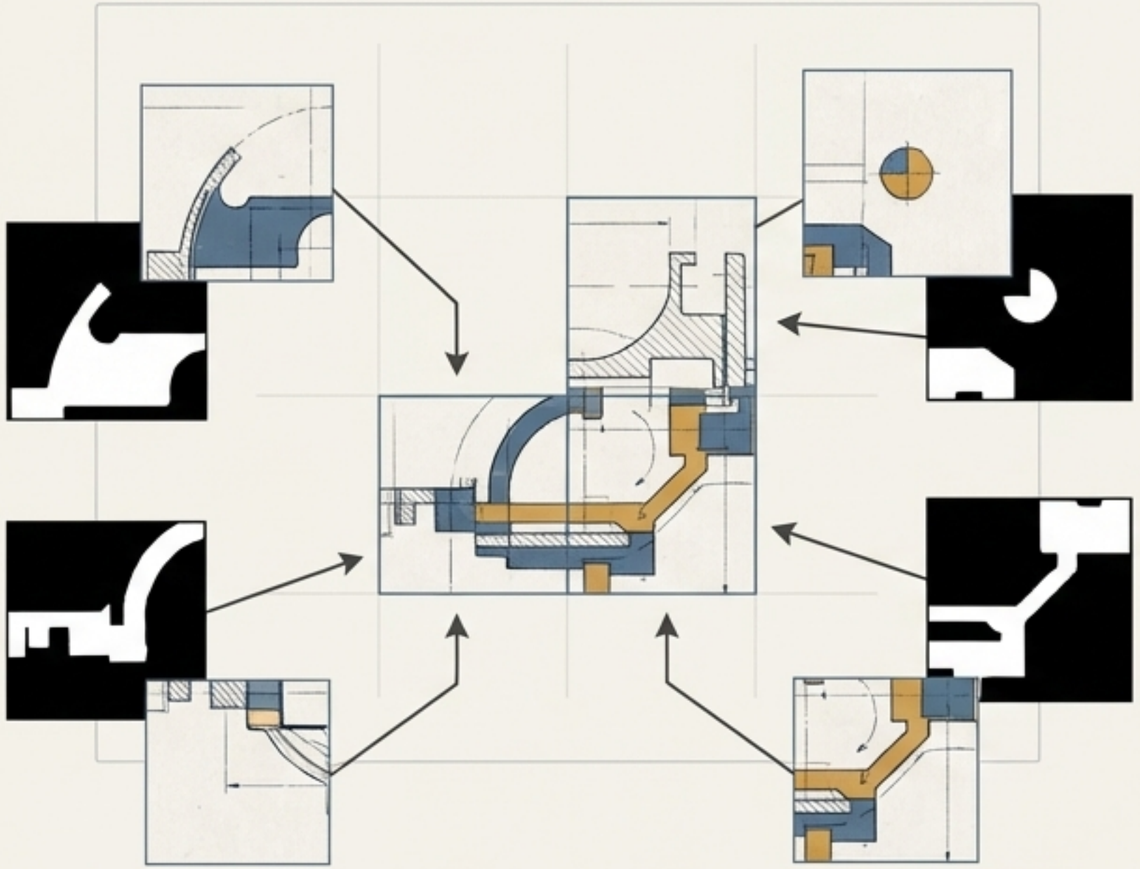


3. Complete grid with fallbacks.

Placement Strategy

1. **Seeding:** The grid is initiated with the highest-scoring tile pair found in the matching stage.
2. **Iterative Placement:** The algorithm iteratively finds the best-scoring available tile that can be placed in an empty slot adjacent to an already-placed tile.
3. **Fallback:** If a tile has no viable high-scoring neighbour, it is placed adjacent to the very first seed tile to ensure it's included.
4. **Grid Finalisation:** The final grid of tiles is normalised to a (0,0) origin, and per-column/row dimensions are calculated to accommodate varying tile sizes. The `final_spin` for all tiles is set to 0.

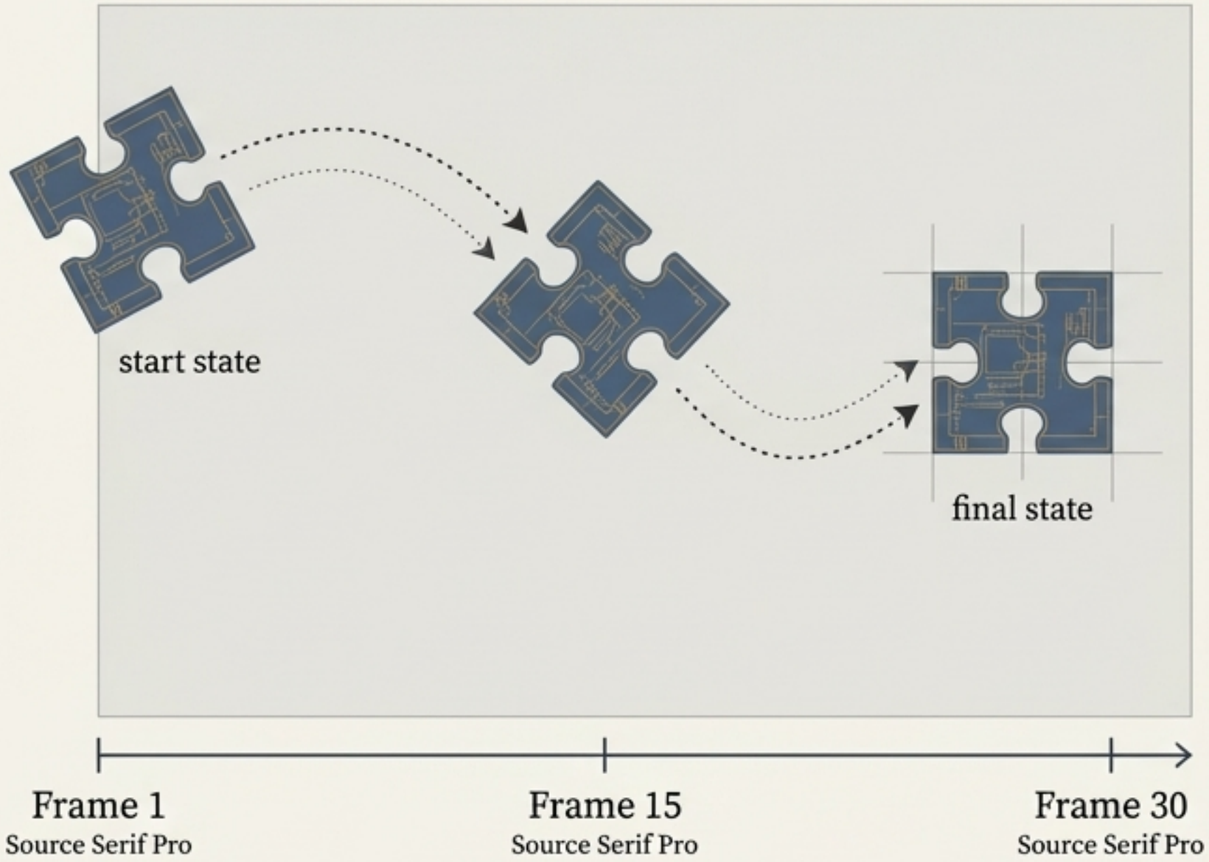
Stage 5: Rendering the Solved PNG Canvas



Process (`compose_grid_image()`):

- **Canvas Creation:** A blank canvas is created, with cell sizes determined by the maximum `target_box` dimensions of all tiles.
- **Tile Resizing:** Each tile's bitmap is resized to match its `target_box` dimensions, ensuring consistency.
- **Mask-Aware Compositing:** Tiles are pasted onto the canvas using their binary masks. This prevents gaps and ensures clean boundaries, even with non-perfectly rectangular rectified shapes.
- **Final Output:** The completed canvas is saved as the final `._solution_{index}.png` file.

Stage 5 (cont.): Animating the Assembly Process



Process (`emit_animation()`):

- **Canvas:** Prepares an 800x800 canvas (or uses source dimensions if available).
- **Per-Frame Logic:** For each of the specified number of frames:
  - Calculates the interpolated position and rotation for every tile between its start and final anchors/spins.
  - Rotates the tile bitmap using `_rotate_fragment()`.
  - Alpha-blends the tile onto the current frame.
- **Video Encoding:** Frames are written to an MP4 file using `cv2.VideoWriter` with the `mp4v` codec at 15 FPS.

Putting It All Together: The `puzzle_runner.py`

Structure: `python puzzle_runner.py [inputs...] [--input-root] [--output-root] [--frames]`

Key Arguments:

- inputs: Positional arguments for one or more input files.
- --input-root: Path to input directory (default: `inputs/`).
- --output-root: Path for output files (default: `outputs/`).
- --frames: Number of frames for the MP4 animation (default: `30`).

Example Usage:

```
$ python puzzle_runner.py test_puzzle.png --frames 60
```

Expected Output Files:

- outputs/test_puzzle_solution_0.png
- outputs/test_puzzle_solution_0.mp4

A Principled Design: Known Assumptions and Limitations

Core Assumptions

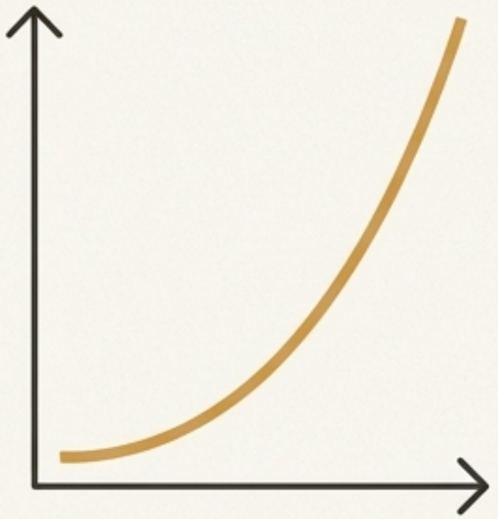
- **Shape:** Input pieces are four-sided (quadrilaterals). Irregular silhouettes are ignored.
- **Clarity:** A clean binary separation between tiles and background is possible for effective contour detection.

Algorithmic Limitations

- **Placement:** The greedy algorithm may yield sub-optimal global solutions.
- **Matching:** MSE-based scoring is sensitive to lighting variations. No histogram balancing or feature descriptors are used.
- **State:** Solution metadata (tile positions, rotations) is not persisted after a run.

Performance Profile and Complexity

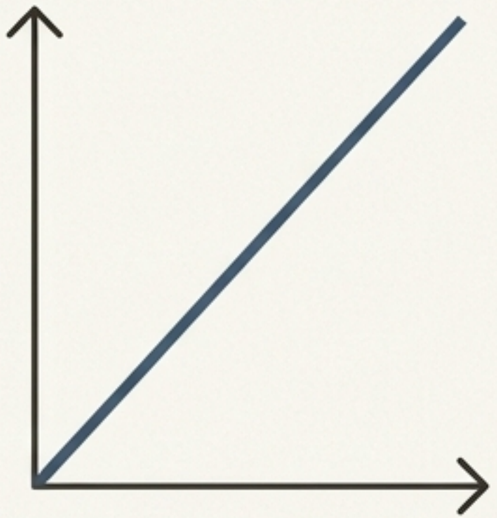
Edge Matching



Complexity: $O(n^2 \times \text{orientations})$ where n n is the number of tiles.

Analysis: This quadratic scaling is acceptable for puzzles with a small number of pieces, but would become a bottleneck for very large puzzles. `cv2.resize` is used for edge alignment.

Animation Rendering



Cost Drivers: Performance scales linearly with `frame_count` × `tile_count` × `tile_area`.

Analysis: The primary cost is the per-frame rotation and blending of each tile's bitmap.

The Path Forward: Suggested Enhancements

Improve Matching Robustness

- Implement rotation search (0/90/180/270 degrees) during edge comparison.
- Introduce more advanced edge descriptors (e.g., color normalization, gradient-based features).

Refine Placement Strategy

- Add a global refinement step post-placement, such as a simple swap-based hill-climbing algorithm.

Enhance Reproducibility

- Persist solution metadata (grid positions, final angles) to a JSON file alongside media outputs.

Increase Flexibility

- Parameterize layout controls like margins and cell padding via CLI arguments.

From Chaos to Order: The Automated Solution

The Image Puzzle Solver pipeline successfully transforms fragmented inputs into a coherent whole, demonstrating an effective, end-to-end automated reconstruction system.