THOMAS MORE

Internship at Smart NV

Realisation document

Marin Janushaj
Student Bachelor Applied Computer Science

2025 - 2026

Table of Contents

1. Introduction

1.1. Project summary

This realisation paper describes my internship work at Smart NV from February to May 2026. The assignment investigated how a physical SmartGames puzzle, IQ Puzzler Pro, could be supported by a digital companion that recognises a real board, reconstructs the logical puzzle state and helps the player continue with validation, hints and solving.

The realised work has three main parts. The front-board system recognises the rectangular 5 by 11 side of the puzzle. The back-board system recognises the diamond-shaped side and translates it into the same 55-cell logical idea. The Remix prototype extends the concept by generating new 55-cell shapes, validating them with the solver and exporting printable material, including STL output for a physical prototype.

The project is best understood as a complete chain rather than as one isolated model. A phone photo is only the starting point. The application must locate the board, recognise the pieces, convert visual evidence into exact puzzle cells, check whether the state is still solvable and then present the result in a way that a player can correct or continue. This is why the work combines machine learning, computer vision, exact-cover solving and user-interface design.

The first figure gives an overview of the delivered application structure before the report goes into the separate front-board, back-board and Remix workflows.

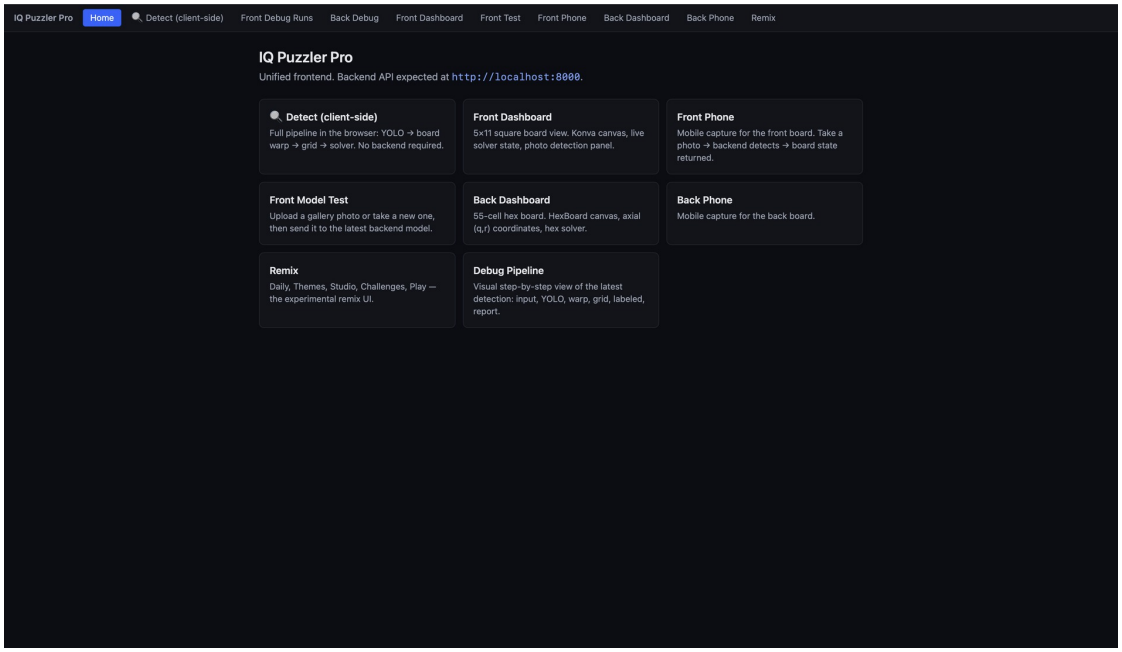

Figure 1. Unified application shell with routes for detection, front debug, back debug, dashboards, phone clients and Remix.

Explanation: This first screen shows the project as one product instead of unrelated experiments. The same app shell gives access to capture, debugging, manual correction and Remix, which made testing and handover easier.

1.2. Contents and document map

The document follows the school template structure. The introduction explains the context, scope and game rules. The analysis chapter explains why certain technical choices were made. The realisation chapter describes the front board, back board, Remix prototype and delivery package. A separate evaluation chapter summarises testing, datasets, results and remaining limitations. The conclusion reflects on the internship and future work.

1.3. Key terms for non-technical readers

Term	Explanation
Board state	A 55-character representation of which puzzle piece occupies each well.
YOLO segmentation	A computer-vision model that returns masks for objects instead of only rectangular boxes.
Mask	A pixel-level outline of a detected board or puzzle piece.
Perspective warp	The correction that turns an angled phone photo into a flat board view.
Exact-cover solver	A search algorithm that checks whether all cells can be covered exactly once by the puzzle pieces.
Synthetic data	Training images generated in Blender with automatic labels instead of being photographed and annotated by hand.
Fine-tuning	Additional model training on real project photos after a base or synthetic training phase.
STL	A 3D model file format used to export printable Remix boards.

1.4. Internship context

Smart NV develops physical logic puzzles under the SmartGames brand. The internship was therefore not only a software exercise; it had to respect a physical toy, its pieces, its play experience and the need for children to keep solving rather than immediately receiving an answer.

I completed the internship together with Ibrahim Afkir and Abdul Salam Aldabik. We each had our own work, but we helped each other when one of us was stuck. Wouter Caeldries supervised the internship in a calm and supportive way. He gave feedback and direction without unnecessary pressure, which made it easier to ask for help during difficult technical weeks.

1.4.1. Game, board and rules

IQ Puzzler Pro contains twelve fixed pieces made of connected balls. The front side is a 5 by 11 rectangular board. The back side is a 55-cell diamond-like board of circular wells. A valid challenge fills the board without overlaps and without leaving holes. For the software, the important point is that every photo must eventually become the same kind of logical object: a validated 55-cell board state.

1.4.2. Scope and boundaries

In scope	Out of scope or provided
Frontend and backend application logic for front-board and back-board workflows	The physical IQ Puzzler Pro product and the base play concept
Computer-vision pipelines, model integration, debug screens and mapping logic	Confidential company material not needed to explain the internship
Synthetic-data scripts, Colab preparation and model installation helpers	Final commercial product decisions after the internship
Remix prototype, solver logic, booklet/export flow and STL proof of concept	A separate confidential labelling/training side task, which is not discussed by name

1.4.3. Working methodology

I worked iteratively. A typical cycle started with a failing photo or a usability problem, followed by an analysis of whether the problem came from the model, the board geometry, the solver or the interface. I then changed one part, ran the app again, saved debug output and compared the result. This was important because a visually good detection could still produce a logically impossible board.

Summary
The internship scope was broader than a single model. The final product direction combined phone capture, computer vision, deterministic puzzle logic, manual correction, hints, solving and a separate Remix generator.

2. Analysis

2.1. Problem decomposition

A human can understand the board by looking at colour, shape and context at the same time. The software had to split this into separate problems: capture a usable photo, locate the board, recognise pieces, map them to exact cells, reject impossible states and still give the player a usable interface when the model is uncertain.

The runtime flow was designed as separate checkpoints instead of one black-box prediction. The phone sends the photo to the backend, YOLO segmentation produces board and piece evidence, OpenCV corrects the board perspective, the mapper creates a 55-cell state and the solver decides whether that state can still be completed.

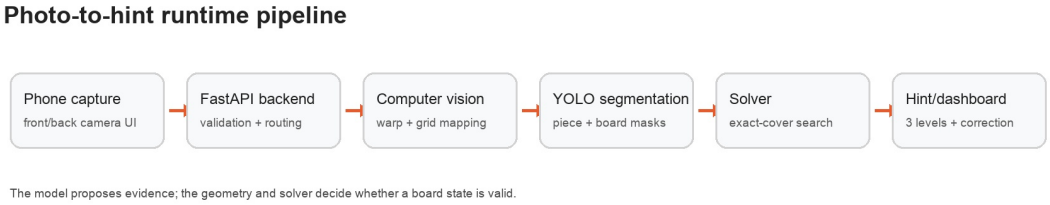

Figure 2. Runtime pipeline from phone photo to validated board state, hint or review state.

Explanation: The pipeline shows why the model is only one part of the solution. The app must convert visual evidence into exact puzzle cells, then use the solver to decide whether the resulting state is trustworthy.

2.2. Technology selection

I compared simple colour classification, cloud vision APIs, rectangular object detection and segmentation. Colour classification was attractive because it was simple, but it failed when lighting, glare and similar colours changed. Cloud vision was rejected because the domain was narrow, repeated calls would add dependency and latency, and local inference was more appropriate for a toy companion. Rectangular boxes helped during early testing but were too imprecise for connected pieces. Segmentation was selected because masks preserve the shape evidence needed by the mapping logic.

For the model architecture, I selected YOLO v26 nano segmentation rather than a larger YOLO variant. The companion workflow is driven by phone photos, so the model needed to stay small, load quickly and keep the detection loop responsive during testing and future phone or edge deployment. A larger model could sometimes improve accuracy, but it would increase memory use and startup time, which did not match the product direction of a lightweight puzzle companion.

Option	Why it was rejected or accepted
Colour sampling	Rejected because phone lighting, reflections and similar colours made it unreliable.
Cloud vision API	Rejected because the project needed narrow-domain, repeatable, local model behaviour.

Bounding boxes	Partly useful, but rejected as the main method because boxes do not describe connected piece shape.
YOLO segmentation	Accepted because masks give board and piece shape evidence while remaining fast enough for iteration.
Deterministic solver	Accepted because it verifies whether the detected state is logically playable.

2.3. Synthetic-data decision

The largest data problem was the number of labelled examples. Manually annotating thousands of puzzle boards would have consumed too much of the internship. I therefore invested time in Blender scripts that generated labelled synthetic images. This was a risk because Blender was new to me, but it created repeatable data and helped the model see many board combinations that would be difficult to photograph manually.

2.4. Architecture principles

- Treat model output as visual evidence, not final truth.
- Keep board geometry, solver logic and UI correction separate.
- Save intermediate debug output so wrong states can be explained.
- Allow manual correction because a camera pipeline will never be perfect in every real environment.
- Keep Remix isolated so experimental generation cannot break the main detection workflow.

The architecture separates capture clients, FastAPI routes, computer-vision helpers, game solvers and dashboards. That separation was important because a failure could then be traced to the model, the board warp, the cell mapper, the solver or the user interface without mixing all problems together.

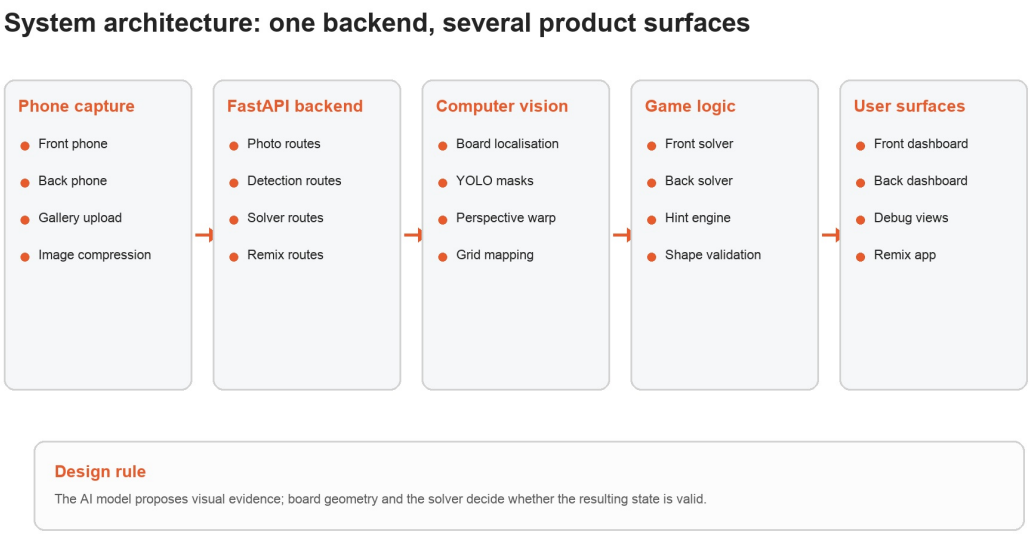

Figure 3. System architecture across capture, backend, computer vision, game logic and user-facing screens.

Explanation: This diagram explains the main design: phone and dashboard screens call a backend; the backend combines model output with board geometry and solver logic; the frontend displays either a playable state or a review state.

This matrix shows that the project is not three unrelated demos. The same core representation appears everywhere: physical evidence becomes exact cells, exact cells are checked by solver logic and the interface exposes the result in a form the player can review or continue.

Feature coverage across the three IQ Puzzler systems

Functionality	Front board	Back board	Remix
Photo capture	Phone upload, compression, gallery test	Dedicated back capture and review state	Not photo-based
Detection	YOLO masks + rectangular warp	YOLO masks + diamond/square mapping	Generated masks, no camera
Manual correction	Shape buttons, drag placement, rotate, remove	Same interaction model on the back board	Paint/edit custom shape
Solver	5 by 11 exact-cover solver	55-cell back-board solver	Mask-aware arbitrary-shape solver
Hints	Progressive piece/area/exact hints	Progressive hints after review/validate	Hints during generated puzzles
Debugging	Saved input, masks, warp, grid, final cells	Original, warped, grid overlay, review flags	Shape validation and solve counts
Exports	Board state and detection artefacts	Mapped state and debug report	Booklet, SVG, STL and share text

Figure 4. Feature coverage across the front board, back board and Remix prototype.

Explanation: The matrix shows that the three product surfaces share core ideas but have different purposes. Front and back are recognition workflows, while Remix is a generation and export workflow.

Summary
The selected solution is a hybrid system. AI recognises visual evidence, deterministic geometry translates that evidence into cells, and the solver validates whether the state is still a real puzzle.

3. Realisation of the internship projects

3.1. Front-board realisation

The front board was the first complete product flow. A user can take or upload a photo, review the processed result, apply it to the digital board, correct mistakes, validate the state, request hints and solve the board. The most important technical challenge was mapping soft visual masks onto exact puzzle cells.

The front board also became the reference implementation for the rest of the project. It forced me to define how pieces are represented in software, how rotations are generated, how collisions are rejected and how a partly filled board is passed to the solver. Once this foundation worked, the same ideas could be reused for the back board and later for Remix, even though their board shapes are different.

The front capture flow is implemented as a phone client and backend pipeline rather than as browser-only image processing. The React phone screen collects the image, shows an upload/progress state and sends it to the detection endpoint. The backend then performs the expensive and inspectable work: segmentation, board localisation, perspective correction, grid mapping and debug-output writing. This design made the system easier to test because each failed photo left evidence on disk.

The front phone client exists so testing can happen from the same kind of input a real user would provide. It supports camera or gallery input, then sends the selected image to the backend instead of running the model in the browser, which keeps inference, model files and debug artefacts in one controlled place.

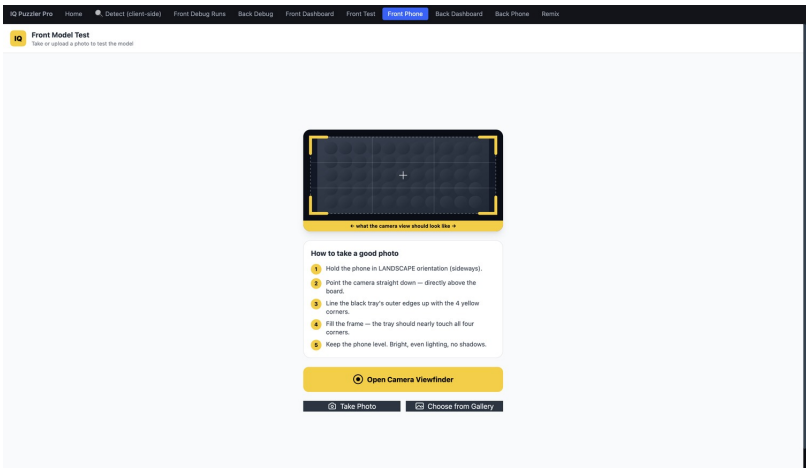

Figure 5. Front phone/gallery capture screen used to supply a real image for detection.

Explanation: The capture screen made testing easier because photos could come from the phone camera or gallery. This helped compare many real examples without changing backend code.

The review step is small but technically important because bad framing creates most downstream errors. By letting the user confirm the photo before upload, the app reduces failed board localisation and avoids wasting time on a YOLO run that was already impossible from the input image.

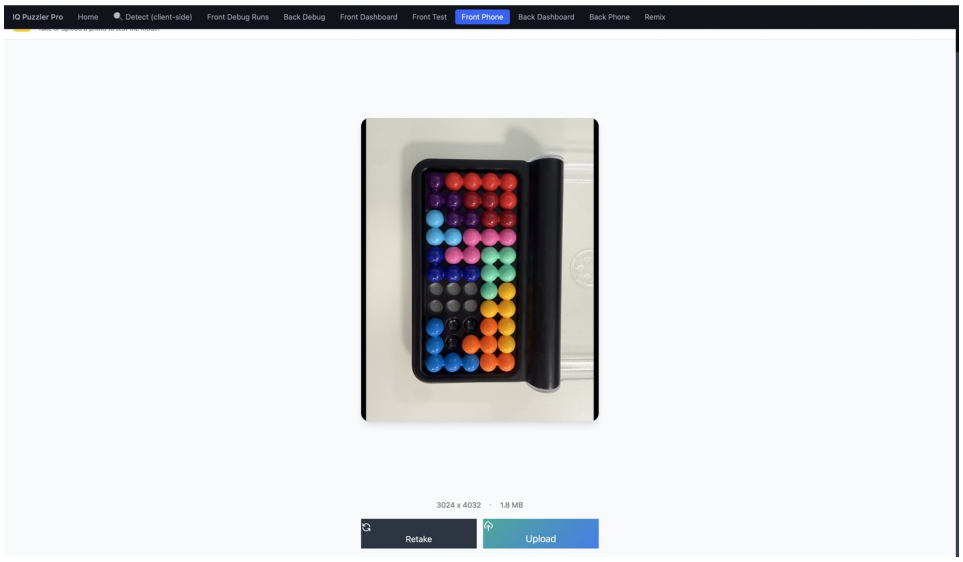

Figure 6. Front phone/gallery upload review before sending a real board photo to the backend.

Explanation: The review step prevents accidental submissions and gives the user confidence that the correct photo is being analysed.

After upload, the phone result confirms that the backend processed the image and returned a candidate detection. The implementation deliberately does not treat this as final truth; it tells the user to continue on the dashboard where the board state can be applied, validated and corrected.

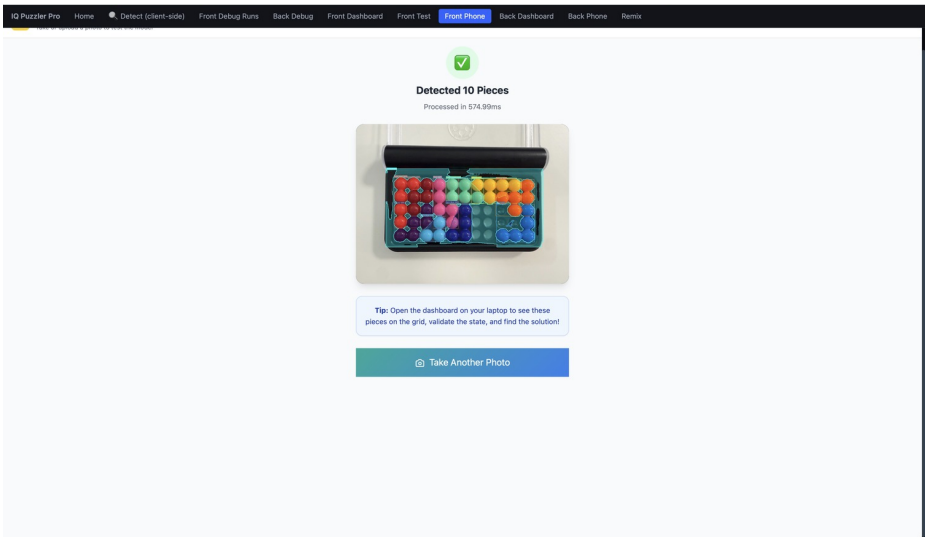

Figure 7. Front photo result after YOLO detection, ready to be reviewed on the dashboard.

Explanation: The phone result confirms that the backend processed the image and produced a candidate board state. The dashboard still remains responsible for applying and correcting it.

Every front detection run is stored as a timestamped debug folder with the original image, annotated model output, warp, grid and final labelled state. This made debugging evidence-based: when a photo failed, I could reopen the exact run and compare each stage instead of guessing from the final result.

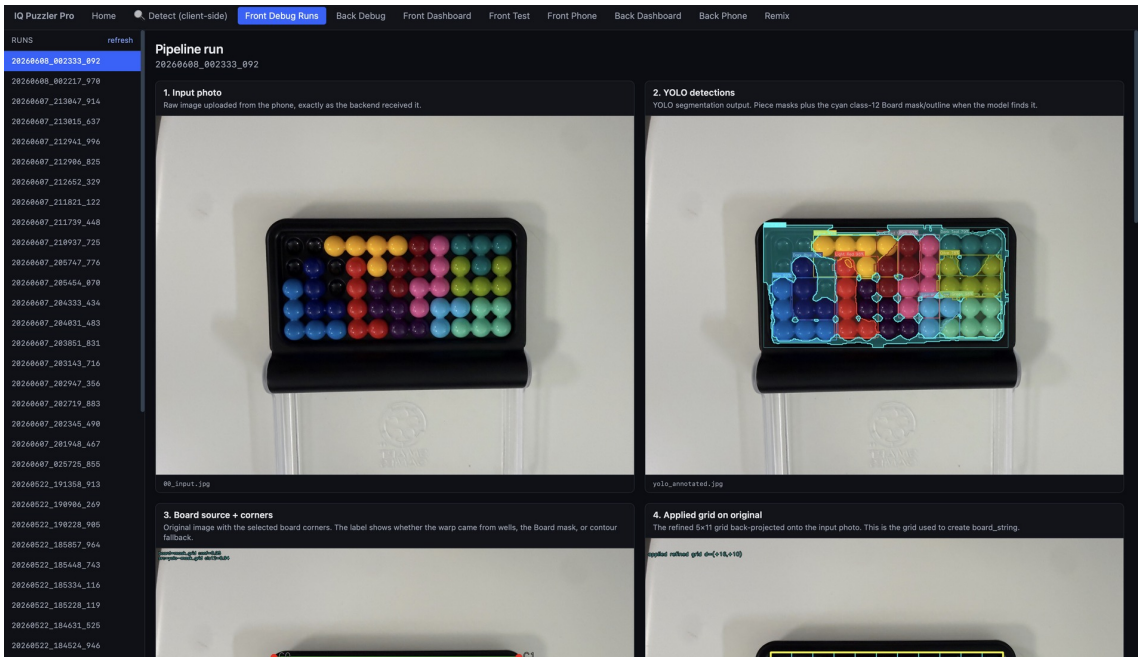

Figure 8. Front debug run list and detection artefacts saved by the backend.

Explanation: The debug list made failed cases traceable. Instead of guessing why a photo failed, I could reopen the saved run and compare each intermediate artefact.

This debug view explains whether the problem belongs to the model or to the geometry after the model. For example, if the YOLO mask is correct but the grid is shifted, the right fix is in board localisation or perspective mapping, not in collecting more labels for the piece class.

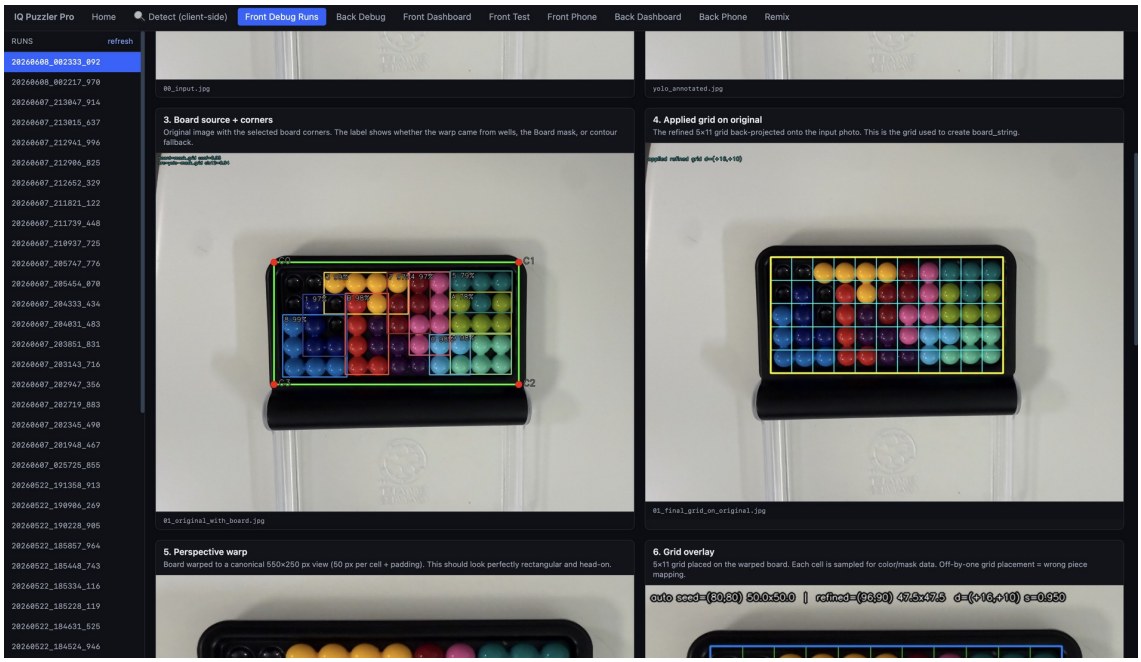

Figure 9. Front debug view showing image, masks and intermediate reconstruction.

Explanation: This screen helped separate model mistakes from geometry mistakes. If the mask looked right but the board state was wrong, the error was likely in mapping rather than in detection.

The final labelled overlay checks the strictest part of the front pipeline: a soft mask must become exact wells on a 5 by 11 board. Bounding boxes can look good while the logical board is wrong, so this screen verifies the actual cell-level result used by the solver.

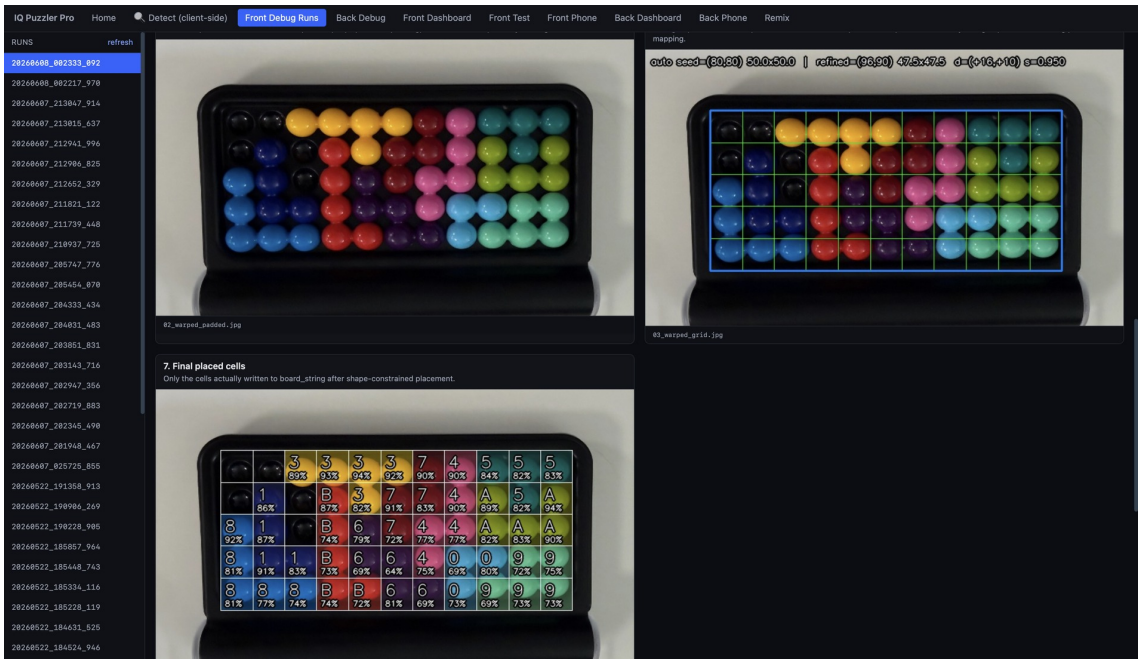

Figure 10. Front debug view showing final labelled cells after shape-constrained mapping.

Explanation: The final overlay shows whether detected pieces actually land on the grid. This was important because a bounding box can look correct while the logical cells are shifted.

The full debug page puts the main pipeline evidence in one place. It is used to compare the original photo, the perspective-corrected board, the recovered grid and the final labels so a later code change can be judged against the same failing case.

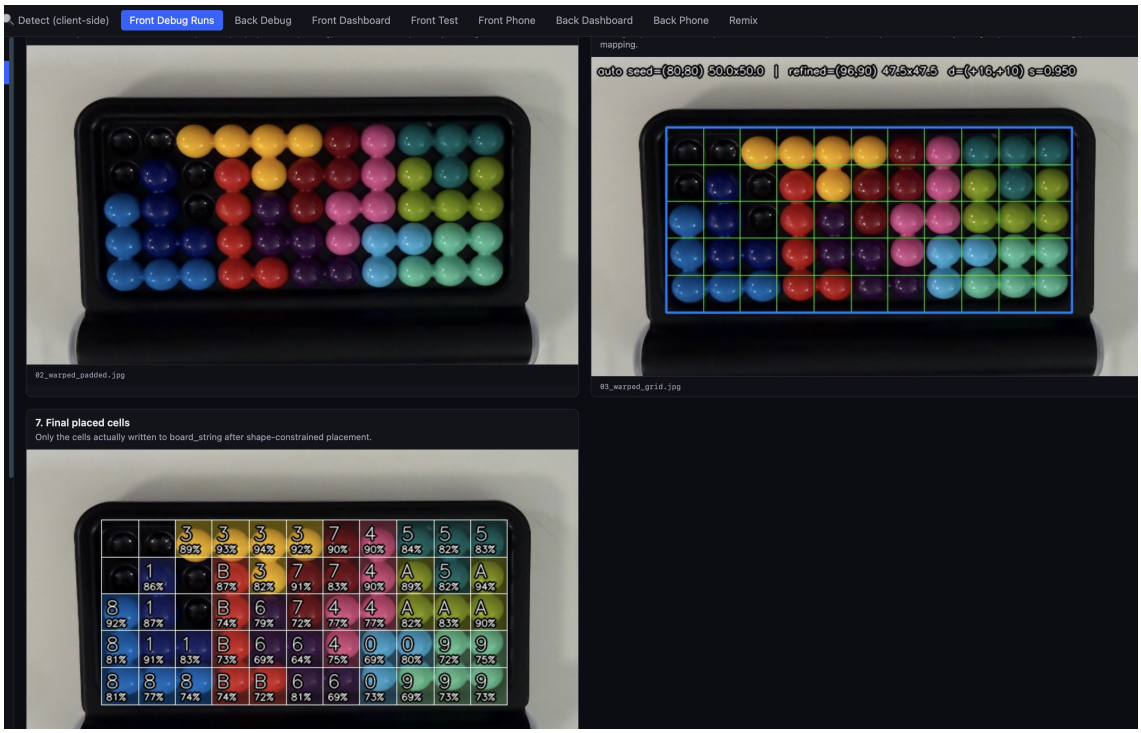

Figure 11. Full front-board debug run with warped board, recovered grid and labelled cells.

Explanation: This full debug view became the main evidence when improving the front pipeline. It shows the complete path from photo to cell labels.

A key correction was discovering that one piece definition in the software did not match the real physical piece. The detector was not the source of that bug. The mapper was searching valid placements for the wrong shape. After checking every piece definition against the real pieces, the mapping became more robust.

3.1.1. Front dashboard functionality

The dashboard is where AI output becomes an editable puzzle state. In the frontend store, the detected board state is kept apart from the currently played board until the user applies it. The same store then calls the backend solver endpoints for Validate, Hint and Solve. This was a deliberate design decision: the model proposes a state, but the deterministic solver and the user decide whether that state should be trusted.

The hint flow has three levels. The backend returns a valid continuation, and the frontend reveals it gradually: level 1 identifies a useful piece, level 2 narrows the target area and level 3 shows the exact cells. This avoids turning every hint into an instant solution while still supporting younger players or players who are completely stuck.

The front dashboard keeps the detected state separate from the playable board until the user explicitly applies it. In the React/Zustand flow this prevents uncertain AI output from silently overwriting manual work, while still making the detected 55-character board state available for review.

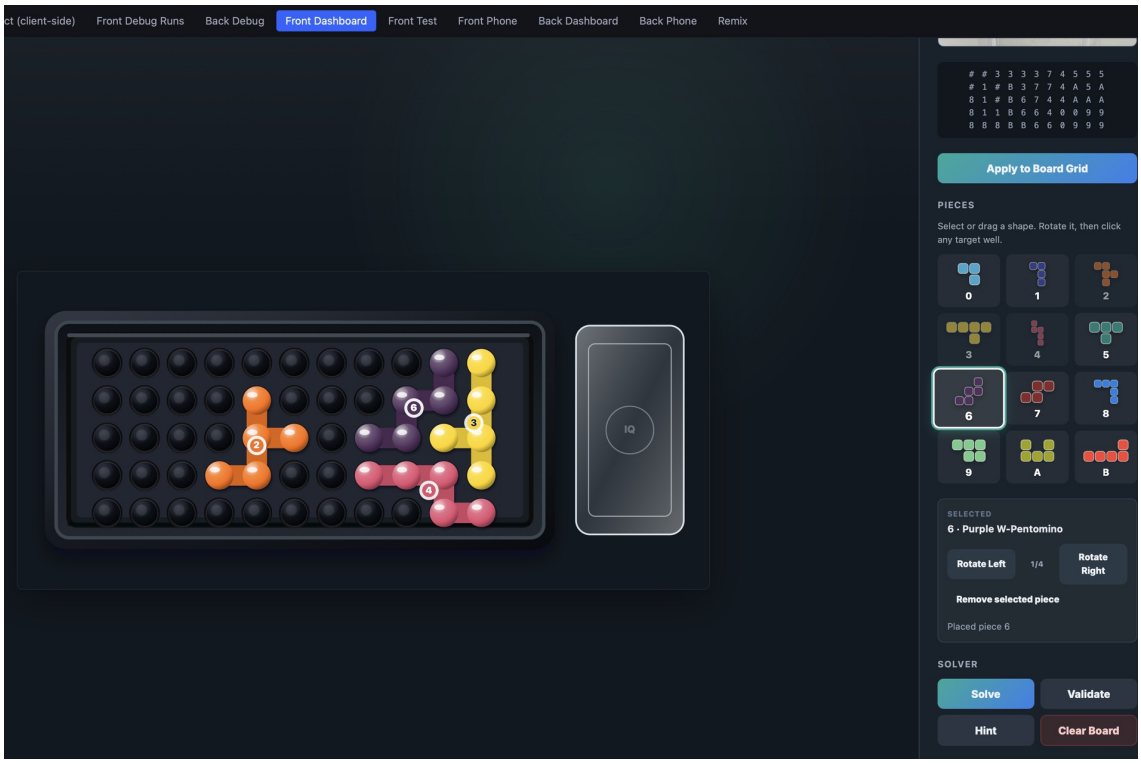

Figure 12. Front dashboard with an Apply to Board State action available after detection.

Explanation: This screen shows the handover between AI detection and the playable board. The user can inspect the proposed state before making it part of the board.

When the board state is applied, the dashboard renders the detected pieces through the same piece definitions used by the solver. The Validate button then posts the current 55-cell state to the solver endpoint, so the answer is based on puzzle logic rather than only on how convincing the screenshot looks.

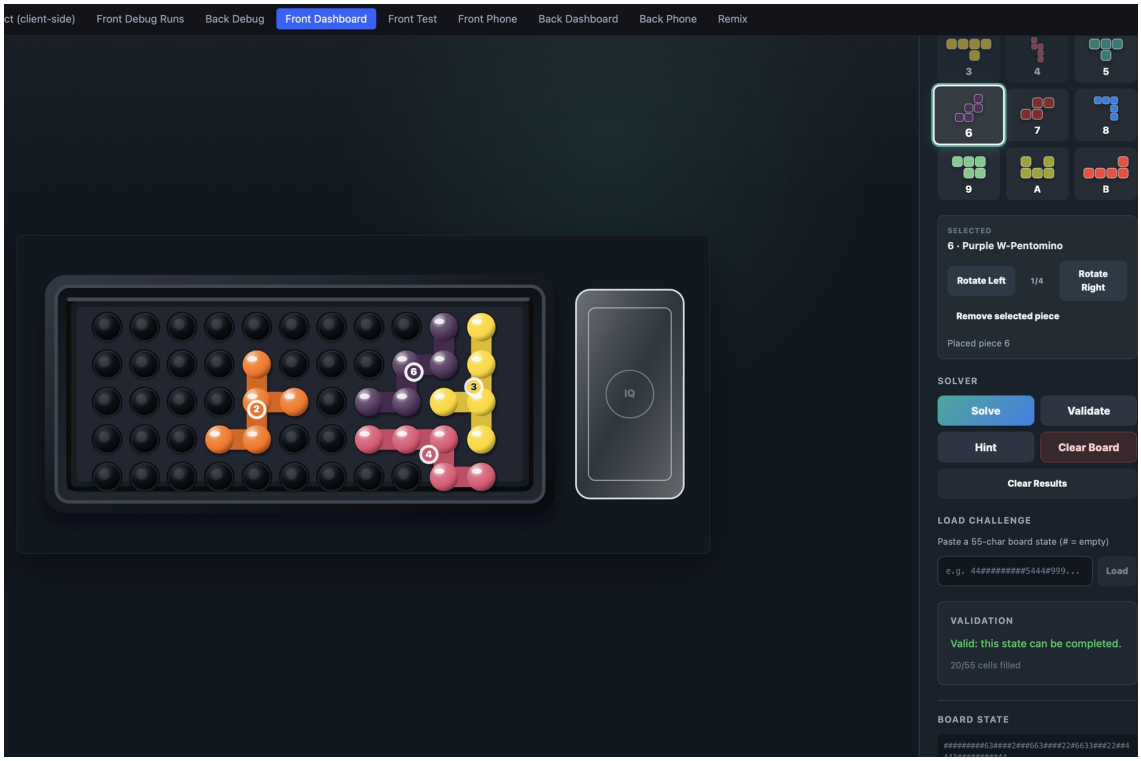

Figure 13. Front board after applying detected pieces and using Validate.

Explanation: Validation is not a visual check. It asks the solver whether the current partial board can still be completed, which prevents a wrong detection from becoming a trusted game state.

The front hint code first requests a full continuation from the backend and stores it as the full hint. Level 1 only projects the next useful piece into the interface, which gives the player direction while preserving the challenge.

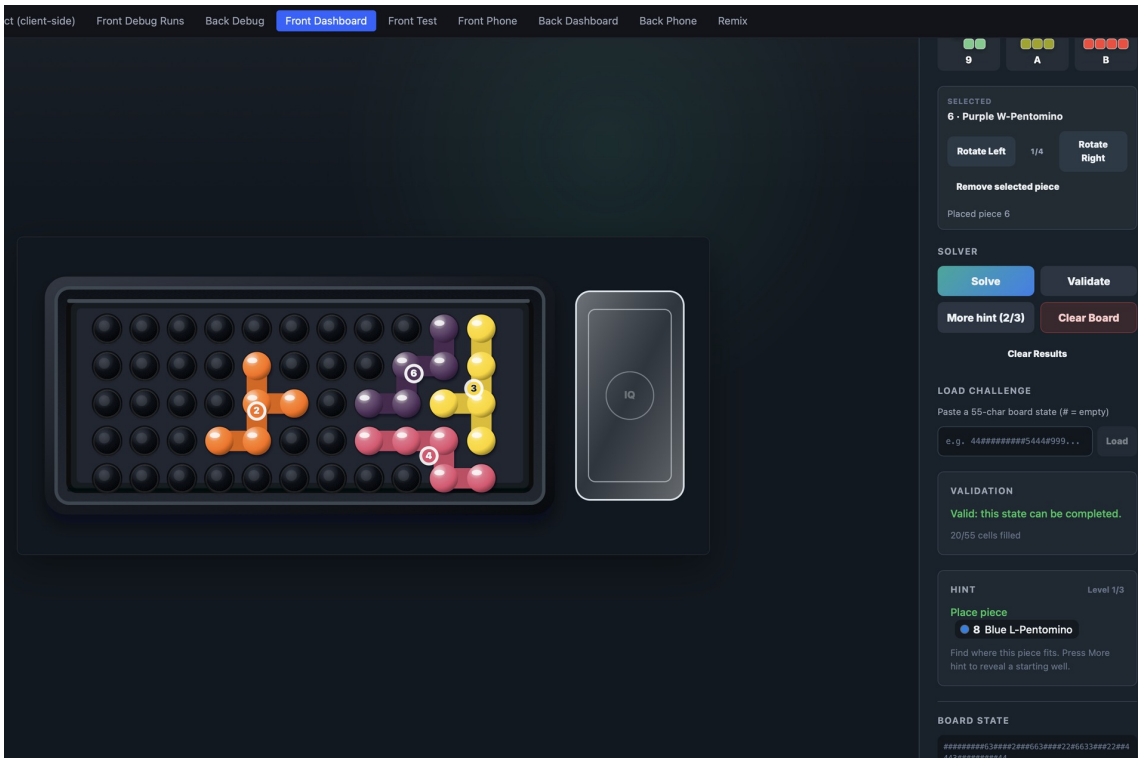

Figure 14. Front hint level 1 identifying a useful next piece without revealing exact placement.

Explanation: The first hint level is intentionally light. It helps the player continue while preserving most of the challenge.

Level 2 reuses the same solver result but reveals a more useful starting area. This progressive-disclosure design is intentional: the backend already knows the answer, but the UI controls how much help the player receives.

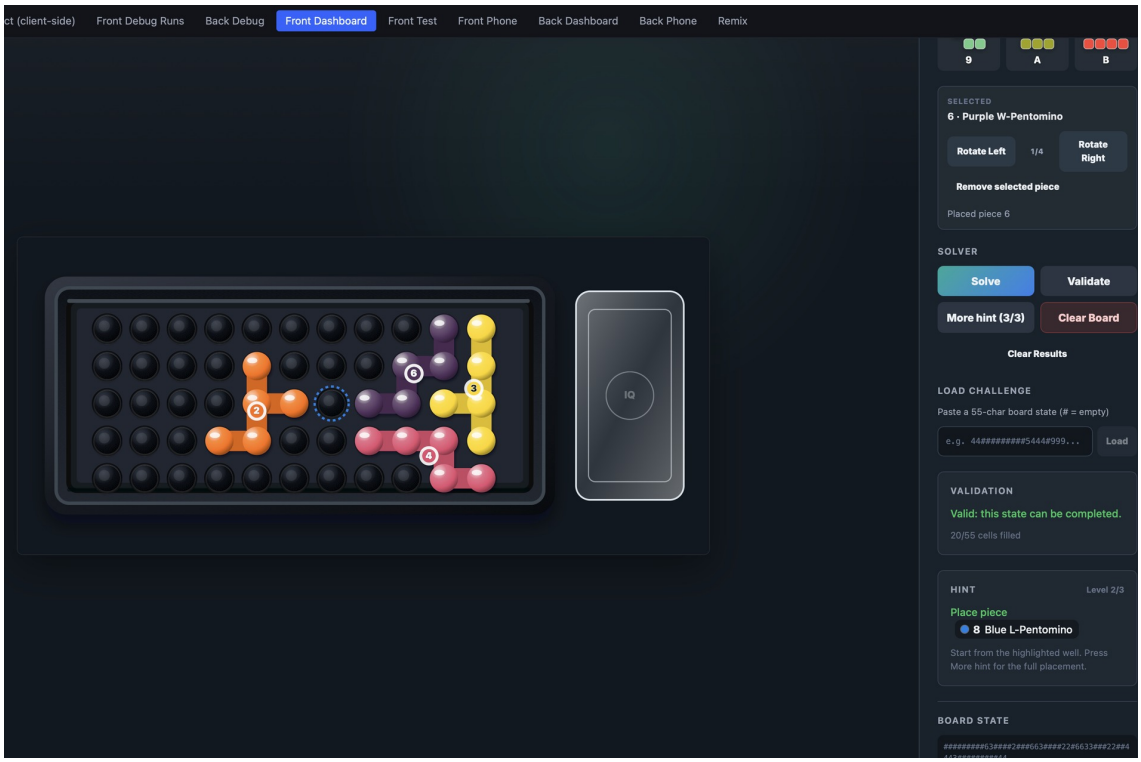

Figure 15. Front hint level 2 narrowing the area where the selected piece belongs.

Explanation: The second hint level gives more help by highlighting the area, but it still avoids immediately solving the placement for the player.

Level 3 reveals the exact cells from the solver's recommended placement. This level is meant for moments where the player is blocked, because it turns solver output into a visible target on the board instead of only returning a text answer.

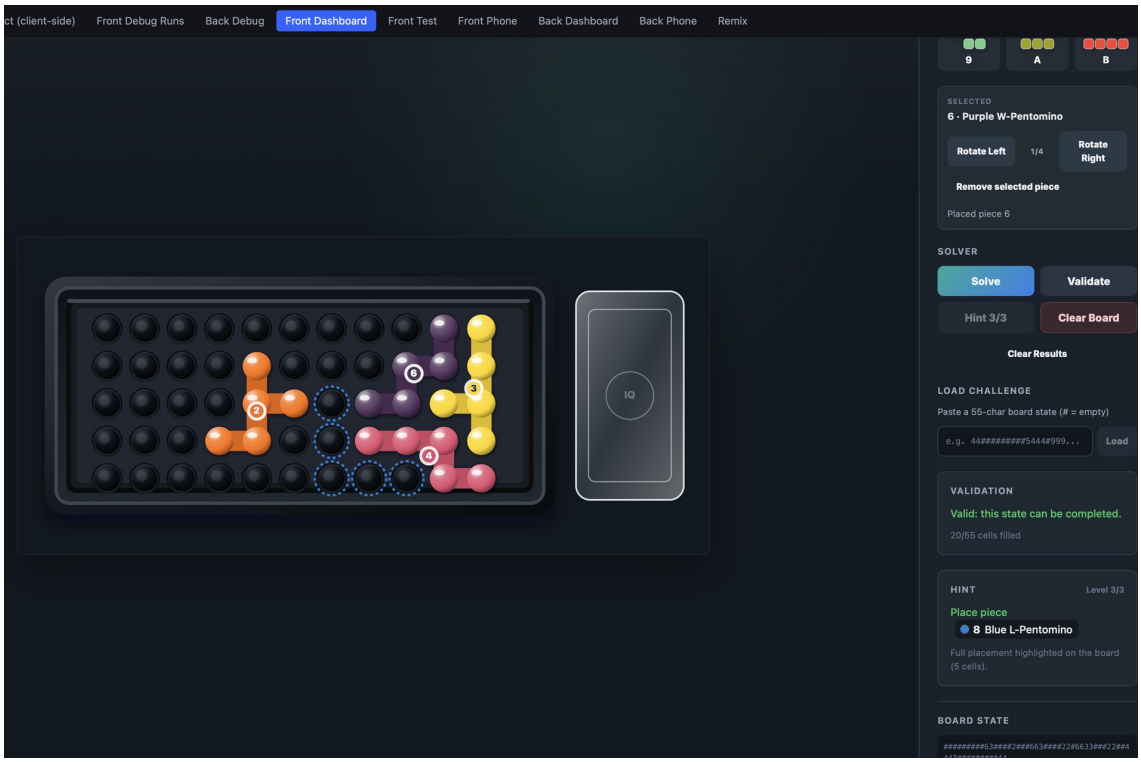

Figure 16. Front hint level 3 revealing the exact target placement.

Explanation: The third hint level is the strongest help. It exists for moments when the player is stuck and needs the exact cells.

After the strongest hint is visible, the app can write that placement into the board state. This matters for usability because a child or non-technical player does not need to translate the hint manually; the interface can apply the exact cells returned by the solver.

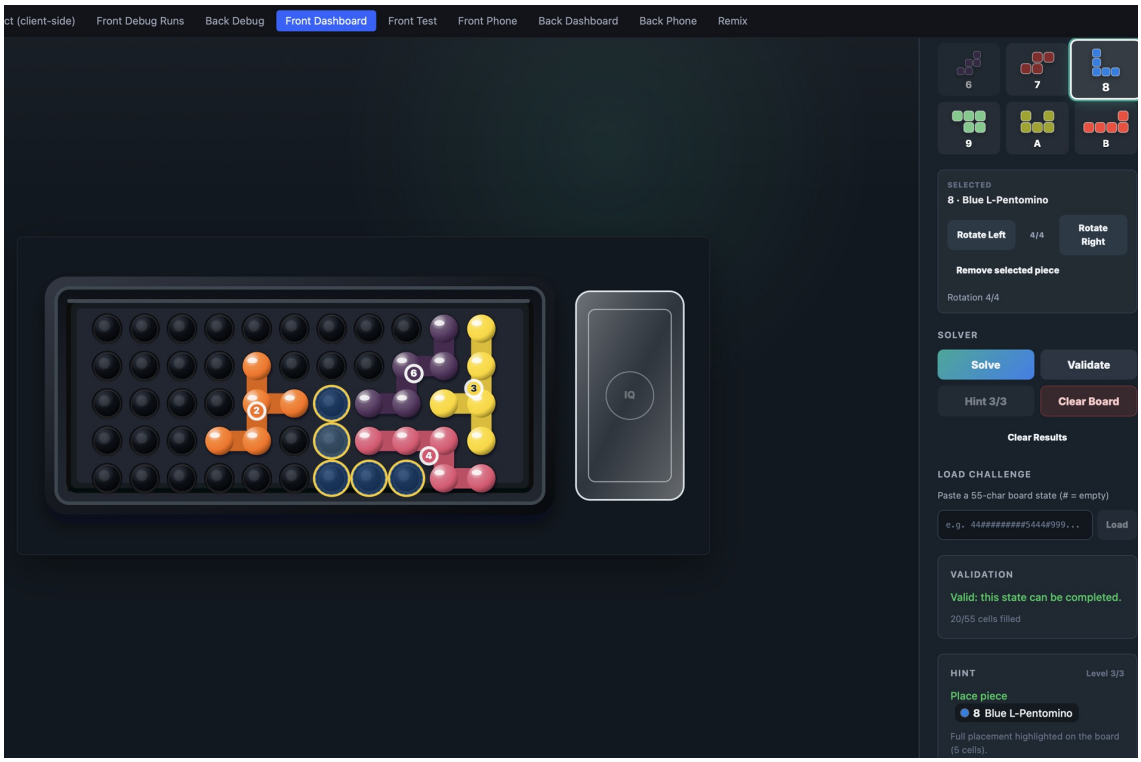

Figure 17. Applying the level-3 front hint onto the board.

Explanation: This demonstrates that the hint is not only text. The app can apply the suggested piece to the exact wells, which makes the feature usable for younger players.

The Solve button sends the current state to the solver endpoint and renders the returned complete board. This proves that the detected or manually corrected state uses the same piece IDs, rotations and cell ordering as the backend exact-cover search.

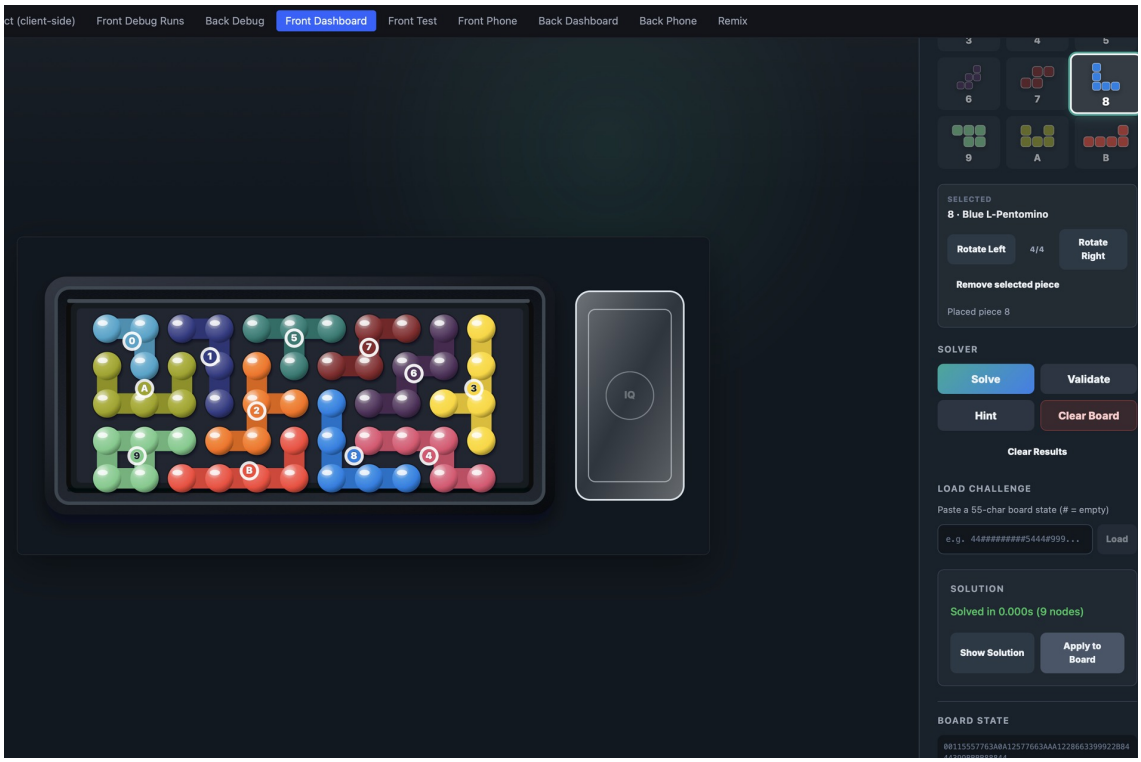

Figure 18. Front solver result after pressing Solve.

Explanation: The solver fills the remaining cells with a complete valid board. This was useful for testing and for proving that the state representation is compatible with the solver.

This screen is the transition between computer vision and gameplay. The YOLO result has been processed and displayed, but the playable board is still waiting for the user to confirm that the detected board state should be accepted.

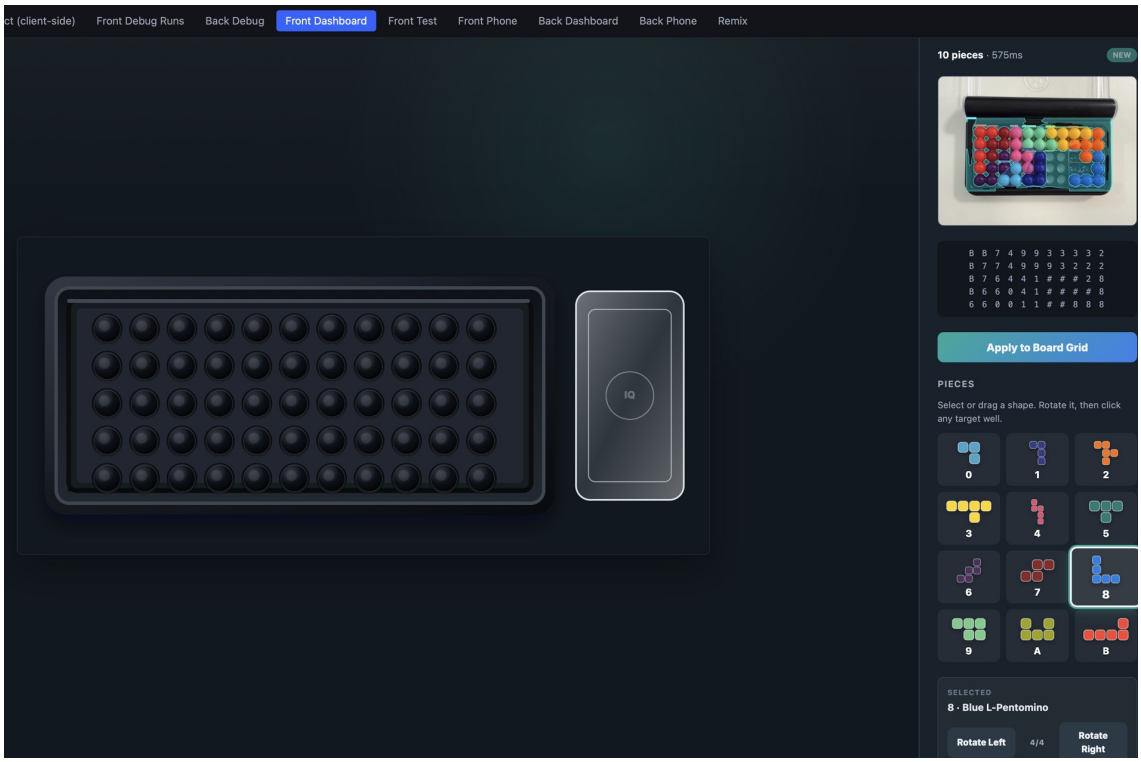

Figure 19. Front board after a processed photo is rendered and waiting to be applied.

Explanation: This figure fixes the flow between detection and dashboard use. The board state exists, but the user has not yet applied it to the playable front board.

After Apply is clicked, the dashboard converts the detected 55-character state into rendered pieces on the front grid. This step proves that the model response, board-state parser and board renderer agree on the same cell order.

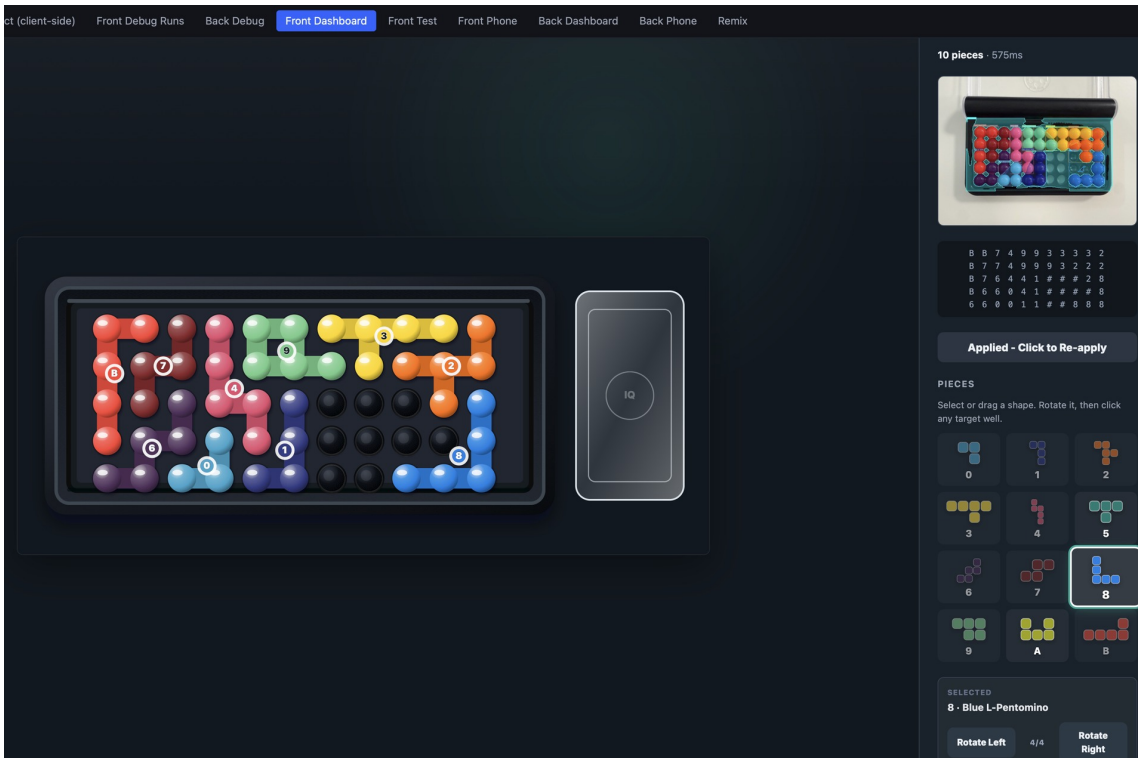

Figure 20. Front board after clicking Apply to Board Grid.

Explanation: This is the follow-up to the previous figure. It shows the detected pieces placed on the playable front board after the user accepts the detected state.

3.2. Back-board realisation

The back board uses the same physical pieces but a different board shape. The difficult part was not only detecting coloured balls; it was making the detector, dashboard and solver agree on the same coordinate system. Earlier interpretations looked plausible visually but produced impossible logical states.

This part of the project showed why a puzzle application cannot rely only on screenshots. A back-board image can look convincing, but if one coordinate convention treats the wells as a hex layout and another treats them as a square-diamond mask, the solver will reject states that appear visually reasonable. The final implementation therefore made every layer use the same board truth: the camera pipeline, the dashboard, the piece definitions and the exact-cover solver all refer to the same 55 cells.

The back phone client posts to a separate back detection route because the response has extra quality information and a different board preview. The upload result uses the solver status, placed-cell count and review flags to decide whether the user can trust the result immediately or should continue on the laptop dashboard. This was important because the back side has more geometric ambiguity than the front side.

The back phone flow is separate because the back side uses a square-diamond active mask instead of the front rectangular board. Keeping it separate made the backend response, review status and preview board match the back-side coordinate rules.

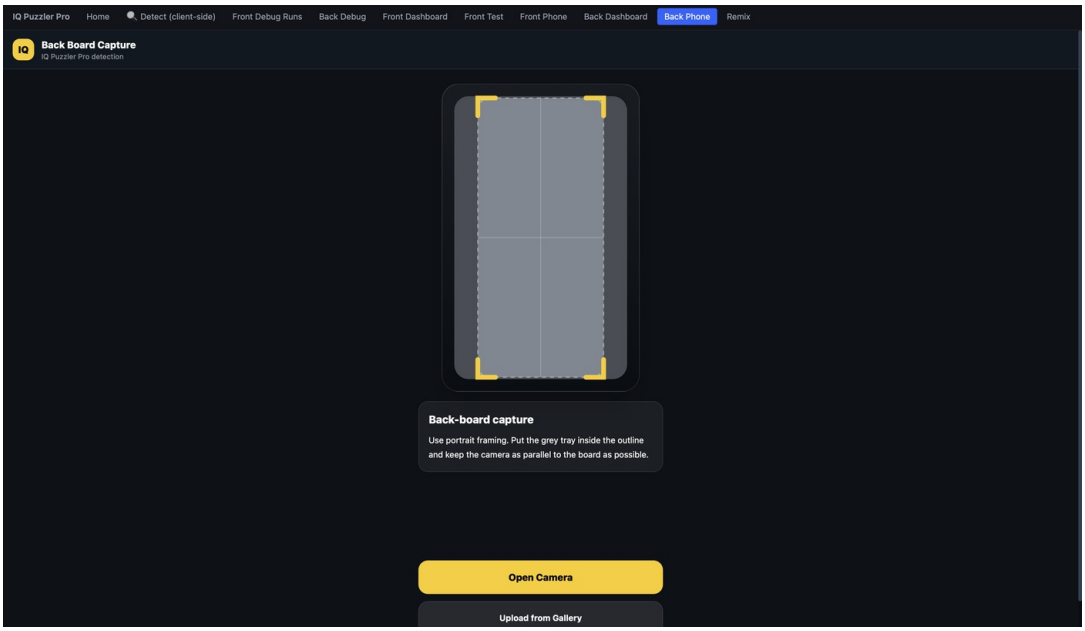

Figure 21. Back phone capture screen before taking or uploading a photo.

Explanation: The back capture flow is separate from the front flow because the board geometry and mapping rules are different.

The back-board review screen reduces a different risk than the front board: the diamond board can be angled and partially cropped more easily. The user can check that the whole active area is visible before the backend starts segmenting pieces and mapping wells.

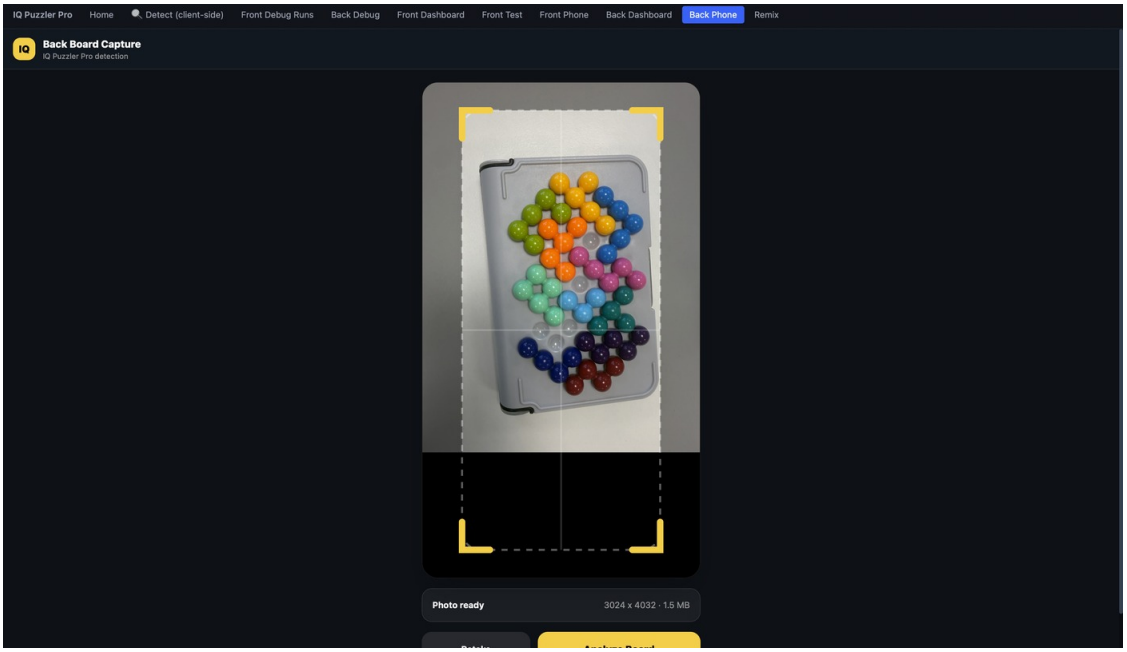

Figure 22. Back phone photo review before sending the image to the backend.

Explanation: The review screen lets the user confirm that the diamond board is inside the guide before processing starts.

The back result combines model output with solver and quality information. The UI marks a result as needing review when validation fails, confidence is questionable or quality flags are present, because the back-board geometry is less forgiving than the front side.

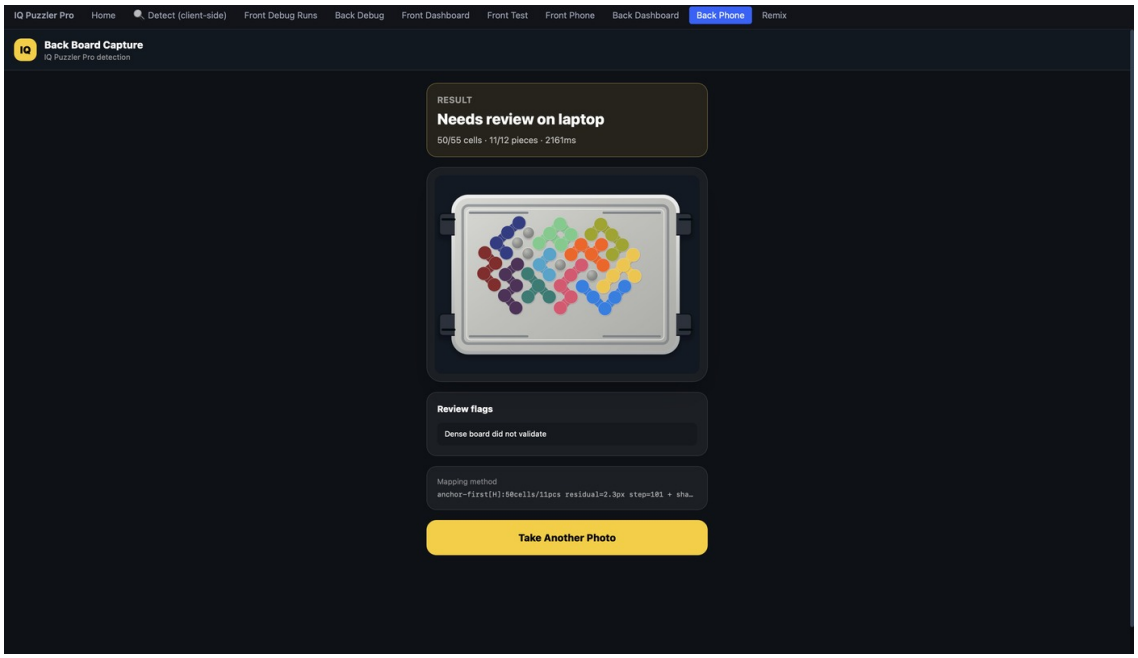

Figure 23. Back phone result screen showing review status and mapped board preview.

Explanation: The result screen can warn that the state needs laptop review. This is important because back-board geometry is less forgiving than the front board.

The back debug view was needed because many wrong results looked visually plausible at first. By comparing the original overlay, warped board and grid overlay, I could see whether the error came from board localisation, perspective correction or the final well mapping.

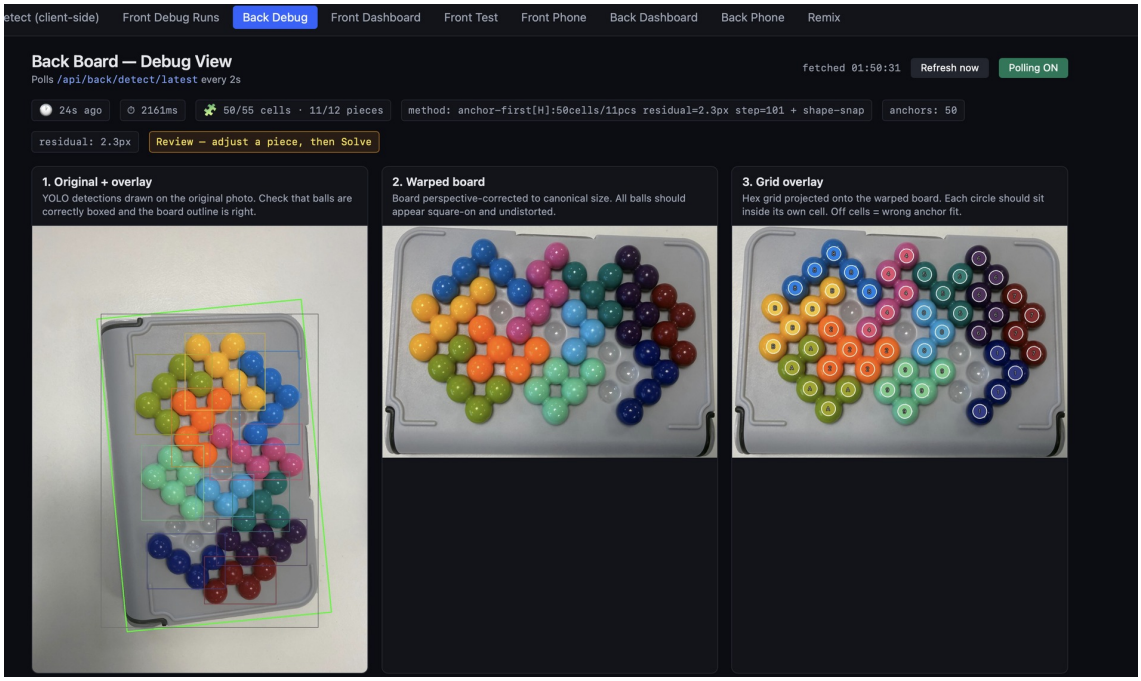

Figure 24. Back debug view with original overlay, warped board and grid overlay.

Explanation: The three-panel debug view shows whether the issue is board localisation, warping or cell mapping. This was essential for fixing the back-board pipeline.

The status table exposes the mapped state, counts, confidence and solver result instead of hiding the result behind a simple success message. This helped identify cases where the model found pieces but the mapped 55-cell state was still impossible.

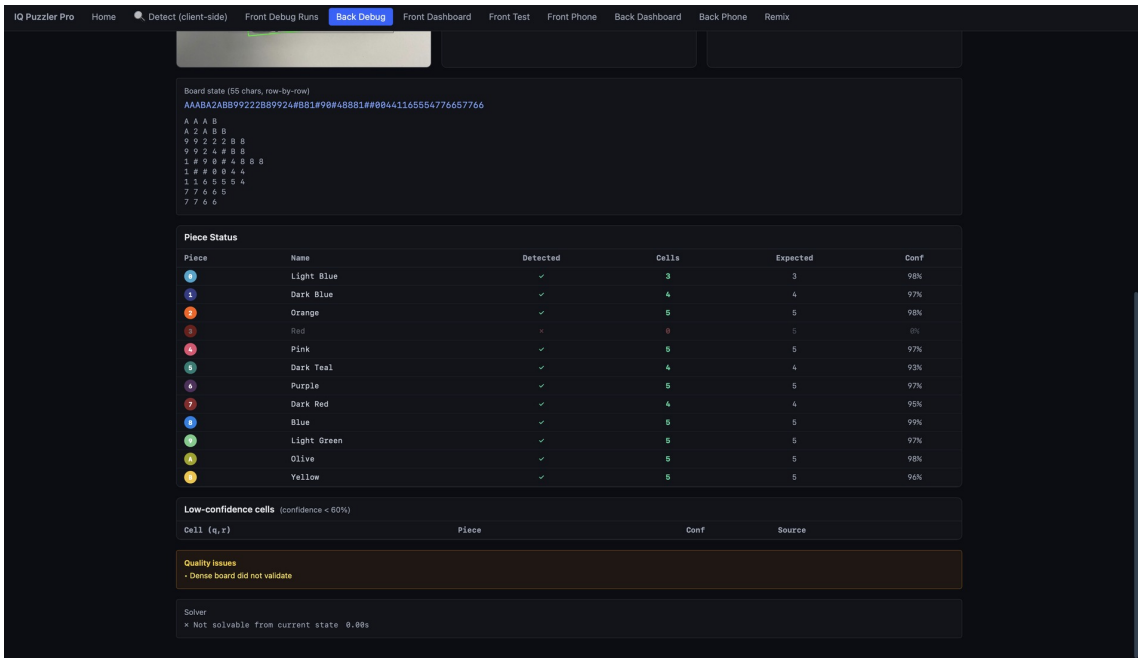

Figure 25. Back debug status table with mapped state, confidence and validation result.

Explanation: The status table shows piece counts, confidence and solver validation. It helps explain why a visually convincing detection may still need manual correction.

The back dashboard then gives the user a correction interface over the detected state. The right side contains the selected piece, rotation controls, solver actions and the three-level hint panel. The board area renders the square-diamond mask, so manual placement uses the same cells as the detector and solver. This was the main fix for cases where a visually correct image still produced pieces in the wrong wells.

The back dashboard is the manual correction layer over the detected state. It lets the player select a piece, rotate it, place it on the square-diamond board, remove mistakes and then validate again, which is necessary because camera detection cannot be perfect in every real setting.

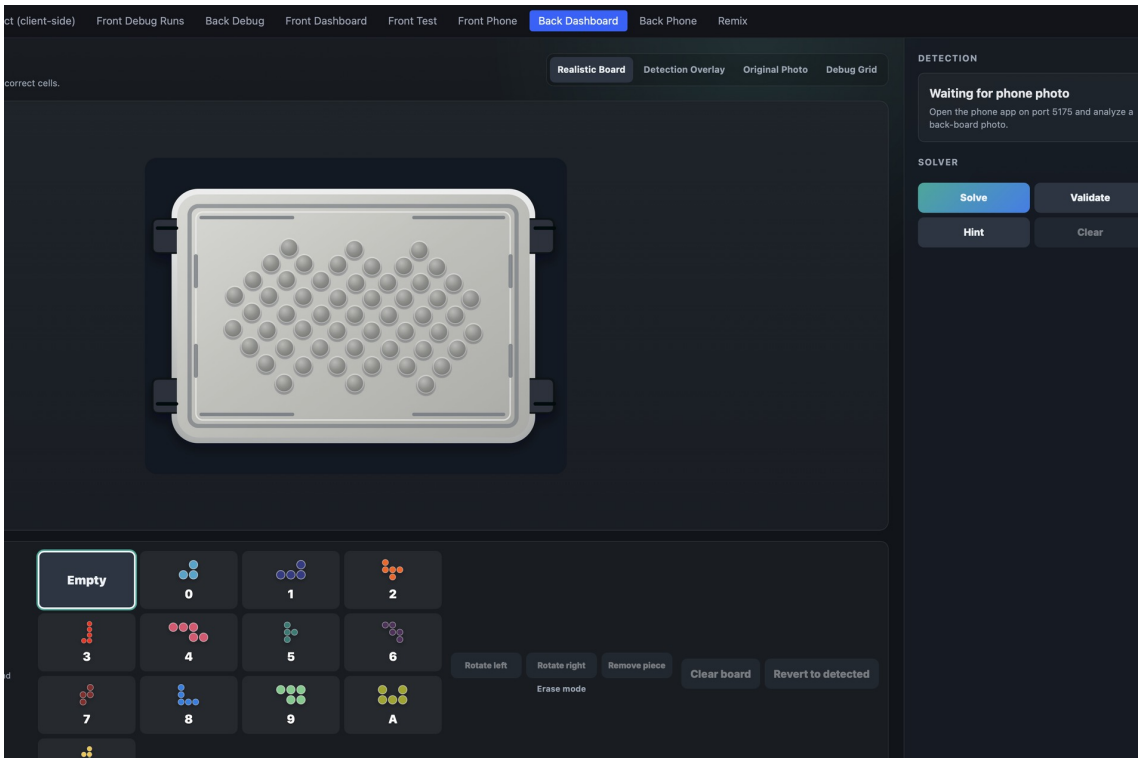

Figure 26. Back dashboard with piece palette and board controls ready for correction.

Explanation: This is the actual back-board dashboard. It uses the diamond board and gives the user controls for selecting, rotating, removing and correcting pieces.

The first back hint follows the same product rule as the front board: help without solving everything immediately. The solver chooses a valid next piece, while the UI keeps the exact placement hidden so the player still has to reason.

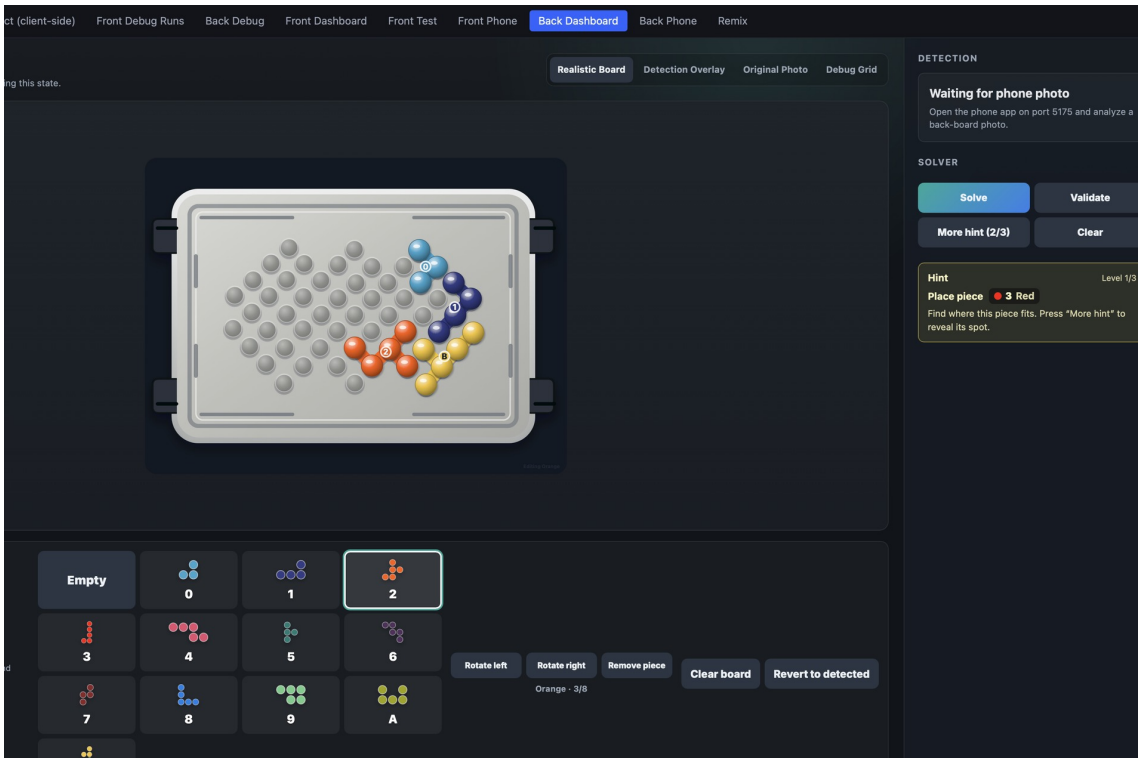

Figure 27. Back hint level 1 identifying a helpful next piece.

Explanation: The first back-board hint follows the same principle as the front board: give useful direction without revealing everything immediately.

Level 2 on the back board reveals a starting well or smaller target area on the diamond coordinate system. This proved that the hint feature was not hard-coded for the rectangular board; it could work after the board geometry changed.

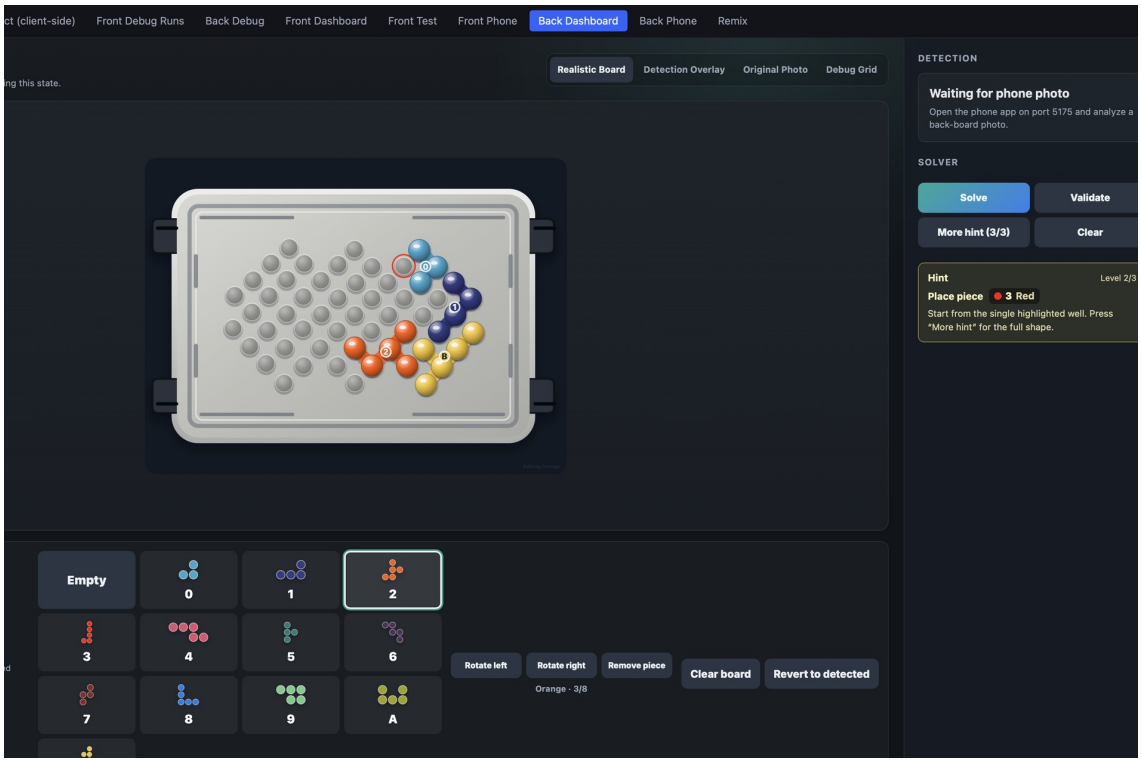

Figure 28. Back hint level 2 narrowing the target area.

Explanation: The second hint level makes the target area clearer while still leaving the player some reasoning work.

Level 3 reveals the full placement on the back board. This was an important validation point because every highlighted cell must match the same square-diamond mask used by detection, manual placement and the solver.

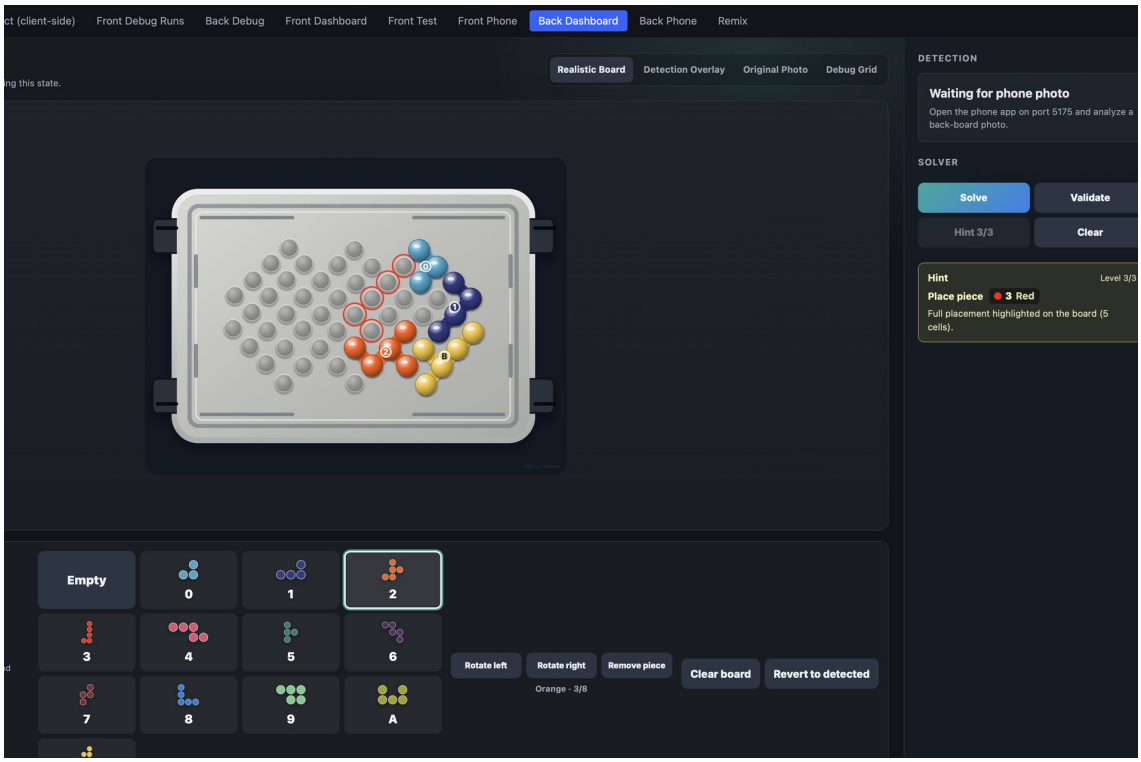

Figure 29. Back hint level 3 revealing exact placement on the diamond board.

Explanation: The third level shows the exact cells. This proves the hint engine works on the back-board coordinate system, not only on the rectangular board.

The back Validate button checks the currently edited board state with the exact-cover solver. This is how the app distinguishes a board that merely looks possible from a board that can actually be completed.

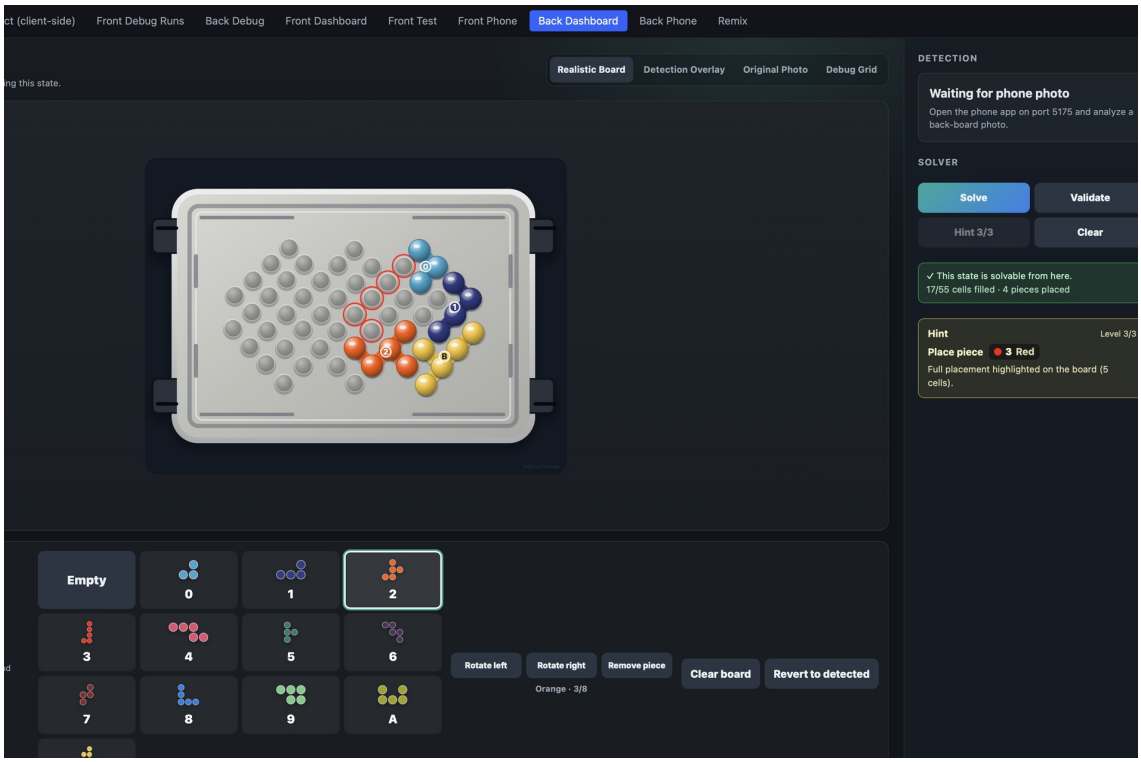

Figure 30. Back validator state after pressing Validate.

Explanation: The validator confirms whether the current diamond board can still be solved. This links the UI to the exact-cover solver.

The back Solve result fills the remaining diamond cells from a valid solver completion. It proves that the coordinate model, piece definitions and dashboard rendering agree after the back-board redesign.

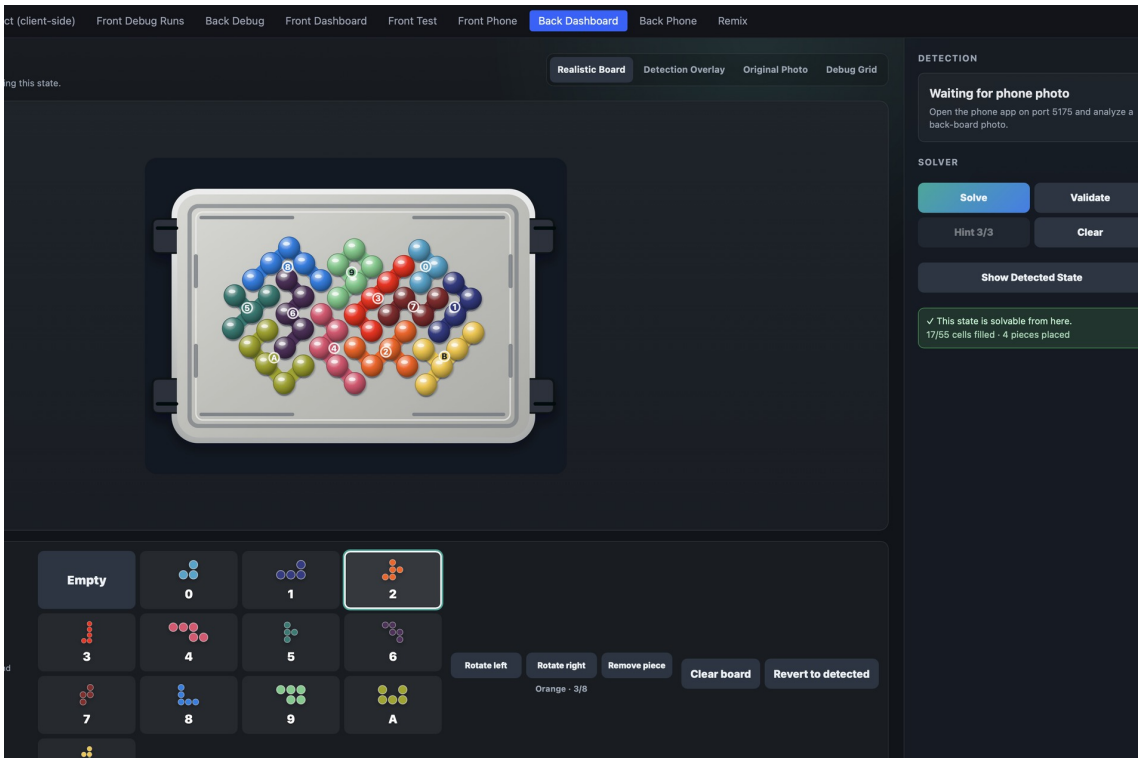

Figure 31. Back solver result after pressing Solve.

Explanation: The solver fills the remaining diamond cells with a valid solution, proving that the geometry and piece definitions agree.

The main lesson from the back board was that a visually plausible grid can still be the wrong mathematical model. The final square-diamond representation made the board mask, dashboard rendering, piece shapes and solver use the same truth.

3.3. Remix prototype

Remix started during an internal AI jam on 21 May 2026. The first idea was a purely digital product, but feedback from an employee changed the direction. He explained that the game would not feel as enjoyable if it only existed on a screen, because the physical placing of pieces is part of the value. We therefore took the concept further and connected generated shapes to printable export.

The technical idea behind Remix is that the original rectangular board is not the only possible shape. If a custom outline contains exactly 55 cells and the twelve IQ Puzzler pieces can cover those cells without overlap, the solver can turn that outline into a real challenge. This changed the project from recognition of an existing product into generation of new puzzle content.

I worked with Bart Van Assche during the jam. My focus was the software: shape validation, solver integration, playable screens, difficulty levels and export logic. Bart worked on the Blender and printing side. Ibrahim Afkir and Wouter Caeldries worked as the other pair during the same AI-jam day.

The Remix code uses a mask-aware representation. Instead of assuming a fixed rectangle, the Play screen sorts the active mask cells and converts them into a 55-character board

state for the solver. Shape Studio validates custom masks only when they contain exactly 55 cells, and the challenge pages create several difficulty tiers by changing how many pieces are already shown. The same validated shape can therefore become a daily game, a themed booklet challenge or an STL export.

Remix extends the project from recognising an existing puzzle to generating new puzzle content. The daily page uses a generated or selected 55-cell mask and presents it as repeatable content, similar to a daily challenge format.

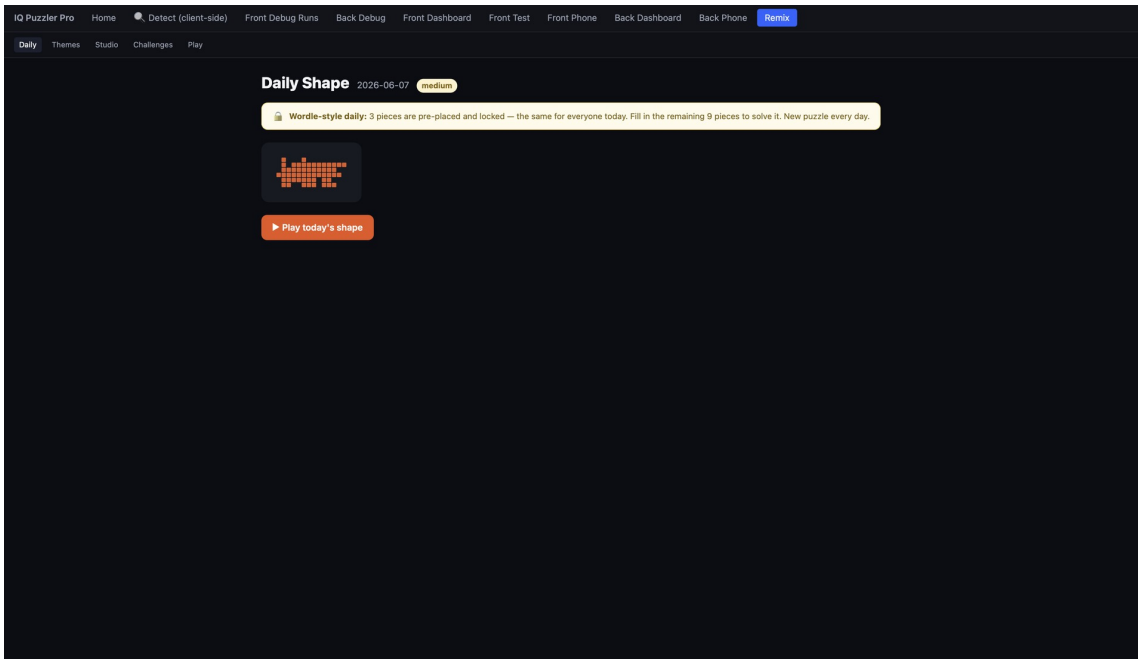

Figure 32. Remix daily page offering a Wordle-style generated shape.

Explanation: The daily page turns the puzzle into repeatable content. Each day can have a new 55-cell shape while still using the same twelve physical pieces.

The Remix play screen uses a mask-aware board component instead of the original fixed rectangle. Only cells inside the active mask can be filled, and piece placement checks the selected piece rotation against that mask and the already placed pieces.

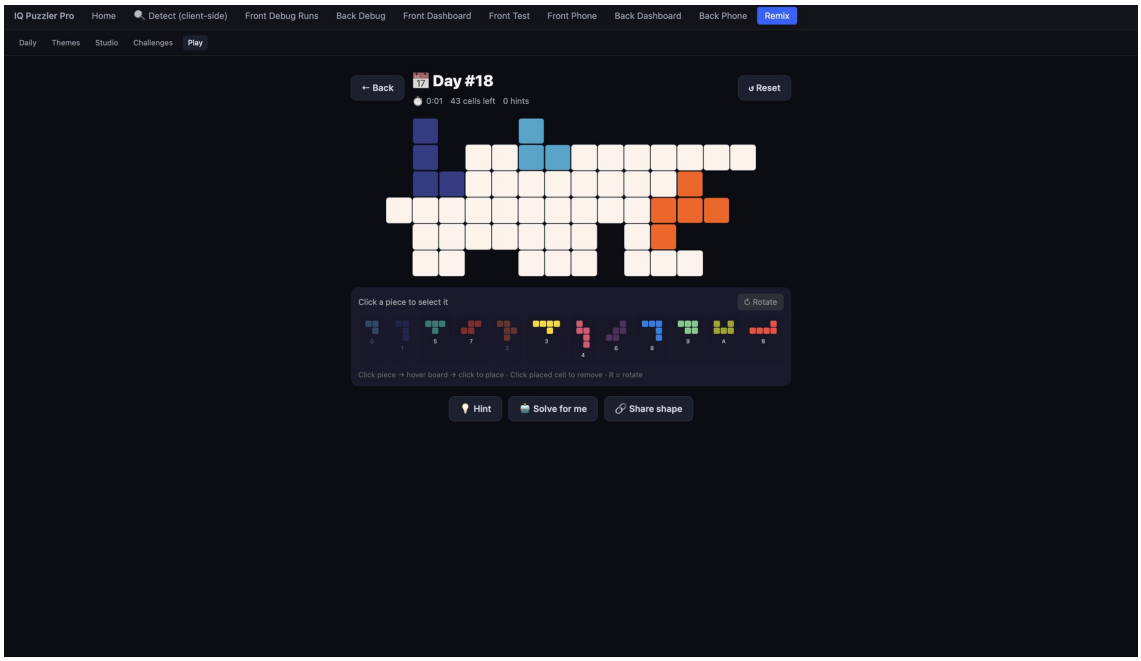

Figure 33. Daily shape in play with a generated board mask and interactive pieces.

Explanation: This screen shows the generated mask as an actual playable puzzle, not only as an abstract solver result.

Remix hints are generated from the current mask and board state through the backend solver. This keeps the hint correct for arbitrary shapes because it is not based on a fixed front-board layout.

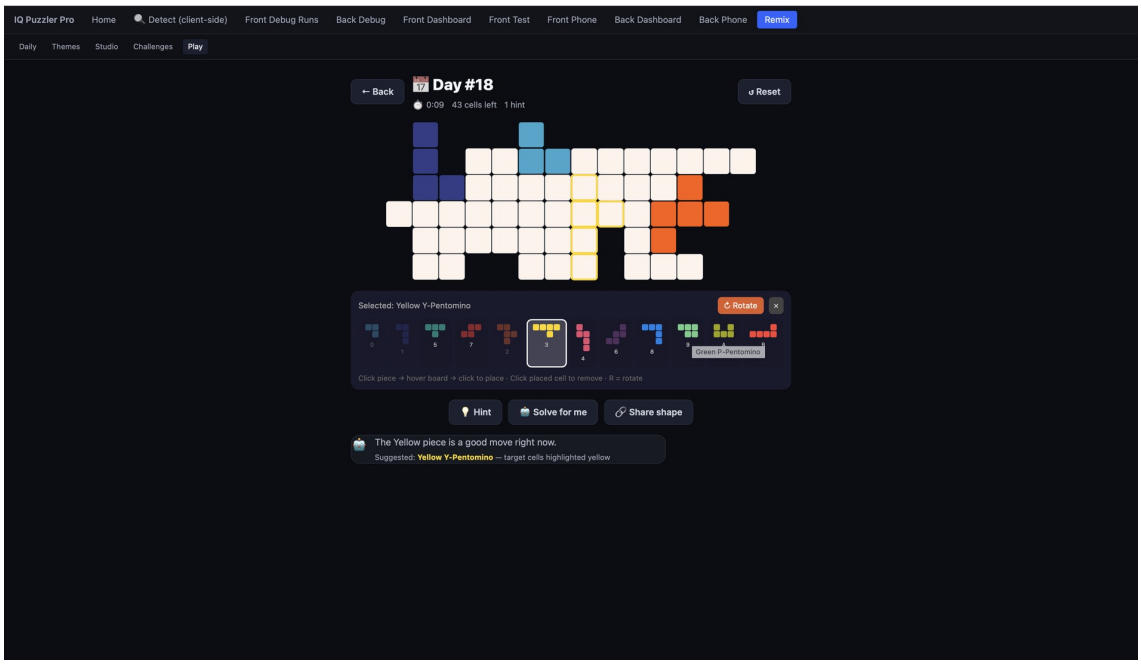

Figure 34. Remix hint state selected by the deterministic solver.

Explanation: Hints in Remix are generated from the same solver logic. The hint is valid because it comes from a real solution path.

The solved screen turns the solver prototype into a game loop. The Play page tracks elapsed time, hint count and completion state, then builds a shareable result so the generated puzzle feels like a product feature.

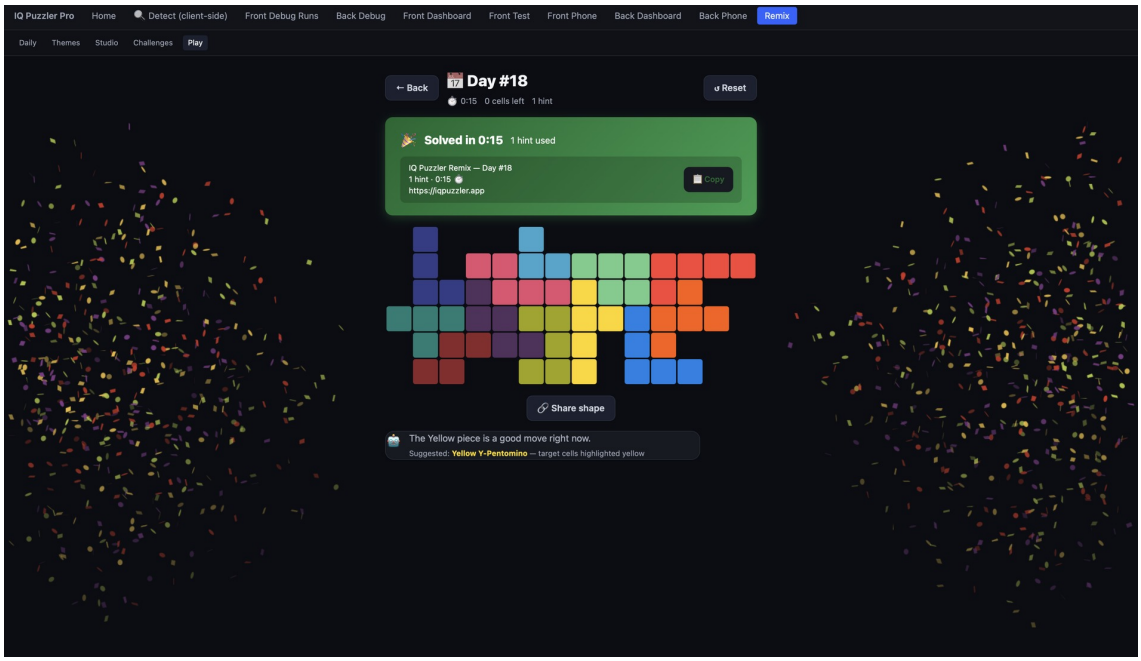

Figure 35. Remix solved screen with time, hint count and share text.

Explanation: The solved screen adds a product layer around the solver by giving the player completion feedback and shareable results.

Themes make Remix easier to understand than a random mask generator alone. Each theme is a predesigned shape family that has already been validated, which helps the user choose a challenge style before entering the booklet or play screen.

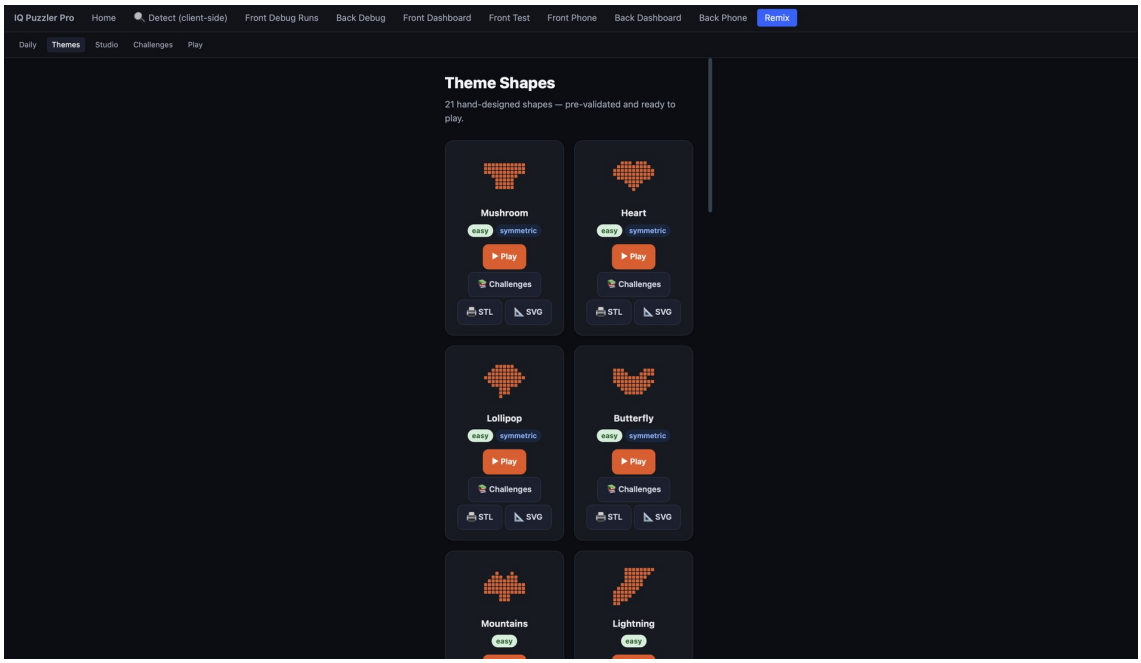

Figure 36. Remix theme-shape page with hand-designed shape categories.

Explanation: Theme shapes group generated challenges so the player can choose a shape family instead of only receiving random boards.

The selected-theme page connects one shape to multiple starts. The implementation can keep the same mask while changing pre-placed pieces, which is how the app creates a booklet-like set of challenges from one generated board.

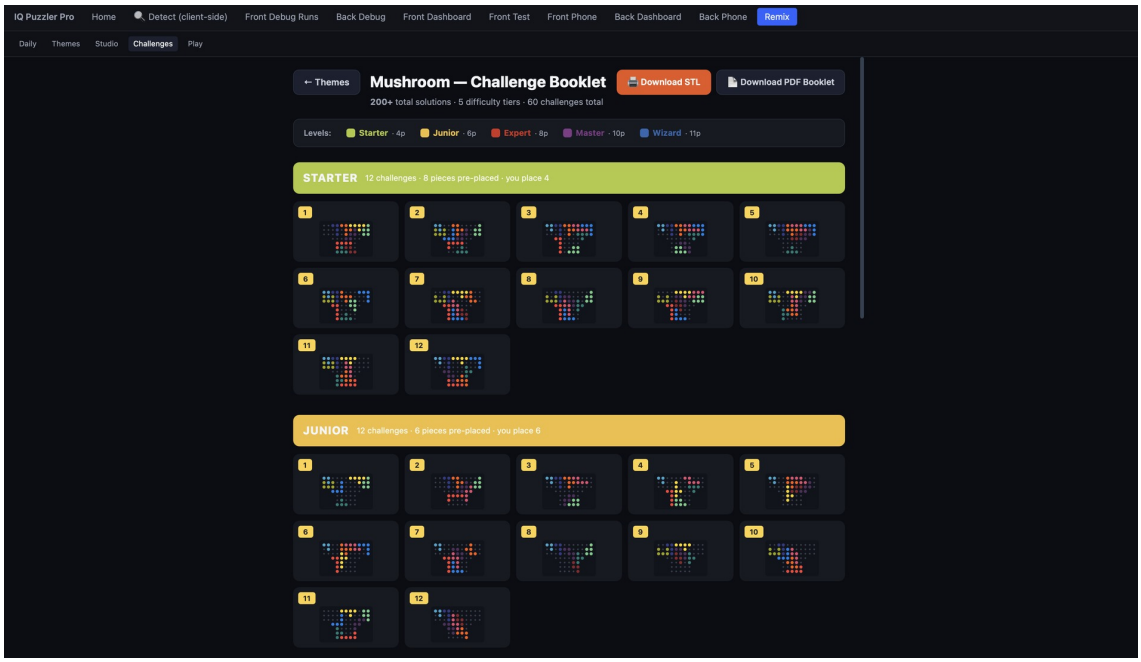

Figure 37. Selected-theme challenge page showing challenges for one theme.

Explanation: The selected-theme page connects one shape to multiple puzzle starts. This is similar to how a physical challenge booklet offers several levels.

Difficulty is expressed through challenge tiers, not only a label. The card generation varies how many pieces are already shown and how much solving work remains, so easy, medium and hard boards feel different while still using the same validated mask.

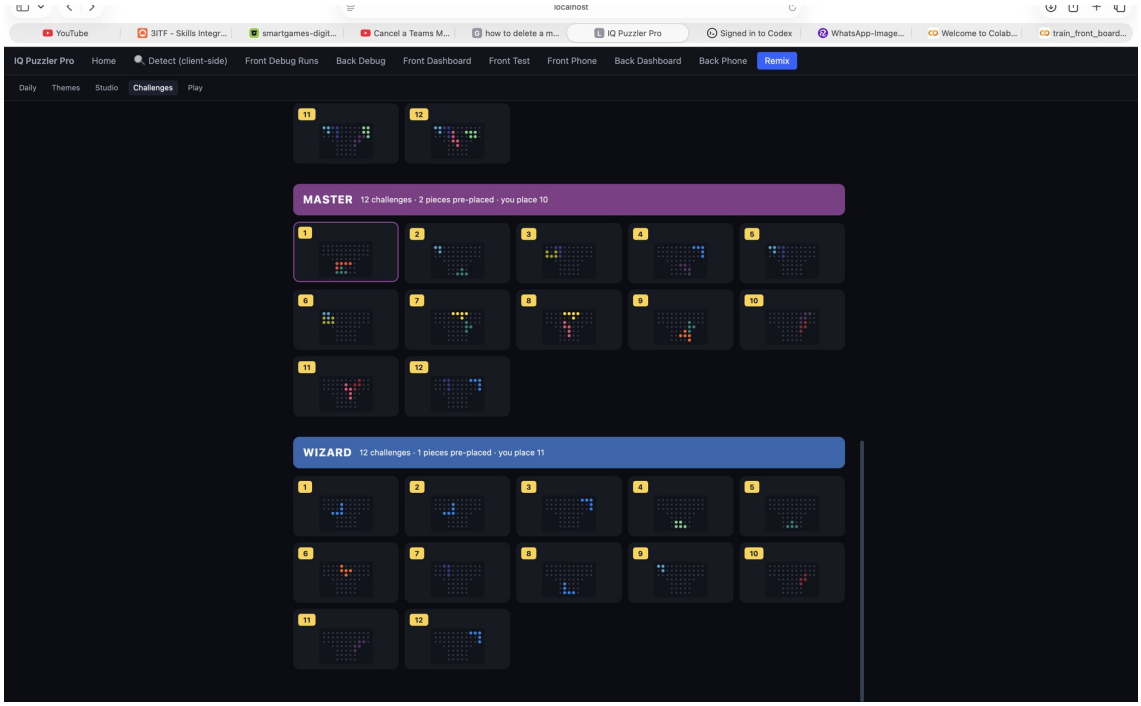

Figure 38. Challenge cards organised into difficulty levels.

Explanation: Difficulty is created by changing how much of the solution is given at the start and by checking solver complexity. Harder puzzles leave more reasoning to the player.

The booklet export shows why Remix was not only a browser experiment. It converts a validated shape and solver-backed challenges into printable material, which matches how physical SmartGames products are normally consumed.

Figure 39. Generated booklet cover for a selected Remix theme.

Explanation: The booklet cover demonstrates that Remix can produce printable material, not only an interactive browser view.

Starter pages contain the player-facing challenges from the generated booklet. They use pre-placed pieces and the selected mask so the player can start from a partial board instead of seeing the complete solution.

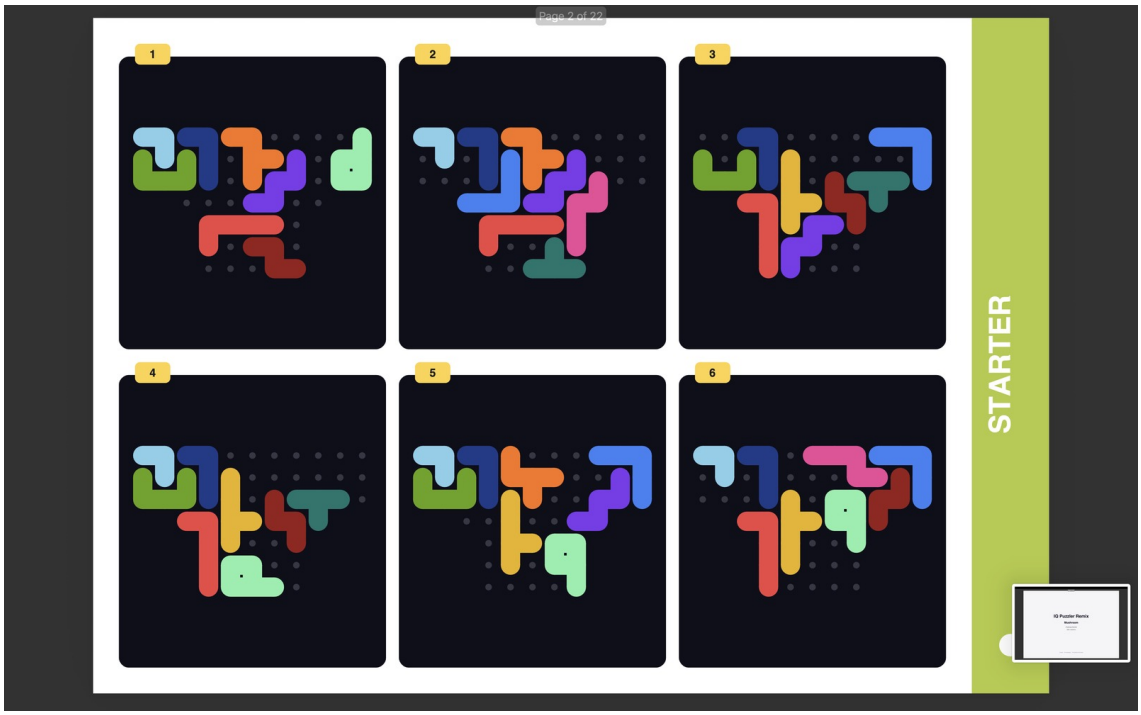

Figure 40. Starter section inside the generated booklet.

Explanation: Starter pages turn generated challenges into something that could be printed and used away from the app.

The solutions divider keeps the printable document usable by separating challenge pages from answer pages. This mirrors the structure of a real puzzle booklet and prevents answers from appearing directly beside the challenges.

Figure 41. Solutions divider inside the generated booklet.

Explanation: The booklet includes a clear separation between challenges and solutions, matching the structure of a real puzzle booklet.

The solution page is backed by actual solver placements, not by manually drawn decoration. This is important because every exported challenge must have a proven completion before it can be trusted as puzzle content.

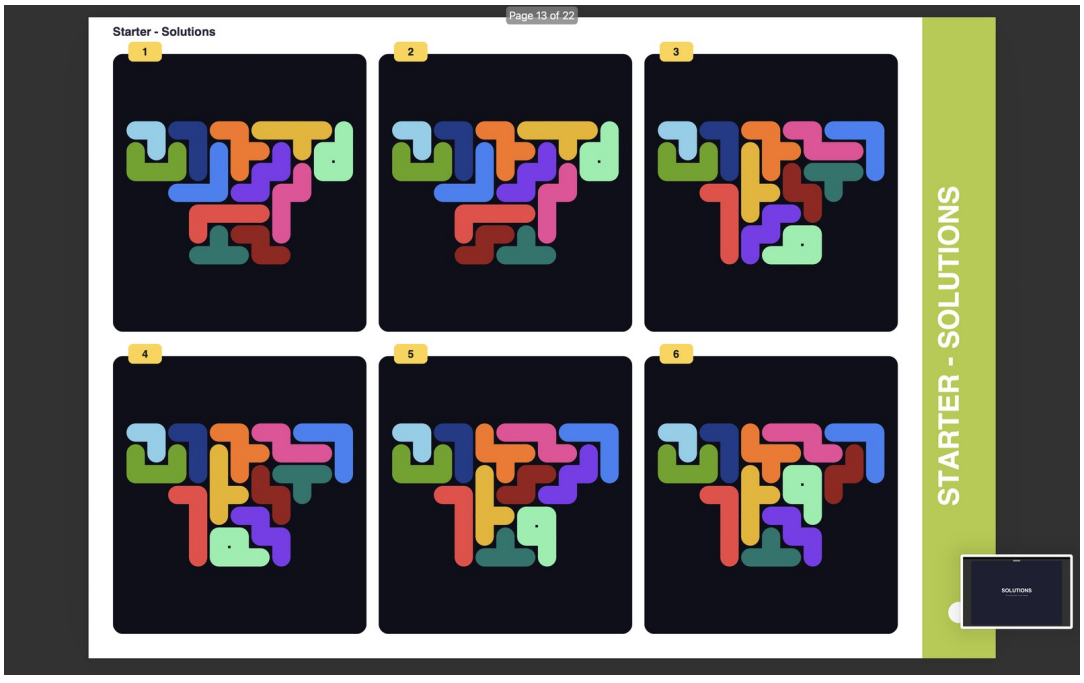

Figure 42. Generated solution page backed by solver output.

Explanation: The generated solutions prove that the booklet is not random decoration. Every shown challenge is connected to a valid solver result.

Shape Studio is where Remix becomes user-driven. The user paints a custom mask, the app tracks the selected cells and validation only begins when the shape reaches the required 55 cells.

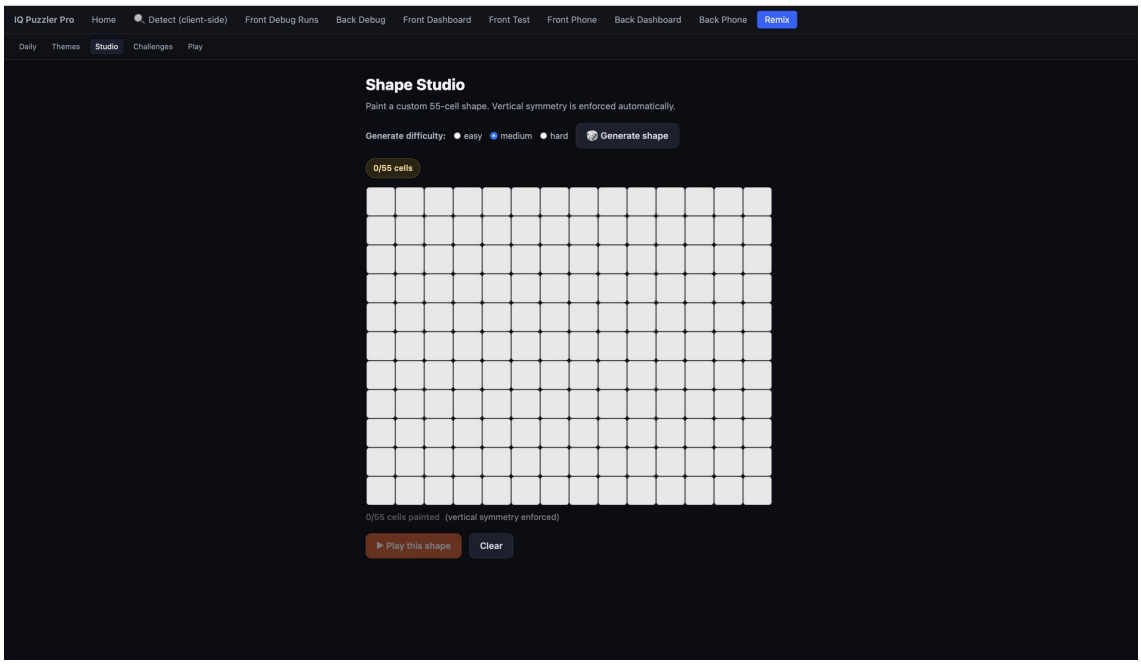

Figure 43. Shape Studio editable grid.

Explanation: Shape Studio lets the user define a custom 55-cell outline. This moved the prototype from fixed themes to user-created shapes.

A 55-cell shape is not automatically a good puzzle, so the Studio sends the mask to the validator. Only when the solver finds a tiling does the app enable play, which prevents impossible shapes from reaching the player.

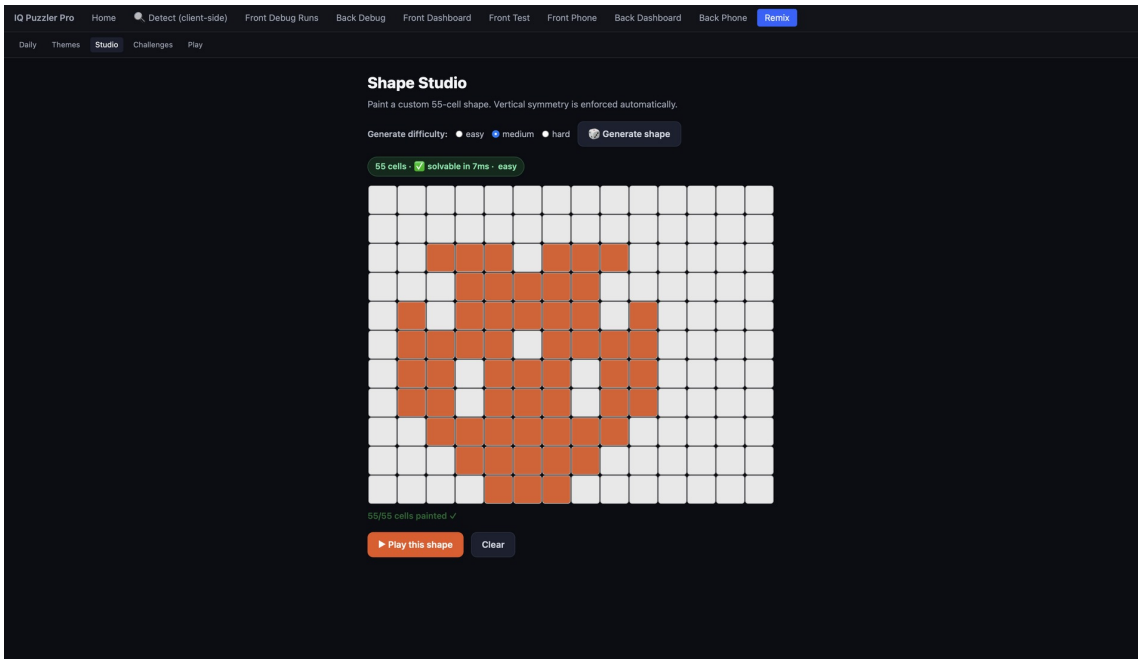

Figure 44. Shape Studio validating a custom shape as tileable.

Explanation: A shape can contain 55 cells and still be impossible. The validator prevents users from starting an unwinnable board.

The invalid-state screen shows the protective side of the generator. It is better to reject a visually interesting shape early than to let a player spend time on a board that the solver cannot tile.

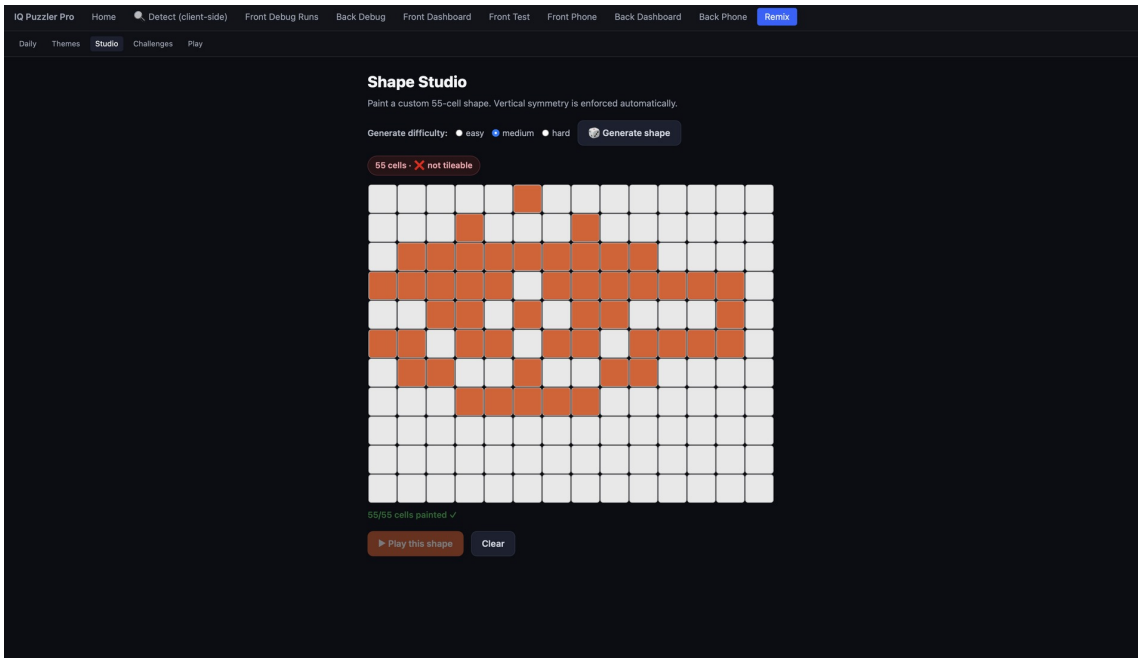

Figure 45. Shape Studio rejecting an invalid shape.

Explanation: This screen shows that validation is strict. It rejects shapes before play begins, which protects the player experience.

The generator can also create shapes automatically using a selected difficulty. A hard result means the shape and starting information leave more reasoning work for the player, so difficulty is connected to solver behaviour and not only to visual appearance.

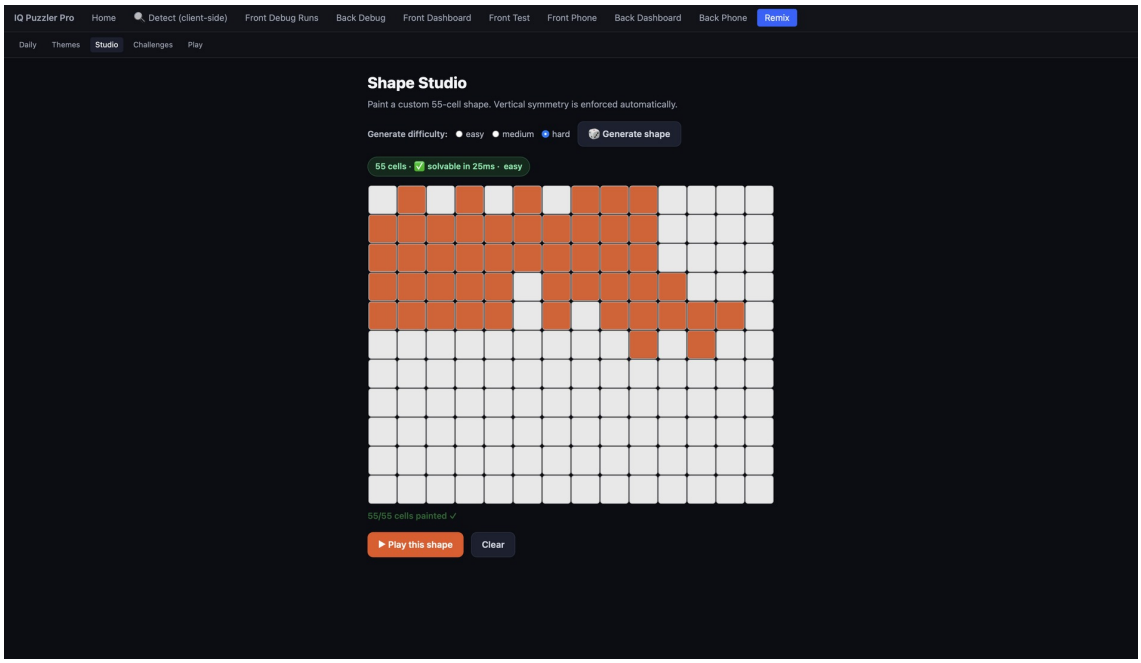

Figure 46. Algorithm-generated Remix shape with hard difficulty.

Explanation: This generated shape demonstrates the difficulty system. A hard board gives less starting support and requires a deeper solve path.

Once a custom mask passes validation, the same Play screen can load it through the encoded shape link. This proves the Remix interaction model is reusable: daily shapes, theme shapes and hand-made shapes all use the same mask-aware board logic.

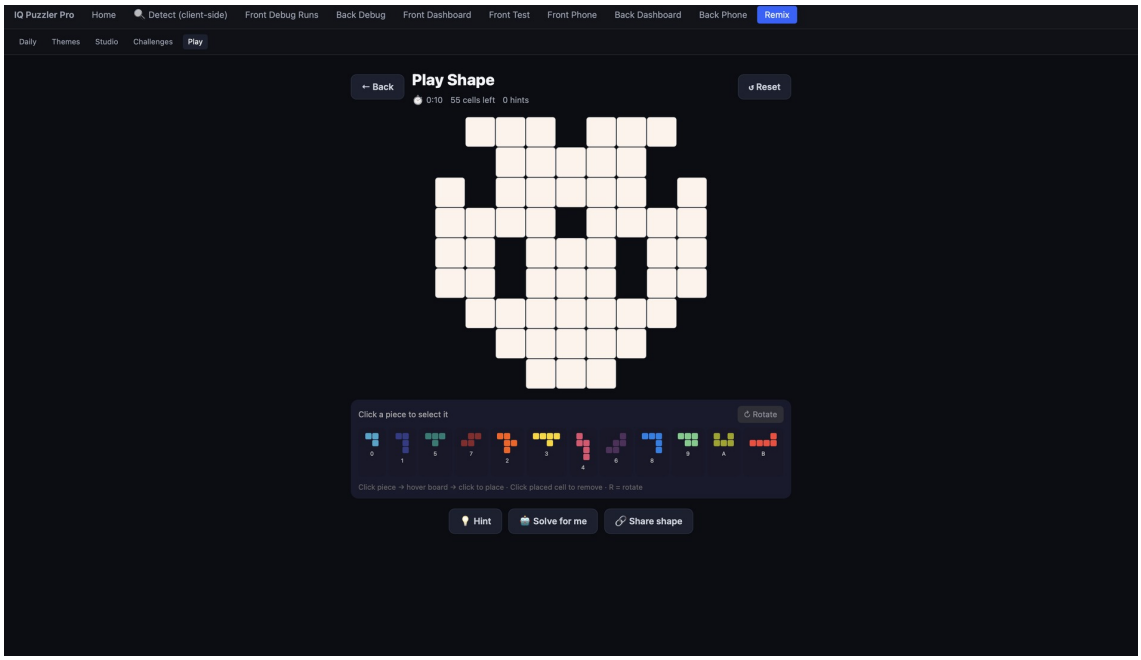

Figure 47. Custom shape opened in a playable state after validation.

Explanation: After validation, the generated outline becomes playable with the same piece palette and interaction model.

The custom-shape hint proves that the hint engine follows the current mask. It asks the solver for a valid next placement using the active cells of this generated shape, which is why the feature still works away from the original board.

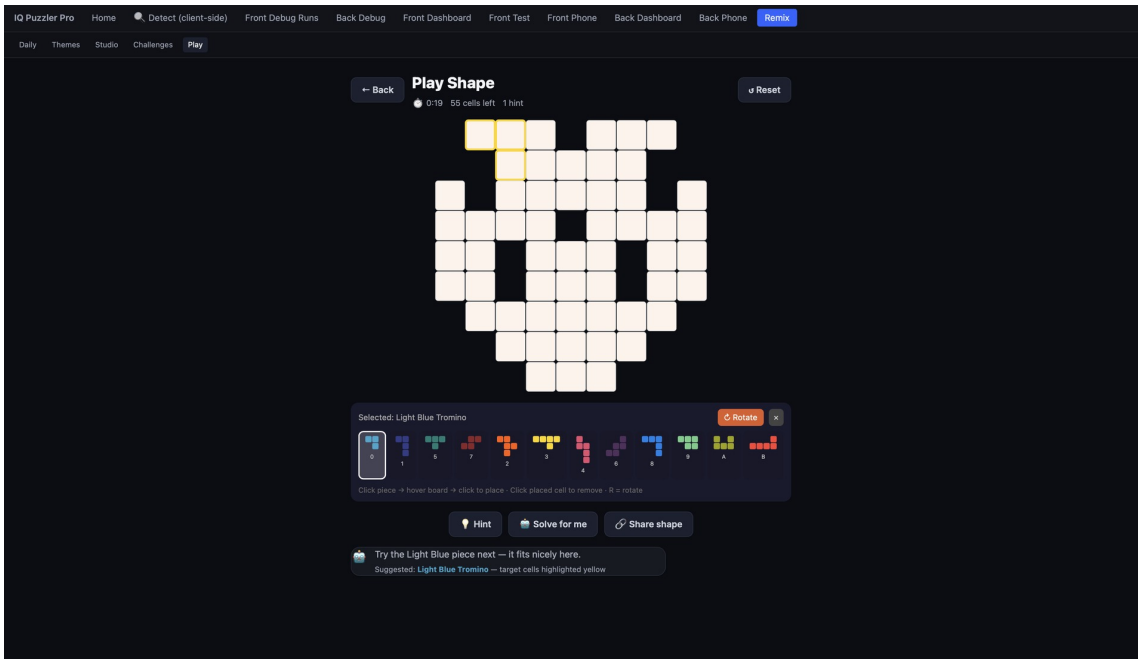

Figure 48. Hint on a custom Remix shape.

Explanation: This proves the algorithm works beyond the original rectangular board: the hint engine still uses exact cells and valid placements.

The STL export translates the validated mask into a physical board with real well spacing and depth. This step connects the web prototype to manufacturing exploration, because the downloaded mesh is meant to fit the existing IQ Puzzler pieces.

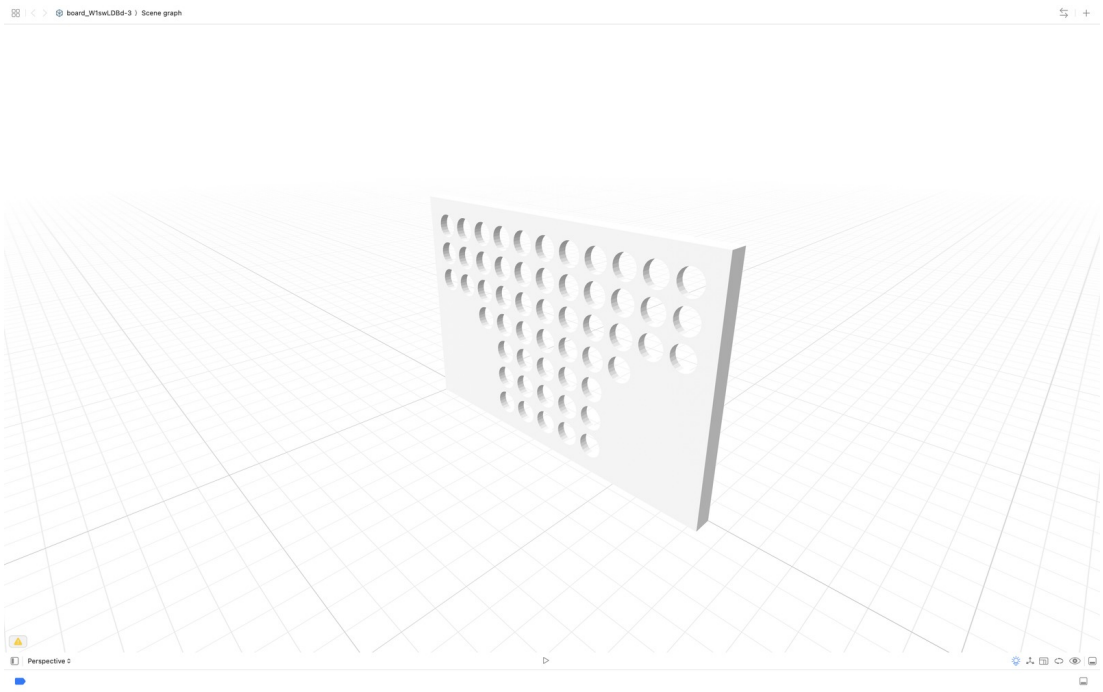

Figure 49. STL result after pressing Download STL in Remix.

Explanation: The STL export uses well depth and spacing designed for the real pieces. This is the bridge from digital generator to physical prototype.

The printed board was the physical test of the export assumptions. The well coordinates and spacing were based on earlier Blender calibration work, so the print checked whether the digital measurements were accurate outside the screen.

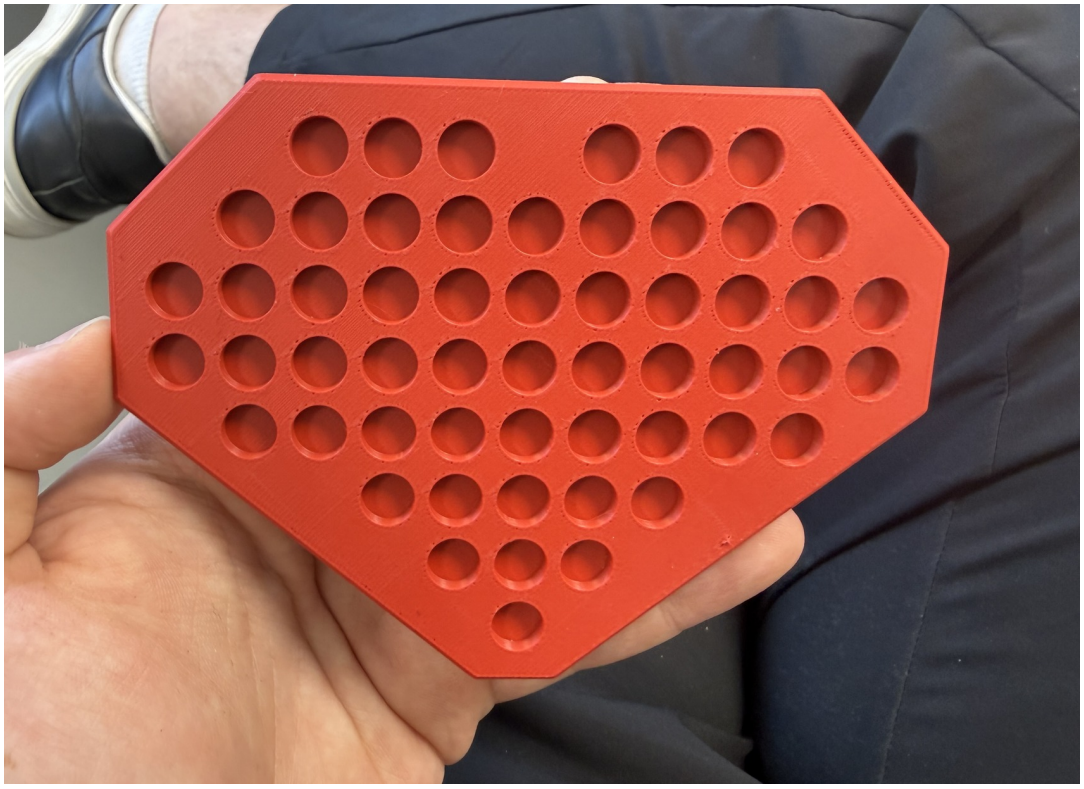

Figure 50. 3D-printed Remix board created from the export direction.

Explanation: This printed board proves that the digital shape can become a physical board. The coordinates and spacing came from the earlier Blender calibration work.

Filling the printed board with real pieces validated the practical value of the calculations. After the print finished, the pieces fitting correctly showed that the generator, STL export and calibration values were precise enough for a physical prototype.

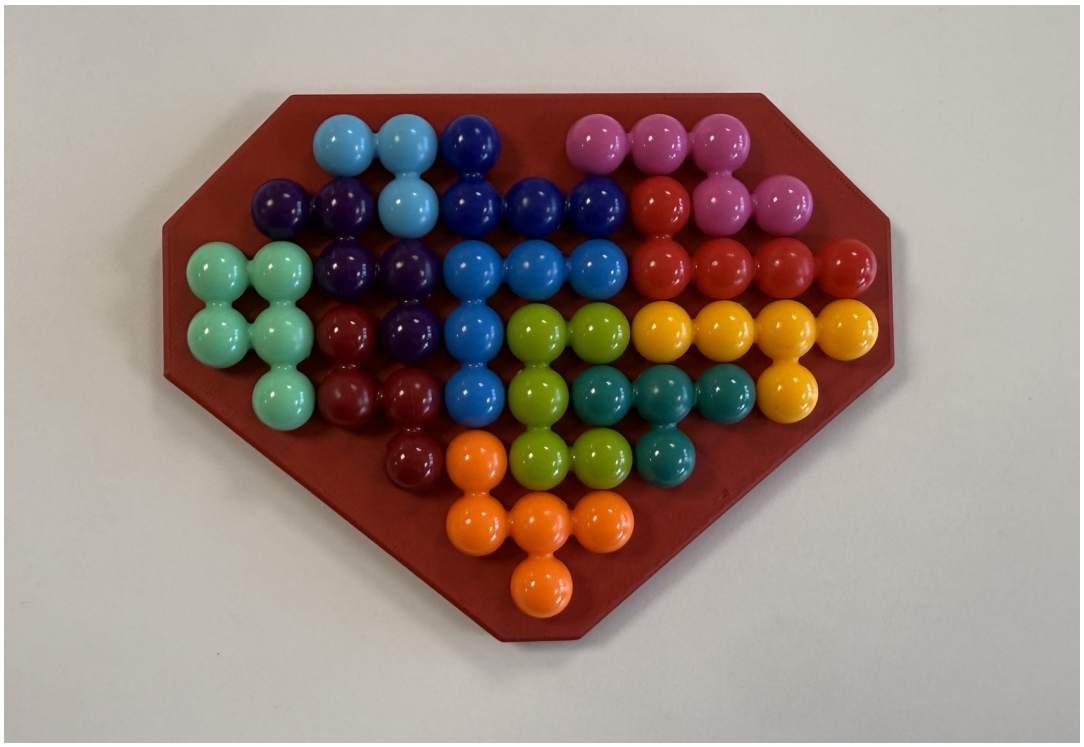

Figure 51. Printed Remix board filled with IQ Puzzler pieces.

Explanation: After roughly two hours of printing, the pieces fit correctly. That was an important proof that the digital calculations were precise enough for a real object.

3.4. Data, training and delivery

Synthetic data and real photos were both necessary. Synthetic renders gave scale, controlled variation and automatic labels. Real photos corrected the domain gap caused by glare, camera noise, plastic reflections and messy room lighting.

The Blender workflow was used because the model needed many examples of boards that would be too slow to label by hand. In Blender, the board and pieces can be placed programmatically, the camera can be moved, lighting can change, backgrounds can vary and the labels can be exported automatically. This gave the model examples of empty boards, partial boards, full boards, difficult colour combinations and different perspectives.

The real-photo workflow served a different purpose. Real images from the phone and Roboflow annotations exposed the imperfections that synthetic renders cannot fully reproduce: reflections on plastic, shadows from the board edge, small blur, camera compression and uneven indoor lighting. The final training direction therefore combined synthetic scale with real fine-tuning data instead of choosing only one source.

The final model direction therefore used YOLO v26 nano as the practical production target: small enough to load quickly, but still capable of returning segmentation masks for the board and pieces. This choice supported the phone-driven capture flow because the user should not wait for a heavy model before receiving a board preview, validation result or manual-correction screen.

The data workflow was cyclical because one training run does not finish a vision product. Real photos exposed failures, Blender generated controlled variation, Colab trained or fine-tuned the segmentation model and the debug screens showed what needed to improve next.

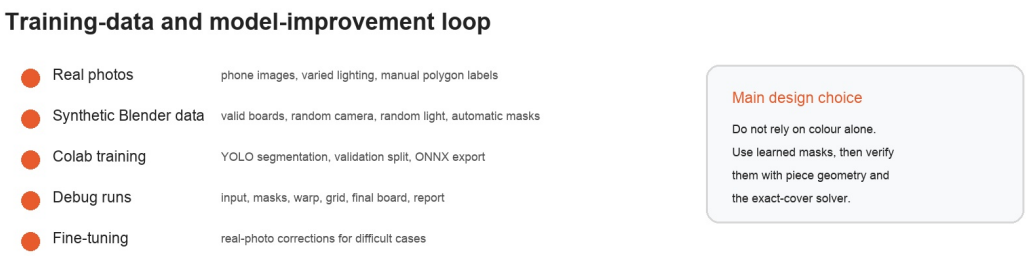

Figure 52. Data loop from real photos and synthetic renders to training, debug review and improvement.

Explanation: The loop explains the improvement process. Errors found in debug screens informed the next dataset or mapping change.

The Blender scene is the controlled environment behind the synthetic data. The script places the board and pieces, randomises camera, lighting and backgrounds, then exports labels that must match the class names and physical spacing used by the application.

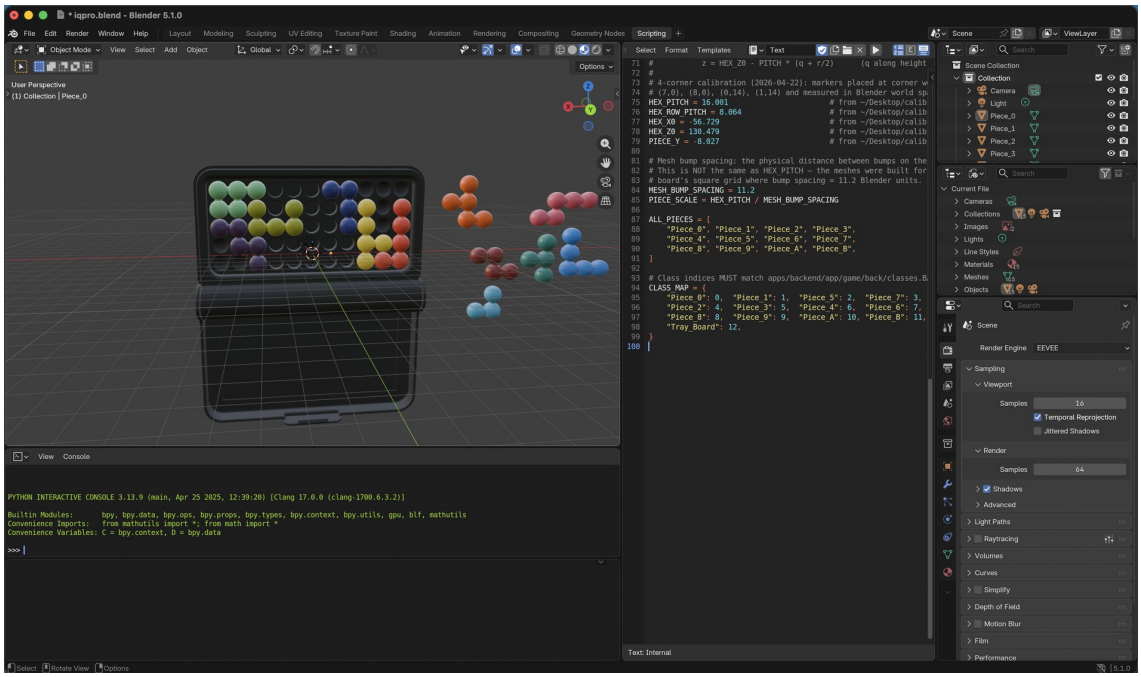

Figure 53. Blender scene used for synthetic data generation and calibration.

Explanation: This is the corrected Blender scene figure. It shows the IQ Puzzler board, placed pieces, calibration constants and class setup used for synthetic generation work.

The calibration values in the Blender scene were not decorative. They connected the synthetic world to the physical puzzle: well spacing, piece scale, board position and class names had to match the application logic. If those values drifted, the rendered labels could train the model on a world that did not match the real board. This is why the synthetic-data scripts and the board solver had to be checked together.

Dataset or package	Verified content	Reason it mattered
Front real photos	About 200 annotated real board photos	Closed the gap between rendered data and real phone images.
Front synthetic set	Generated images with random camera, lighting, fill level and backgrounds	Provided enough masks to train repeatably without hand-labelling thousands of images.
Front fine-tune package	Synthetic data combined with real Roboflow annotations and board-mask class	Improved board localisation and piece mapping on real photos.
Back real fine-tune package	127 train, 36 validation and 15 held-out test images	Adapted the diamond-board model to real examples.
Debug run archive	Original images, masks, warps, overlays and mapped states	Made failed detections explainable instead of mysterious.

3.5. Critical technical reflection

The hardest technical period was mapping detected pieces onto the digital board correctly. I spent several weeks on cases where the model output looked good but the board state was still wrong. The solution came from reading the debug output carefully, trying different mapping assumptions and asking for help when I was blocked.

A mistake I corrected during this period was assuming that a visually good detection meant the computer-vision part was finished. In reality, the harder problem was the translation from pixels to puzzle rules. I learned to ask more precise debugging questions: Did the model detect the right piece? Did the board warp place the wells correctly? Did the shape definition match the real piece? Did the solver reject the state because the mapping was wrong or because the board was genuinely impossible?

The internship also challenged my time management. The office was more than one hour away, and several technologies were new to me, especially Blender for synthetic data and some Unity-related exploration. I learned that professional progress is not only writing code quickly; it is also planning, asking for help earlier and communicating when an issue is risky.

Personally, the internship made me more open to working in an office environment. I became more comfortable discussing unfinished work, receiving feedback and explaining technical blockers. My classmates helped me when I was stuck, and the company environment helped me move through problems that I could not solve alone at first.

Summary
The realised project is a complete proof of concept: camera input, model masks, geometry

mapping, validation, hints, solving, debugging, synthetic data and Remix generation are connected into one explainable workflow.

4. Evaluation and Results

4.1. Evaluation method

I evaluated the project at three levels. Model quality checks whether the masks and classes are reasonable. Board reconstruction checks whether the photo becomes the correct 55-cell state. Usability checks whether the user can correct mistakes, request hints and continue playing without touching code.

Evaluation layer	How it was checked	What it proves
Model	Training runs, validation splits and visual inspection of masks	Whether piece and board masks are plausible.
Pipeline	Saved debug runs with original image, warp, grid and final labels	Whether the photo becomes the correct board state.
Solver	Validation and solve checks from partial states	Whether the board is logically completable.
Usability	Manual correction, hints, validate, solve and apply flows	Whether a player can recover from detection mistakes.
Remix	Shape validation, generated challenges, booklet and STL output	Whether generated shapes remain playable and exportable.

4.2. Results

Area	Verified result	Remaining risk or limitation
Front board	Photo pipeline, dashboard apply flow, hints, validation and solver are integrated.	Difficult lighting can still require manual correction.
Back board	Back phone flow, debug view, dashboard and solver use the square-diamond representation.	Dense or angled photos can still need review.
Training hardware	Colab training prepared around 640 by 640 YOLO segmentation workflows.	Exact final model metrics should be added if exported from Colab logs.
Synthetic data	Blender generation with randomisation and automatic labels is available.	Synthetic data must be balanced with real photos.
Remix	Daily, themes, Shape Studio, hints, booklet output and STL direction are implemented.	The prototype needs product evaluation before commercial use.

4.3. Validation checks

The most important validation was not one metric but the combination of visual and logical checks. If a model mask was good but the solver rejected the state, the result still needed correction. This made the app more robust than a pipeline that trusts confidence values alone.

5. Conclusion

The internship succeeded because the project became more than a model demo. It became an integrated product prototype that links physical puzzle play with computer vision, solver logic and a usable interface. The front-board system proved the complete recognition flow. The back-board system showed how important correct geometry is. Remix showed that the same solver ideas can generate new physical puzzle concepts.

The final value of the project is the connection between the physical and digital world. The software does not replace the toy. Instead, it can recognise a real board, help when the player is stuck, validate whether a position still makes sense and even explore new physical board shapes through Remix. That connection is what made the work meaningful for Smart NV and for me as an intern.

The most important technical lesson is that AI should not be trusted alone in a rules-based product. The model can propose masks, but deterministic board geometry and the exact-cover solver must verify the result. This combination made the system easier to debug, explain and improve.

The most important personal lesson is that difficult projects require communication. There were weeks where I was stuck on the same issue, especially with mapping pieces to the digital board. I improved by reading more carefully, trying smaller changes, using debug evidence and asking my company supervisor and classmates for help.

5.1. Recommendations and future work

- Export final model metrics from Colab and add them to the evaluation table.
- Collect more real photos under difficult lighting and phone angles.
- Continue improving manual board interaction for children and non-technical users.
- Evaluate Remix with real players before deciding whether it should become a product direction.
- Keep the confidential side project documented separately without exposing names or internal details.

6. Use of Generative AI Tools

6.1. Tools used

I used generative AI tools as a development assistant during the internship. They helped with code exploration, debugging ideas, document wording, Blender script generation and explaining possible approaches. The project decisions and final validation still had to be checked by me through running the application, reviewing debug outputs and testing with real photos.

6.2. Representative uses

Use	How output was evaluated
Debugging and code explanation	I compared suggestions with the actual repository and tested changes locally.
Blender/synthetic-data scripts	Generated ideas were checked by rendering images and inspecting labels.
Document drafting	Text was revised against coach feedback, project facts and confidentiality limits.
Reflection support	Personal details were supplied by me and rewritten into a professional tone.

6.3. Limitations

Generative AI can propose incorrect assumptions, especially about code it has not inspected or metrics that were not exported. For that reason I did not invent final model metrics. Where exact numbers were not available, the report describes the evaluation method and remaining work honestly.

7. Reference list

Ultralytics. YOLO documentation. https://docs.ultralytics.com/

FastAPI. FastAPI documentation. https://fastapi.tiangolo.com/

React. React documentation. https://react.dev/

Vite. Vite documentation. https://vite.dev/

Blender Foundation. Blender Python API documentation. https://docs.blender.org/api/current/

Roboflow. Roboflow documentation. https://docs.roboflow.com/

Google Colab. Colaboratory documentation. https://colab.research.google.com/

SmartGames. IQ Puzzler Pro product context and physical puzzle rules. https://www.smartgames.eu/

8. Attachments

8.1. Attachment A: technology stack

Layer	Technology	Role
Backend	Python, FastAPI	Photo upload, detection routes, solver routes and Remix routes.
Computer vision	OpenCV, NumPy, YOLO v26 nano segmentation	Board localisation, masks, perspective warp and cell mapping with a lightweight model choice.
Machine learning	Ultralytics YOLO v26 nano, PyTorch, ONNX	Segmentation training, export and inference with fast loading and smaller model size.
Frontend	React, Vite, TypeScript	Dashboards, phone clients, debug views and Remix UI.
Synthetic data	Blender Python	Rendered training data and automatic labels.
Solvers	Exact-cover search	Validation, solving, hints and shape checks.
Exports	PDF, SVG, STL utilities	Remix booklet, board outline and printable assets.

8.2. Attachment B: important folders

Folder	Purpose
apps/backend/app/api	FastAPI routers for photo, detection, solver and Remix endpoints.
apps/backend/app/cv	Computer-vision helpers, YOLO loading, board localisation and mapping.
apps/backend/app/game/front	Front-board pieces, board model and exact-cover solver.
apps/backend/app/game/back	Back-board square-diamond mask, piece shapes and solver.
apps/backend/app/game/remix	Mask-aware Remix solver, generator, cache, exports and theme shapes.
iq-puzzler-pro/frontend	Unified React/Vite application with the user-facing routes.
fine_tuning	Dataset preparation, Colab training and model installation scripts.
company_delivery	Clean package prepared for company handover.

8.3. Attachment C: glossary

Term	Meaning in this project
Board state	The 55-cell logical representation used by detection, dashboard and solver.
Mask	Pixel-level object outline returned by the segmentation model.
Warp	Perspective-corrected board image used for grid mapping.
Review state	A detection result that is shown to the user for manual correction before being trusted.
Exact cover	The solver problem where every target cell must be covered once by one piece.
Synthetic data	Automatically generated Blender training images with labels.
Fine-tuning	Additional training with real photos to reduce the synthetic-to-real gap.