SOMA FIGURE SOLVER

Blake Masters, Matt Acarregui, Rodrigo Canaan (Instructor)
CSC 481 - Cal Poly,
San Luis Obispo, United States
{bemaster,macarreg,}@calpoly.edu

***Abstract*—This project presents an interactive graphical user interface (GUI) for visualizing and solving Soma Cube puzzles in addition to abstract grid-based piece generation. Users can select pieces and use controls to fit soma pieces within the selected model in order to solve for a solution. The application includes backend logic to check submitted configurations against a growing set of previously discovered solutions, enabling detection and prevention of duplicate entries. Users can define a 3D puzzle grid of customizable dimensions and upload rule files that specify the quantity and size of polycube components. The system dynamically carves unique puzzle pieces from the grid space based on these rules, using rotational normalization to ensure that each piece is geometrically distinct. Within the interface, users can manipulate, place, and orient pieces in a real-time 3D environment to construct solutions. Additionally, the system integrates with an external solver to support automated solution validation. Designed to support both educational exploration and formal analysis, this framework facilitates the study of spatial reasoning, tiling constraints, and combinatorial structure within abstract volumetric puzzles.**

I. INTRODUCTION

The Soma Cube is a three-dimensional solid-dissection puzzle invented by Danish polymath Piet Hein in 1933 while attending a lecture by Werner Heisenberg on quantum mechanics [1]. It comprises seven unique pieces formed from three or four unit cubes and can be assembled into a 3×3×3 cube in 240 distinct solutions, excluding rotations and reflections [2].

We approached this project with the goal of designing a graphical user interface (GUI) for the Soma Cube puzzle, allowing users to construct, visualize, and verify valid solutions in a 3D environment. While the classic 3×3×3 Soma Cube has a well-documented and finite solution set, we recognized it as an ideal starting point for a knowledge-based system due to its constrained complexity and strong foundation in combinatorial reasoning. Our aim was to build a system that could both generate and evaluate solutions by bridging intuitive visual manipulation with an underlying inference engine grounded in the formal rules of the puzzle.

To support this objective, we examined leveraged an existing Soma solver and reverse-engineered their formats to develop a consistent representation for input and output data. We focused on using a minimalist piece format that could be seamlessly translated to and from the canvas, enabling interactive placement and manipulation. This structure allowed us to concentrate on establishing a bi-directional interface between user actions and automated reasoning.

Although the broader societal applications of this work are limited, the project offered a rigorous and rewarding opportunity to engage with principles of knowledge representation and inference. Unlike most existing solvers, which operate entirely in the background, our system empowers users to actively construct and validate solutions through a hands-on interface. One of our primary contributions was extending a backend solver to track and store distinct solutions, comparing each new submission to a canonicalized solution set to avoid redundancy. This ensures consistency with traditional solvers while supporting user exploration of both known and novel configurations.

Throughout development, we leveraged core concepts from artificial intelligence to implement rule-based piece generation and solution checking algorithms. These systems enforce domain constraints and incorporate validation logic to handle edge cases, ensuring robust behavior under variable grid sizes and user-defined rules. Ultimately, our project serves as an interactive and educational platform for exploring spatial reasoning, while demonstrating how knowledge-based systems can be applied to abstract problem-solving in a constrained domain.

II. RELATED WORK

While a number of Soma Cube solvers exist online, few offer the level of visual clarity or interactivity provided by our design, but rather use minature internal structure locked to a 3x3x3 grid represented in python lists [3]. Early in the development process, we investigated various existing implementations in hopes of finding a well-documented rule set and a straightforward representation of puzzle pieces suitable for knowledge-based modeling. Instead, most solutions relied on automated algorithms that brute-force placements until a solution is found, or they simply displayed precomputed answers using static 3D visualizations.

This observation inspired an effort to bridge the gap between visual interfaces and solver logic. The goal was to create a modular architecture that could not only represent piece placement intuitively but also integrate seamlessly with an inference engine. In doing so, the project could transition from a basic command-line utility into a fully interactive experience within a web-based canvas.

Among the tools explored was YASS (Yet Another Soma Solver), a C++ solver developed by Mark R. Rubin [4]. YASS provides a robust command-line interface capable of processing input shapes defined in a simple text format and solving them by recursively placing the standard seven Soma pieces in unique arrangements. The ease of executing YASS with configurable flags made it an ideal candidate for backend integration. By treating YASS as a programmable engine, we were able to visualize and design clean API endpoints that communicate effectively with our frontend system.

In essence, this project transforms a powerful yet text-based solver into a user-friendly graphical interface, enabling real-time interaction, shape generation, and solution validation within a highly modular and extensible architecture.

The format for the input shapes is as follows:
'*' and 'o' represent to-be-filled spaces. This is where it will try to fit the pieces. '.' represents unfillable spaces. These are just to fill in space to determine dimensions, but they also allow the program to handle separated puzzles (ones with gaps between them). 'c, t, l, z, p, n, 3' are the 7 unique puzzles which the program will place in designated areas.

Here is an example for a pyramid which can be constructed using the solver (left is input shape, right is output format):

```
. . * . . — . . t . .
. *** . — . z t t .
***** — n n t p p
. *** . — . n l p .
. . * . . — . . l . .

. . . . . — . . . . .
. *** . — . z z c .
. *** . — . 3 c c .
. *** . — . n l p .
. . . . . — . . . . .

. . . . . — . . . . .
. . * . . — . . z . .
. *** . — . 3 c c .
. . * . . — . . l . .
. . . . . — . . . . .
```

With this, we can convert our 3D grid into .json format, rearrange it so that it matches our input shape obtained from the figures folder, then feed this grid state to the backend to see if there is a solution. This will let users place blocks wherever they choose, check for a solution, and keep track of each shapes' solutions. In theory, this would also allow us to generate hints for the users' current grid state, but this proved to be more difficult than the project team was able to handle given the shorter timeframe and smaller team. This could be used as a build upon for future groups, where some features still desired such as hinting or abstract solution generation make good challenges.

III. METHODS

The intended algorithmic solutions can be found in detail in subsections below. *Something about solution checking here*. The puzzle piece generation algorithm relies on a head first DFS approach, akin to that of maze solvers

Describe in as much detail as you can fit into the report.

A. *Solution Checking*

The solution checking algorithm determines whether a given 3D grid state forms a valid and new (unsolved) solution for a particular Soma cube figure.

First, the algorithm verifies that all pieces are correctly placed: filled cells must match allowed positions from the original figure, and all required positions must be covered.

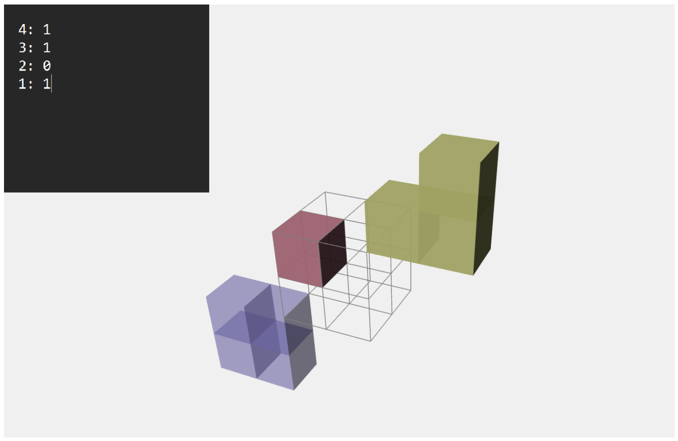
If valid, the grid is transformed into a 3D array and normalized by testing all possible 3D rotations. The smallest canonical form is used to eliminate duplicate orientations.

This normalized configuration is compared to previously found solutions. If no rotation matches any existing entry, the current state is recorded as a new solution.

By leveraging the current YASS solver with some new rules, we can ensure that only correct and unique solutions are stored, enabling efficient tracking of all valid configurations.

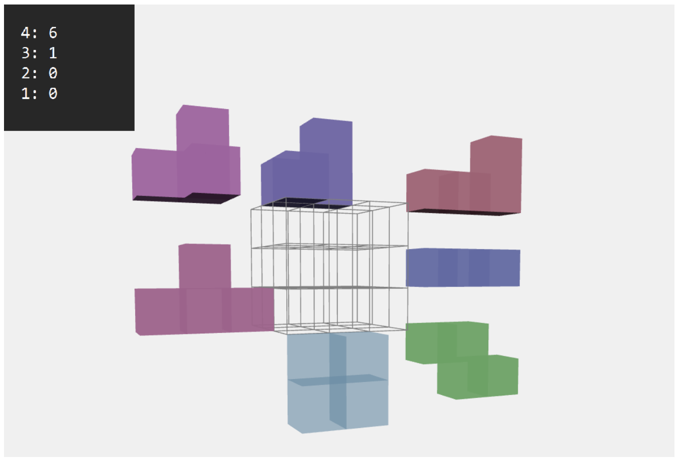
Algorithm 1 Soma Cube Solution Checking

```
Require: Grid state, shape ID, check-only flag
Ensure: Validity, novelty, normalized form, and total count
 1: Call HANDLESOLUTION($ID, gridState, checkOnly$)
 2: function HANDLESOLUTION($ID, solution, checkOnly$)
 3:     if not ISVALIDPLACEMENT($solution, ID$) then
 4:         return (False, False, None, 0)
 5:     end if
 6:     normalized $\leftarrow$ NORMALIZESOLUTION($solution, ID$)
 7:     solutions $\leftarrow$ LOADSOLUTIONS($ID$)
 8:     isKnown $\leftarrow$ normalized $\in$ solutions
 9:     if not isKnown and not checkOnly then
10:         Add normalized to solutions
11:         Save solutions to disk
12:     end if
13: return ($\mathbf{true}, \neg isKnown, normalized, count$)
14: function ISVALIDPLACEMENT($solution, ID$)
15:     Check if pieces are only in allowed cells ($*$ or $o$)
16:     Check empty cells (.) match original
17:     Ensure all required cells are filled
18:     return validity
19: function NORMALIZESOLUTION($solution, ID$)
20:     Convert to 3D grid
21:     Apply all 3D axis rotations
22:     Return minimal (canonical) representation
23: function LOADSOLUTIONS($ID$)
24:     Load JSON of existing normalized solutions
25:     return solution set
```



```
4: 1
3: 1
2: 0
1: 1
```

Fig. 1: 2×2×2 Generation



```
4: 6
3: 1
2: 0
1: 0
```

Fig. 2: 3×3×3 Generation

B. Puzzle Piece Generation

The piece generation algorithm attempts to divide a 3D grid of dimensions X×Y×Z into a set of connected polycube shapes (pieces) such that:

Each piece matches a specific size as defined in a set of rules (e.g., 3 pieces of size 4, 1 piece of size 3).

All pieces tile the entire grid without overlap or unfilled space.

Each generated piece must be unique up to 3D rotation.

The process involves recursively finding the first empty cell in the grid and generating all possible connected shapes of the required size from that cell using depth-first search (DFS). To avoid duplicates, each shape is normalized and checked across all valid 3D rotations. The algorithm continues until all required pieces are placed or it determines that no valid carving is possible.

Algorithm 2 Soma Cube Piece Generation

```
Require: Grid size $(X, Y, Z)$; Rules $\{(s_i, c_i)\}$ mapping size to count
Ensure: Dictionary of size $\rightarrow$ list of unique pieces or None
 1: Initialize empty 3D grid of size $X \times Y \times Z$
 2: Flatten rules into list $S = [s_1, \dots, s_n]$ with $c_i$ copies of each $s_i$
 3: if $\sum S \neq X \cdot Y \cdot Z$ then
 4:     return None
 5: end if
 6: Call PLACEPIECE$(0, \{\})$
 7: function PLACEPIECE$(i, \texttt{usedKeys})$
 8: if $i \geq$ length$(S)$ then
 9:     return true
10: end if
11: $size \leftarrow S[i]$
12: $start \leftarrow$ FINDFIRSTEMPTY()
13: for all $shape \in$ GENERATECONNECTEDSHAPES$(start, size)$ do
14:     $key \leftarrow$ canonical key of $shape$
15:     if $key \in$ usedKeys then
16:         continue
17:     end if
18:     Place $shape$ in grid; add $key$ to usedKeys
19:     if PLACEPIECE$(i+1, \texttt{usedKeys})$ then
20:         return true
21:     end if
22:     Remove $shape$ from grid; remove $key$ from usedKeys
23: end for
24: return false

25: function GENERATECONNECTEDSHAPES$(start, size)$
26: Use DFS to build connected polycube shapes of given size from $start$
27: Normalize each shape under all valid 3D rotations
28: Return unique list of shapes
```

C. Piece Generation Images

Puzzle piece generation examples for 2×2×2 and 3×3×3 cubes. The requirement text file appears in the upper left corner of each screenshot.

The runtime displayed show two legal sets. During the presentation, we were asked to use the following ruleset for a theoretical 2x2x2 solution:

3: 2

2: 1

1: 0

This ruleset presented a challenge at the time, where pieces rotated about the Y-axis were not considered to be the same, resulting in generation of a puzzle piece set that contained two of the same size puzzle piece. This is theoretically not possible for a unique puzzle piece solution, and has since been fixed. This set has become the defacto-insolvable but legal requirements testing set, relying on two stages of canonically similar piece removal, with the first after each piece is generated, and a final check with the entire set before returning the pieces to the user.

IV. Evaluation / Results

To evaluate our system, we used a combination of functional validation and rotational equivalence testing across several puzzle configurations. The primary criteria for success included:

- **Correctness of solution checking:** We made sure to convert solutions into a format that could be verified with the backend solver. This included rearranging the 3D array in all ways to ensure all valid solutions were checked while maintaining knowledge of duplicates.
- **Known-solution checking:** The system must identify whether the solution is unique by checking against a file of previously discovered solutions.
- **Rotation normalization:** The solution must not be accepted more than once if it is geometrically equivalent (via rotation) to a previously submitted solution.
- **Grid Validation:** Each initial step in generating a puzzle piece is responsible for identifying and ensuring the current grid availability can begin to form a new piece.
- **Piecewise Construction:** By identifying open neighbors, piece DFS paths become limited to what is available in the grid.
- **Unique Pieces (1):** After creating a piece based on the largest size available in the requirements, that piece is rotated and checked canonically against existing pieces in the set.
- **Unique Pieces (2):** After a set is created, the original API call double checks the set for any canonically congruent pieces before shipping the set out to users.

Testing involved manually placing known valid configurations into the grid and submitting them through the interface. After a submission, the system was expected to:

1) Add the normalized form to the stored solution set if it was valid and unique.
2) Reject the solution as a duplicate if submitted again (even from a different camera angle or rotation).
3) Accept distinct valid shapes, while ignoring cosmetic variations caused by piece reorientation.

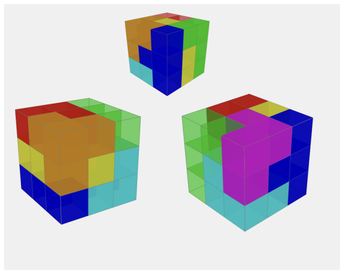


Fig. 3: Similar and unique solutions

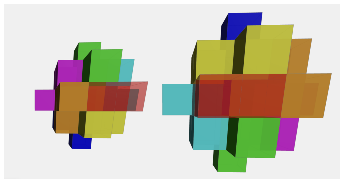


Fig. 4: Similar pyramid rotations

We confirmed the solution checking pipeline by examining both console outputs and the saved solutions file. Figure 3 illustrates how the same solution was normalized and rejected on repeated entry due to rotational equivalence. The solutions on the left and right are the same solution, but flipped. The solution above shows a unique solution. All of these behave exactly how we intend, indicating to the user that it is either a new solution or known solution.

We can also observe this with the pyramid that we discussed earlier. Figure 4 are two similar solutions which behaved as intended. We accepted the first one, and rejected the next as it is the same solution seen at a different angle.

The system's correctness and normalization functionality were further tested by rotating known solutions along different axes before submission. The normalized form remained consistent across submissions, confirming that duplicate rejection was functioning as intended.

Although the program is currently limited to figures provided by the YASS submodule, it can be extended to solve any well-formed Soma figure, provided the input is properly formatted. A valuable external resource for exploring such figures is the Bundgaard gallery of Soma Cube configurations [5] which we could use to create convert all figures to YASS format in order to solve it with our program.

In future work, we aim to streamline the integration of these figures by building a parser or GUI connector that automatically converts user-selected figures from that site into valid YASS input format. Alternatively, the system could incorporate a curated database of canonical Soma challenges to support broader exploration directly within the interface.

V. Conclusions

This project aimed to transform the traditionally static experience of solving Soma Cube puzzles into an interactive, extensible, and knowledge-driven graphical interface. By integrating a frontend 3D environment with a backend reasoning system, we allowed users to generate puzzle pieces, build solutions, and verify submissions against a growing collection of known configurations.

The project demonstrates how classical artificial intelligence/knowledge-based concepts, such as rule-based inference, structured search, and geometric normalization, can be applied to domains involving spatial logic. We developed a solution validation pipeline that first verifies logical correctness and then checks whether a submission is novel by comparing it to canonicalized versions of existing solutions.

One of our main achievements was connecting abstract puzzle state generation with the YASS backend solver through a custom API. This integration provided a formal mechanism for solution verification while enabling users to interact visually with puzzle logic. Our generation algorithm proved flexible across different grid sizes and rule sets, ensuring only valid and distinct pieces are created and returned.

While the system successfully addressed our primary goals, some limitations remain. The current implementation does not support hint generation or interactive solver feedback during piece placement. These features offer meaningful directions for future development, particularly in support of real-time puzzle-solving assistance and deeper inference capabilities.

We also recognize the limited (though vast) set of solutions

In summary, this project bridges spatial reasoning with interactive design and formal validation. It not only replicates the functionality of previous solvers but also improves accessibility and understanding through visualization and guided user interaction. The resulting system provides a foundation for future research into knowledge-based systems applied to constrained logical environments such as the SOMA cube or other constrained puzzles.

REFERENCES

[1] P. Hein, "Invention of the soma cube," Discussed during lecture by Werner Heisenberg, 1933, as cited in Martin Gardner's Scientific American column.

[2] M. Gardner, "Mathematical games: The soma cube," *Scientific American*, p. 65, 1961, describes origin and enumerates 240 solutions.

[3] Hickford, "some-cube-solver," GitHub repository, 2023, https://github.com/hickford/soma-cube-solver.

[4] M. R. Rubin, "Yass: Yet another soma solver," GitHub repository, 2021, https://github.com/thanks4opensource/yass.

[5] S. Bundgaard and B. Bundgaard, "All known soma cube figures," https://www.fam-bundgaard.dk/SOMA/FIGURES/ALLFIGS.HTM, 1999, accessed: 2025-06-13.