Generating and solving 3D nonogram puzzles

Connor Halford

University of Abertay Dundee

School of Arts, Media and Computer Games

May 2016

I. Abstract

Nonograms are a type of visual logic puzzle where the player must use numeric clues to deduce which cells of a grid to fill in. Once the grid has been configured such that none of the clues are contradicted the puzzle is solved and the player will have drawn some recognisable image in the grid. These puzzles can be extended to 3D so that rather than filling in squares in a grid the player is removing cubes from an orthogonal field. Solving the puzzle essentially involves carving out a voxel model from an initial solid block.

This project sets out to generate solvable 3D nonograms by voxelizing existing polygonal models. Voxelization results in a voxel model that serves as the solution to a puzzle, from which numeric clues can then be generated. To show that these are genuine puzzles solvable only using logical deduction a series of 3D nonogram solvers were developed which use the same deductive reasoning that human players do. These solvers extend existing work on solving 2D nonograms by adding support for 3D puzzles. Different approaches to the order in which logic is applied to the puzzle were created and a quantitative comparison of their effectiveness was carried out.

The results of this project show that voxelization can indeed be used to generate solvable 3D nonogram puzzles. The strengths and weaknesses of this approach are discussed as well as how voxelization and automatic 3D solving can be used as tools to aid human puzzle designers. The results of comparing the different solver approaches show that applying logical deduction to the puzzle space in a similar order to how human players do is the most effective method for algorithmically solving a puzzle.

Keywords: nonograms, 3D puzzles, logical deduction, puzzle generation, puzzle solving, voxelization, algorithms.

II. Foreword

I would like to take this opportunity to thank the staff at the University of Abertay Dundee for all their support during this project, particularly Dr. Paul Robertson for his constant encouragement and advice.

I would also like to thank Mrs. Valerie Sturrock, Dr. John McDermott and Prof. Peter Cameron of the University of St. Andrews for their help in mathematically describing one of the many problems I had to solve during this project. I would also like to thank Abertay University's Dr. Karen Meyer and Dr. Craig Stark for putting me in contact with them.

Finally I would like to thank my friends for keeping me sane through the many hours spent moving blocks around a spreadsheet and mumbling about algorithms, especially David Ferguson and Rachel Crawford.

III. Contents

IV. List of Tables

V. List of Figures

1. Introduction

1.1 Nonograms

Nonograms (also known as griddlers, Japanese crosswords, hanjie or picross) are a type of grid-based logic puzzle solved with deductive reasoning. The player is presented with a white grid of squares with sets of numbers along the left and top edges. Each number represents how many consecutive cells in a line should be filled in, i.e. the length of the contiguous block of cells in that line. If there are multiple numbers it means that there are multiple contiguous blocks of those lengths in that order, with one or more empty cells left between them. The goal is to find the solution to the puzzle: the single configuration of the grid wherein none of the numeric clues are contradicted, and in so doing the player reveals an image of some kind in the grid. Figure 1 shows an example nonogram and its solution.

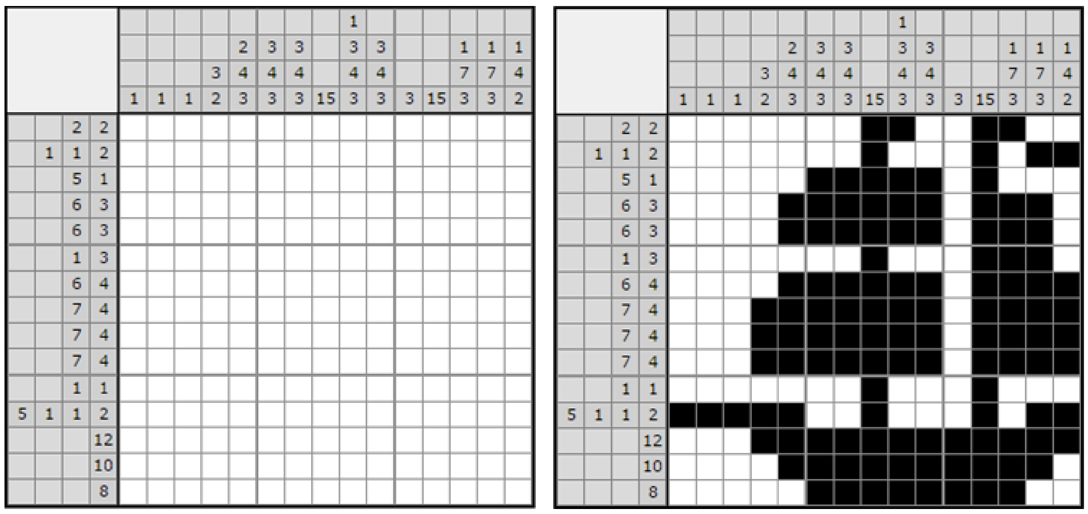


Figure 1: A 15×15 example puzzle (left) and its solution (right) (Chugunny 2016a).

Nonograms were created by Non Ishida in 1987 and were later named after her. Ishida started publishing puzzles in 1988 and the British newspaper *The Sunday Telegraph* started printing weekly puzzles in 1990 (which continue to this day). From there the puzzles grew in popularity and are now commonly found in newspapers, magazines and puzzle books worldwide.

Nintendo began releasing a series of nonogram video games in Japan in 1995, with the first international release being *Mario's Picross* (1995) for the Game Boy. With *Picross DS* (2007) they saw critical and commercial success and continued to update the game with bi-weekly puzzles. In *Picross 3D* (2009) Nintendo created the first 3D nonograms and went on to release a sequel with *Picross 3D 2* (2015). Rather than filling in squares in a grid, 3D puzzles ask the player to remove cubes from an orthogonal field.

There are two important distinctions to be made between 2D and 3D nonograms. The first is that 2D puzzles were popularised in print format before transitioning to games, but 3D nonograms can only practically be distributed as software and so only exist as such. The second is that the clues act differently. In a 2D puzzle if there are multiple blocks there will be multiple clues, but in a 3D puzzle only the sum of the clues will be given instead. If there are two contiguous blocks on the line there will be a circle around the clue, if there are three or more there will be a square. Many lines won't have a clue at all. A typical 3D puzzle can be seen in Figure 2.

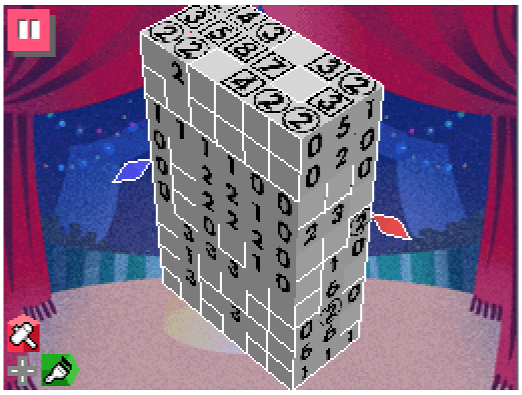


Figure 2: Screenshot from *Picross 3D* (2009) showing the differences between 2D and 3D puzzles.

Being able to automatically find the solution (or lack thereof) for a puzzle is a great aid in the puzzle design process. As Wolter (2013a, section 5.3.3) puts it "*Puzzle designers need to solve their puzzles many times as they make small changes in hopes of improving the playability. Doing this manually is extremely tedious, and error prone. … [Automatic solving] is designed to solve cooperatively with the puzzle designer, allowing the puzzle designer to assess how well the puzzle solves and how difficult it is without having to solve it entirely by hand.*" Thus it would be very beneficial to the puzzle creation process to have an algorithm that can solve 3D nonograms automatically.

Much work has been done on writing 2D puzzle solving algorithms due to the established popularity of the puzzles and the interesting nature of the challenge. Many 2D solvers exist, from assembly-language programs for graphical calculators (Vauter 2014) all the way through to complete puzzle creation tools with built-in solvers to support the puzzle designer as mentioned above (Wootten 2012). No work could be found in extending these solvers to support 3D puzzles.

1.2 Voxelization

Voxelization is the process of taking a polygonal model and approximating it as a set of orthogonal cubes. Figure 3 gives an example (Westerdiep 2010).

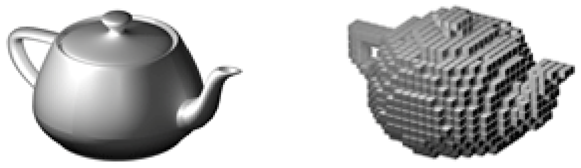


Figure 3: Rendering the Utah teapot using polygons (left) and voxels (right).

The most important difference between polygonal and voxel models is that a polygonal mesh only represents the visible *surface*, whereas a voxel model represents a complete *volume*. This makes them suited for different purposes. It is far less computationally expensive to render a polygonal model than a voxel one and graphics hardware has been designed for years with polygons in mind. There are therefore significantly more polygonal models than voxel models. Voxels are generally used when computation on spatial information from the scene is required, such as for volumetric light shafts (Pestana 2014) or fluid dynamics (Evans and Kirczenow 2011). This means that voxel models are generally created at run time by voxelizing an existing polygonal scene, rather than being created specifically as voxels. There are of course exceptions to this, with games such as *Voxatron* (2011) and *Resogun* (2013) working entirely in voxels for rendering, gameplay and physics.

1.3 Research Question

By solving a 3D nonogram the player is essentially manipulating a voxel model. Voxelizing existing polygonal models is a cost-effective way to quickly create a large variety of voxel models by utilising existing assets. The goal of this project is to explore the combination of these two ideas. Put simply:

How suitable is voxelizing existing polygonal meshes as a method of generating solvable 3D nonogram puzzles and how can 2D solving techniques be applied to 3D puzzles?

A valid puzzle should be solvable only using logical reasoning and should have only one solution, which will ideally be some sort of recognisable object or shape. By creating a 3D puzzle solver the validity of the generated puzzles can be assessed. Ultimately this project aims to show how voxelizing existing polygonal models and having an algorithmic 3D puzzle solver could benefit the 3D puzzle creation pipeline.

This question leads to the following set of objectives:

- Create an application which allows users to load in polygonal models and voxelize them at a user-defined resolution in order to create 3D nonograms
- Algorithmically attempt to solve these 3D nonograms in a variety of ways to show whether there is a solution and roughly how difficult it was to find, to further support the puzzle designer in their creation process
- Assess whether the voxelization process can consistently create solvable 3D puzzles and how effective it could be as a tool for puzzle designers
- Assess the effectiveness of the 3D puzzle solvers and discuss how they could be improved

2. Literature Review

2.1 Types of puzzles

There are black-and-white and colour variants of both 2D and 3D puzzles, as seen in Figure 4. The differences will be briefly explained here. This project focuses on 3D black-and-white puzzles.

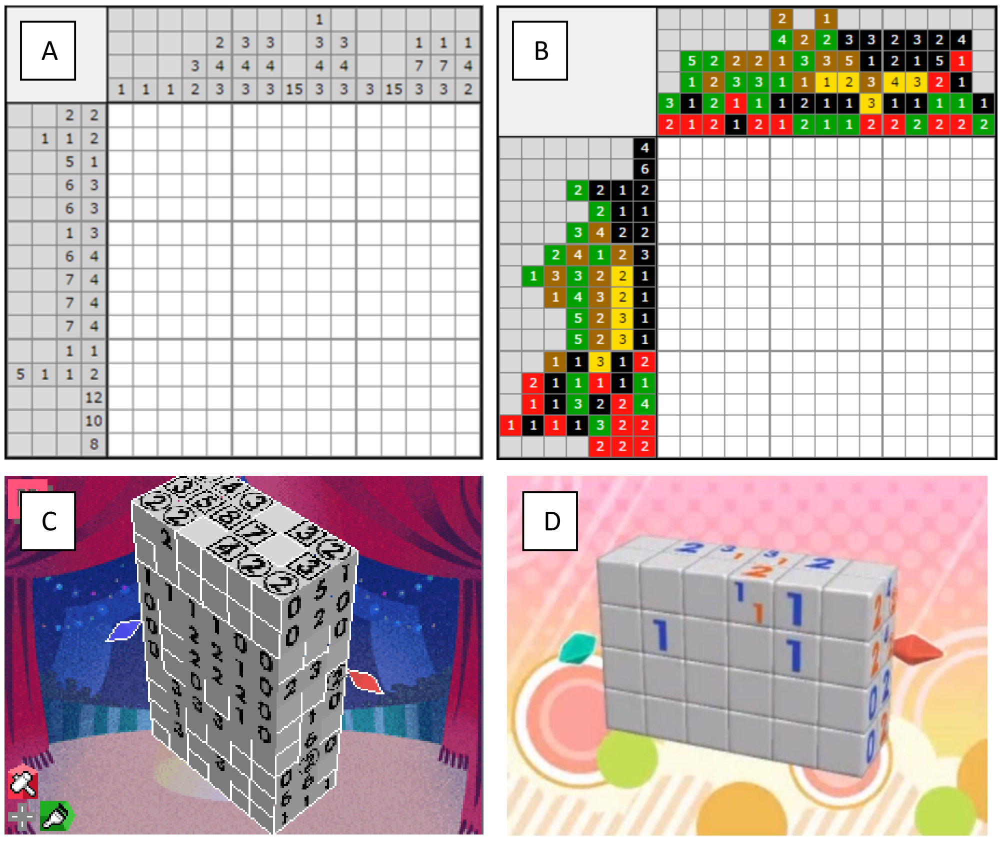


Figure 4: Types of nonogram. A) 2D black-and-white, B) 2D color, C) 3D black-and-white, D) 3D color (Chugunny 2016a; Chugunny 2016c; *Picross 3D* (2007); *Picross 3D 2* (2015))

The majority of work in algorithmically creating and solving puzzles has been for the 2D black-and-white type. Here the numbers indicate how many consecutive black cells to fill in each line, where multiple numbers mean multiple blocks with at least one white cell between them. Typically every line has clues and completely empty lines are very rare.

In 2D colour puzzles the coloured numbers indicate how many consecutive cells of that colour to fill in each line, where multiple numbers mean multiple blocks. There must be at least one white cell between consecutive blocks of the same colour (otherwise it would just be one larger block) and there can be *zero or more* white cells between blocks of different colours. Typically every line has clues and completely empty lines are very rare.

In 3D black-and-white puzzles the clues indicate the total number of cells in that line to fill. If the number is plain then there is one contiguous block of that length. If there is a circle around the number then there are exactly two blocks whose lengths sum to the clue, with at least one empty cell between them. If there is a square around the number there are *three or more* blocks whose lengths sum to the clue, with at least one empty cell between each consecutive pair of blocks. Many lines don't have clues and many of them are completely empty.

3D colour puzzles are very new, having first been created for *Picross 3D 2* which released in October 2015 and is currently only available in Japan. This has made fully understanding the specifics difficult. It combines the coloured clues of 2D colour puzzles with the plain/circle/square clues of 3D black-and-white puzzles. Unlike in 2D colour puzzles the ordering of the blocks isn't indicated by the clues because the same line can be viewed from either end in 3D space. It has been difficult to discover whether there can be empty cells between differently coloured blocks on one line; what little gameplay footage there currently is only shows differently coloured blocks being immediately adjacent. Many lines don't have clues and many of them are completely empty (in fact the game introduces a new mechanic for automatically clearing all empty lines).

2.2 Existing solvers

It has been shown that determining whether a nonogram has a solution and finding what that solution is are both NP-complete problems (Ueda and Nagao 1996; van Rijn 2012). This means that it is possible to verify a given solution in polynomial time, but that the time taken to actually find a solution increases very quickly with the size of the puzzle for all currently known algorithms.

There is an important distinction to be made between solving and validation. Solving a puzzle involves trying to find a solution and will either find one or prove that there aren't any. Validating a puzzle involves finding all solutions, whether that is zero, one or more; it is an exhaustive proof of how many solutions there are. Checking that a solution is unique is a substantially more difficult problem than just finding a solution. Both algorithms for solving and those for validation are collectively referred to simply as 'solvers'.

In an extensive survey of 2D solvers Wolter (2013a) found that the vast majority of solvers were only capable of solving black-and-white puzzles, though some were capable of solving colour puzzles too. It was also shown that all of the solvers shared to varying degrees the same handful of solving techniques, discussed in section 2.3.

Only one attempt prior to this project was found for solving 3D black-and-white puzzles but it cannot handle numbers with circles or squares and is therefore not a true 3D solver, though it is interesting in its approach. The solver treats black-and-white nonograms as an instance of the Boolean satisfiability problem and uses the Chaff algorithm to find a solution, based on a set of propositional logic sentences used to represent a puzzle (Johnston 2014). It therefore appears that this project is the first attempt at algorithmically solving true 3D black-and-white puzzles. No public work on solving 3D colour puzzles could be found.

2.3 Solver techniques

The most effective strategy for finding the state of cells is to use the same logical rules that human players do. While some solvers use an explicit set of rules (Ortiz-García et al. 2007b; Yu, Lee and Chen 2011), the general approach is to encapsulate them all into a linesolver algorithm. A linesolver operates on a single line of the puzzle and deduces as much as possible given the current state of the line and its set of clues. Deductions are made by comparing the leftmost and rightmost possible solutions for the line (technically the extreme solutions - the linesolver is independent of line direction but for simplicity's sake they will be referred to as leftmost and rightmost). Finding the leftmost and rightmost solutions for a line is generally considered the most difficult part of this approach (Wolter 2012); this project details two straight-forward algorithms for finding them. This project also explores different approaches for the order in which linesolvers are applied.

Not all puzzles are solvable purely through linesolving (i.e. they are not line-solvable) though the vast majority are. As Wolter puts it "*we are solving puzzles designed by humans for humans to solve, and those just aren't usually that difficult. Nobody designs nonograms on 10,000 by 10,000 grids because nobody wants to solve them*" (2013a, section 6). This project limits itself to line-solvable puzzles because the ultimate goal is to aid the creation of puzzles for human players.

When linesolving fails there is essentially only one thing left to do: guess (though there are many ways to do this). By guessing the state of a cell then using linesolvers to continue down that path it is possible that a contradiction will be found. In that case any deductions made since the guess must be undone and the guessed cell can definitely be said to be the opposite state of what was guessed. This process is variably known as chronological backtracking (Yu, Lee and Chen 2011), bifurcation (Simpson 2000) or simply guessing (Wolter 2012) and could take the form of a depth-first search. A process known as probing can make the guessing more effective by guessing every possible state of a set of candidate cells selected based on the state of their neighbours, assigning values to each based on how much further the linesolvers got before stalling again, then actually making the guess which resulted in the most progress (Pomeranz, Raziel and Rabani 2010; Wolter 2012).

As well as being useful as a method of guessing once logic fails, a pure depth-first search approach can also be used to entirely solve a puzzle (Simpson 2000). Another approach is to use evolutionary or genetic algorithms (Batenburg and Kosters 2004) to mutate an invalid solution towards a valid one. It has been shown that a genetic algorithm can out-perform a DFS, though they are prone to getting stuck in local optima (Wiggers 2004). Other approaches have included integer programming (Goldner, Kimura and Schwartz [no date]), a learning algorithm using simulated annealing (Wang and Tang 2014) and the aforementioned Boolean satisfiability problem (Johnston 2014).

2.4 Limitations of the linesolver technique

Although a linesolver deduces everything possible when considering a single line, there are many strategies that advanced players use which are still logic-based but that require considering multiple lines at once. One technique known as 'smile logic' will be briefly described here for example purposes; Wolter ([no date]) gives a much more exhaustive list.

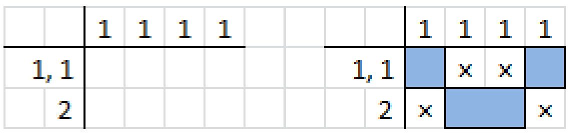


Figure 5: An example of smile logic.

The key here is that there can only be one solid in each column. This means the blocks in the rows can never overlap vertically, and since the two blocks in the top row must have a space between them it forces the solution shown on the right, where the blocks from each row alternate. This puzzle cannot be solved by only considering one line at a time.

There are other, far more abstract kinds of logic that human players can use by considering an entire puzzle at once or looking for patterns in the clues. Algorithmic solvers tend to struggle on problems which require such lateral thinking. One example is the *n-dom* series of puzzles (Wolter 2013b).

2.5 Puzzle Creation

2.5.1 Creating 2D puzzles

There has been much work on using image processing to turn an input image into a 2D black-and-white puzzle (Ortiz-García et al. 2007a; Batenburg et al. 2009; Henstra 2011; Goldner, Kimura and Schwartz [no date]). The common general method is explained in Figure 6.

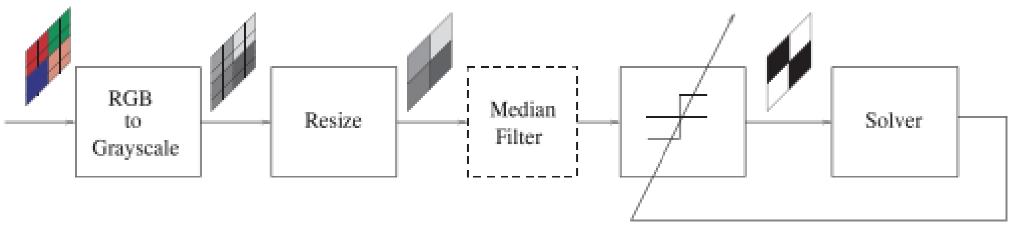


Figure 6: Stages of the 2D black-and-white puzzle generation process (Ortiz-García et al. 2007a).

First the colour input image must be converted to greyscale then resized to match the desired puzzle bounds. Filtering can then optionally be applied to remove noise while preserving edges. A puzzle is then constructed using a threshold value: cells below a certain level are considered black and those above are considered white. This is run through a solver to check that it is a valid puzzle with only one solution. Based on the results of the solver the puzzle can then be iteratively modified either because it wasn't solvable or because multiple puzzles are to be generated. There are many modifications that can be made and in various ways, as detailed in the references. These modifications are often centred on the removal of white cells as they are the main source of insolvability (Ortiz-García et al. 2007a).

The process for generating 2D colour puzzles is somewhat similar:

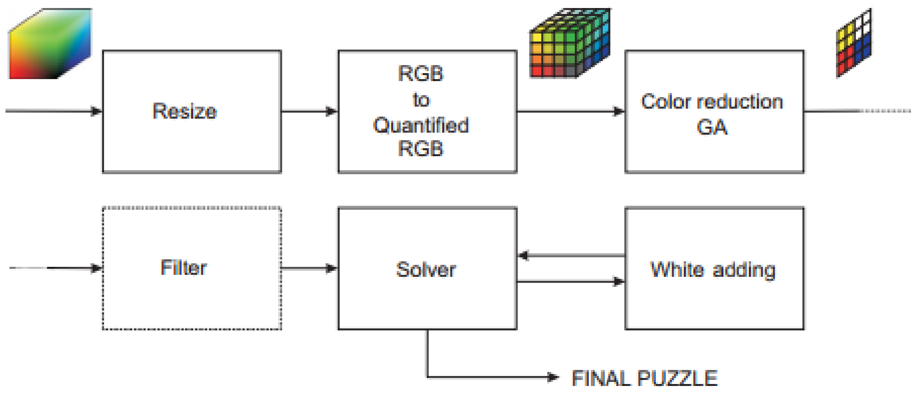


Figure 7: Stages of the 2D colour puzzle generation process (Ortiz-García et al. 2007a).

As with black-and-white puzzles the first stage involves reducing the set of colours down to that of the final puzzle, in this case using a genetic algorithm. Again an optional edge-preserving filter can then be applied. The familiar loop of iteratively making modifications to the puzzle based on the results of a solver is then begun. This same loop is why automatic solvers are a great tool for puzzle designers as they allow them to make many small, iterative improvements without the need to solve the puzzle by hand after each change.

2.5.2 Creating 3D puzzles

No existing work on generating 3D puzzles or tools for users to create 3D puzzles could be found. The closest is *NonoBlock* (2014) though the game doesn't feature true 3D puzzles and instead uses layers of 2D puzzles to build up a voxel model one slice at a time. The game does allow users to design their own puzzles but again these are simply collections of 2D puzzles which when layered on top of each other produce a final voxel model, rather than having a voxel model as a single true 3D puzzle.

2.5.3 Voxelization

The problem of creating voxel models is almost as old as computer graphics itself. How to create voxel approximations of geometric shapes (Kaufman and Shimony 1986) and how to voxelize polygonal models (Huang et al. 1998) are important problems to solve and have been worked on by many people for many years.

Voxels have many uses, from advanced lighting effects like ambient occlusion (Takeshige 2015), volumetric light shafts (Pestana 2014) and shadow volumes (Eisemann and Décoret 2006), through to physics simulations such as rigid bodies (Cepero 2013) and fluid dynamics (Evans and Kirczenow 2011). Because of their usefulness there are now many methods for voxelizing a polygonal scene, including fully GPU-based approaches (Takeshige 2015; Eisemann and Décoret 2006).

The approach taken in this project focuses on raycasting (Thon, Gesquière and Raffin 2004). This approach was chosen both for its relative simplicity and because of how the method can be easily extended to support 3D anti-aliasing, so that higher quality voxel approximations of fine detail can be created. An overview of this method is given in section 3.1.1.

3. Methodology

This project was written in C++ and HLSL with DirectX 11. It uses the following third-party libraries:

- *Assimp* (Assimp Development Team 2014) for model loading
- *DirectX Tool Kit* (Microsoft Corporation 2016) for vector, matrix and raycasting maths
- *ImGui* (Cornut 2015) for the graphical user interface
- *IProf* (Barrett 2003) for performance profiling

3.1 Puzzle Generation

3.1.1 Voxelizing polygonal meshes using raycasting and quadtrees

The first step in generating 3D nonogram puzzles is the creation of a voxel model through the voxelization of a polygonal mesh. The used in this application is the raycasting-based approach detailed by Thon, Gesquière and Raffin (2004). This approach casts a ray through the centre of each line of voxels along one axis (in this case along the z-axis) and stores all intersections with the mesh in order of their depth. Once all intersections have been found, the position of the centre of each voxel along the line is compared with the set of intersections; if the number of intersections along the ray after this point is odd then the voxel is inside the model, if even then the voxel is outside (assuming a closed, continuous mesh). This is explained in Figure 8, taken from Thon, Gesquière and Raffin's paper (2004).

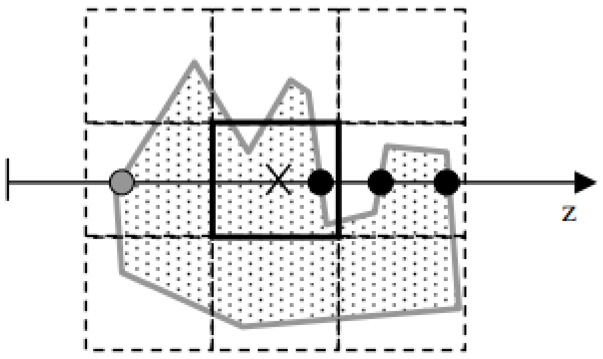


Figure 8: The number of intersections after a point determines the inside/outside state.

This raycasting approach can be significantly sped up by first partitioning the triangles in the mesh into a quad-tree, so that for any given ray it isn't necessary to look for intersections with every single triangle in the mesh, rather only those triangles in that general region. This is shown in Figure 9 (also from the 2004 paper). Here the dotted lines represent the voxels, with the centre of the one we want to cast a ray along being at (x_0, y_0). Only intersections with the triangles in the small region of the quadtree containing the point (x_0, y_0) need to be checked for.

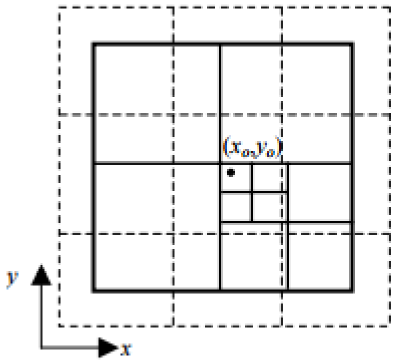


Figure 9: Determining which subset of triangles to check for ray intersections with.

In a typical quadtree implementation every element in the structure is referred to by only one leaf node; however for this application the elements of the quadtree are triangles and therefore represent an area instead of a single point. One triangle may therefore be referred to by multiple leaf nodes in the tree, as it must be referred to by every leaf node that the triangle overlaps. An example of this is shown in Figure 10. Once a leaf node has references to some arbitrary limit of triangles it will be subdivided into four smaller leaf nodes and become a branch node (as with traditional quadtree implementations only leaf nodes store references).

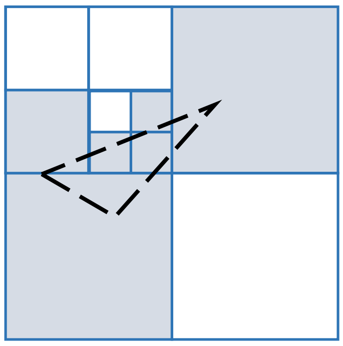


Figure 10: Triangles may overlap with and so be referred to by multiple leaf nodes in the quadtree.

3.1.2 Creating numeric clues

The act of voxelizing a polygonal model results in the solution of a 3D nonogram; the creation of numeric clues based on that solution constitutes the actual puzzle.

It is important here to remember the distinction between 2D and 3D nonograms. In a 2D nonogram every row and column has a sequence of one or more numbers representing the lengths of contiguous blocks of solid cells in that line; multiple numbers means multiple blocks with at least one space between them. It is essentially a form of discrete tomography: a representation of an image through the projections of that image along each axis. A player solves the puzzle by cross-referencing these two axes of information and making logical deductions based on the results. Players are always given all clues for all lines in the puzzle. It is possible to remove clues from a puzzle and have it still be solvable, although the removal of clues can also render a puzzle unsolvable as shown in Figure 11.

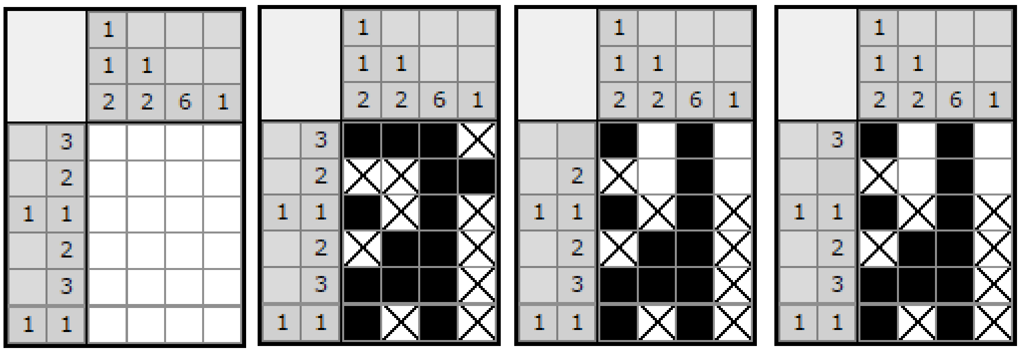


Figure 11: From left to right: a simple 2D puzzle (Chugunny 2016b); the solution; the puzzle is unsolvable without the top row clue; it is still solvable without the second row clue.

In a 3D nonogram there are three axes of information for the player to cross-reference and make deductions from. This amount of information makes 'solving' the puzzle trivial and uninteresting. Designers of 3D puzzles combat this by removing information in two ways: by removing clues and by combining clues. Figure 12 depicts a typical puzzle from *Picross 3D* (2009). Many of the lines in this puzzle have no numeric clue at all, while some clues are surrounded by either a circle or a square. A circled number indicates how many total cells in that line are solid and that these solids are split into exactly two contiguous blocks, although the lengths of these blocks aren't known to the player. For example a line with the clues '2, 3' in a 2D puzzle would become '⑤' in a 3D puzzle and could potentially be split into blocks of lengths '1, 4', '2, 3', '3, 2' or '4, 1'. The player must deduce not only which cells should be solid but also which set of blocks to expand this circular clue into. A number with a square around it means that the solids in that line are split into *three or more* blocks, so not only are the lengths of the blocks unknown but the quantity of blocks is unknown.

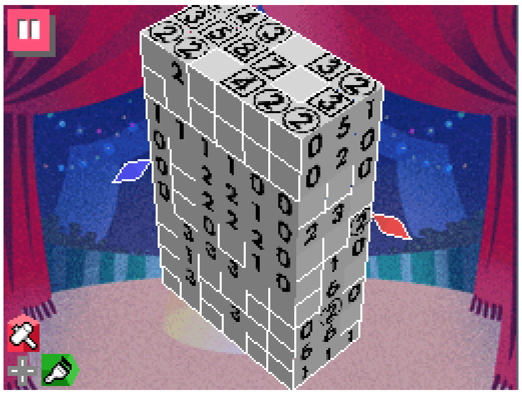


Figure 12: A screenshot from *Picross 3D* (2009) showing the different types of clues.

Generating the numeric clues for a 2D puzzle is simply a matter of listing the lengths of contiguous blocks of solids in each line (remembering that order matters). For a 3D puzzle these lengths must then be summed and if there were exactly two blocks it becomes a circle clue and if there were three or more blocks it becomes a square clue.

The puzzle generation process created in this project combines clues but doesn't remove them. The removal of clues is an important part of 3D puzzle design and can drastically change the difficulty of a puzzle. Unfortunately it was simply too much additional work for the timeframe of this project. Ultimately the goal of the puzzle generation and solving is to aid 3D puzzle designers and improve their workflow. Further development could integrate the work done here into a puzzle design tool which would then allow the designer to remove clues and check that the puzzle is still solvable through the use of one of the automatic solvers. Although the removal of clues would add a lot of complexity to the puzzle generation process it would add virtually none to the puzzle solvers since they operate on the puzzle one line at a time. Lines without clues would simply be skipped.

3.2 Cell state deduction techniques

There are two techniques for logically deducing state: one which finds solid cells and one which finds empty cells. Both operate on a single line and work by comparing the leftmost and rightmost possible solutions for the line - see section 3.4 for details on how to find these. Note that these techniques work irrespective of line direction, but for the purposes of explanation the discussion and diagrams here deal only with horizontal lines and name the extreme solutions 'leftmost' and 'rightmost'. Each row in a diagram is a line of a puzzle, with the set of clues on the left. The numbers across the tops of the diagram are cell indices, *not* perpendicular clues (as in a full 2D puzzle). Lines are dealt with individually.

There is considerable variety in the potential difficulty of nonograms. This project deals only with *simple* nonograms as defined by Batenburg and Kosters (2009). A 'simple' nonogram has a single solution which can be found through the application of logical rules to individual lines. This is generally the type of puzzle found in newspapers, puzzle books and video games. Although 'simple', puzzles of this type can pose a considerable challenge to human players. It is never necessary to guess the state of a cell when solving a simple puzzle. For non-simple puzzles it may be necessary to guess the state of a cell then make further deductions based on that guess and check for contradictions to known state; if a contradiction is found then all deductions based on the guess must be undone and the cell can definitely be said to be the opposite state of what was guessed (e.g. if guessing that a cell was solid led to a contradiction then that cell must in fact be empty). This is known as chronological backtracking (Yu, Lee and Chen 2011) and is a form of depth-first search, a technique which is easy for computers but difficult for humans. The solvers created in this application could easily be extended to support chronological backtracking and therefore be able to solve non-simple puzzles, but the focus here is to generate simple 3D puzzles and so this work was not necessary.

3.2.1 Finding solid cells

Consider the example of placing a block of length three into a five-wide line with no existing known state. There are three possible solutions, as seen in Figure 13.

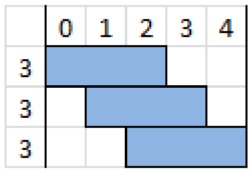


Figure 13: Possible solutions when placing a block of 3 into a 5-wide line.

The state of cell two can safely be said to be solid as it is solid in every possible solution of the line. Rather than having to compare every possible solution it is only necessary to compare the leftmost and rightmost possibilities as they represent the limits of the solution space for this line, effectively encompassing every intermediate solution. The detection of overlaps between the leftmost and rightmost solutions is how the location of solid cells is deduced. As later deductions are made in perpendicular lines the leftmost and rightmost possible solutions for any given line will change, resulting in more overlap and therefore more deductions, eventually converging on the single solution for the line.

Note that this method only works when it is the same block overlapping a cell. The first two rows of Figure 14 depict an overlap between the second block of the leftmost solution and the first block of the rightmost solution. Although there is an overlap, cell three is not guaranteed to be solid; the third row depicts a valid solution where cell three is empty. It is only when the leftmost and rightmost possible placements for a particular *block* overlap that deductions can be made.

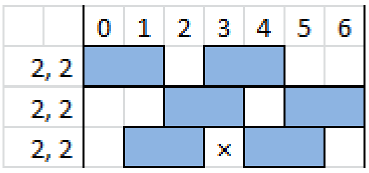


Figure 14: Overlap between different blocks does not guarantee solids.

Figure 15 gives a pseudocode description of this technique for finding solid cells.

```
for each block
    if leftmost and rightmost possible positions overlap
        overlap = leftmost end cell - rightmost start cell
        for i = 0 to overlap
            index = rightmost start cell + i
            cell[index] = solid
```

Figure 15: Pseudocode for finding solid cells.

3.2.2 Finding empty cells

The technique for deducing empty cells is similar to that for deducing solids; it works by comparing the leftmost and rightmost possible solutions for a line on a per-block basis. The first two rows of Figure 16 show the leftmost and rightmost solutions (in this case the only possible solutions) when placing two blocks of lengths two and one respectively into an eight-wide line, when cells two and six are already known to be solid.

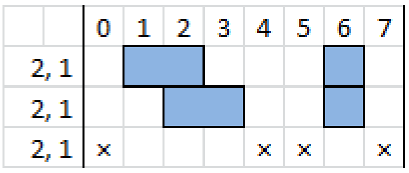


Figure 16: If a cell isn't solid in any possible solution then it must be empty.

The third row shows the empty cell deductions that can be made from this scenario. Cells zero, four, five and seven cannot possibly be solid in any valid solution of this line and therefore must be empty. This technique essentially finds cells which are outside the valid range of any block and deduces that they must therefore be empty, which includes the cells to the immediate left and right of a completed block.

Note that it isn't as simple as checking if a cell is empty in both the leftmost and rightmost solutions, as it could easily be solid in one of the solutions between them. Rather than compare the leftmost and rightmost solutions in their entirety comparisons must be made on a block-by-block basis. Figure 17 shows the leftmost and rightmost possible solutions when placing two blocks of length one into a seven-wide line.

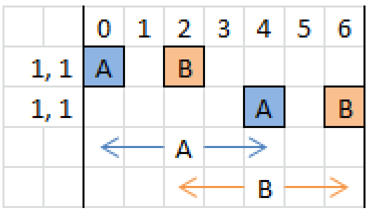


Figure 17: An example of the *range* of two blocks.

Although cells one, three and five are all empty in both the leftmost and rightmost solutions, no deductions can be made from this scenario; it is easy to picture numerous solutions filling each of those cells. By comparing the leftmost and rightmost solutions on a per-block basis we are effectively checking the possible *range* of those blocks, as shown above. If a cell is within the possible range of any block then there exists a valid solution where that cell is solid, therefore it cannot be guaranteed to be empty. Figure 18 describes this algorithm for finding empty cells using pseudocode.

```
for each cell
    empty = true
    for each block
        if cell is within block's possible range
            empty = false
            exit block loop
    if empty
        set cell to empty
```

Figure 18: Pseudocode for finding empty cells.

3.3 The linesolver algorithm

The combination of these two techniques for finding the solid and empty cells respectively is named the linesolver algorithm. The linesolver operates on a single line of a puzzle (irrespective of direction) and logically deduces the state of as many cells as possible given the current state of the line. The application of a linesolver to different lines of a puzzle will deduce information which then affects the possible solutions of the perpendicular lines, and so on and so forth until the puzzle is solved. Different methods for applying this linesolver are detailed in sections 3.6, 3.7 and 3.8.

The calculation of the leftmost and rightmost possible solutions can be skipped entirely if the state of every cell in the line can be deduced immediately from the set of clues. There are two scenarios where this can happen:

1. **The line has a single clue: zero**. This means there are no blocks on this line, so every cell can immediately be set to empty.
2. **The clues for the line offer a *complete description***. A line has a complete description when the set of blocks for the line can only fit if there is exactly one empty cell between each consecutive pair, without room for empty cells before the first block or after the last. Figure 19 gives an example of this, while Figure 20 describes the scenario mathematically.

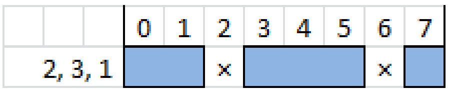


Figure 19: An example of a line with a complete description.

$$n + \sum_{i=1}^{n} b_i = L + 1$$

Figure 20: The definition of a complete description, where n is the number of blocks on the line, b_i is the length of the i^{th} block and L is the length of the line. For Figure 19 n = 3, b_i = 2, 3, 1 and L = 8.

Note that a completely solid line would also qualify as a complete description under this definition. With these two early-outs, the complete linesolver algorithm is as follows:

```
if the clue for the line is 0                           # whole line is empty
    for each cell
           set cell to empty
    return

if sum_of_clues + number_of_clues == line_length + 1   # complete description
    position = 0
    for each block
           for i = 0 to block_length           # place block
                  cell[position] = solid
                  position += 1
           if not last block                   # space between blocks
                  cell[position] = empty
                  position += 1
    return

calculate leftmost and rightmost possible solutions

for each block                                          # find solid cells
    if leftmost and rightmost positions overlap
           overlap = leftmost end cell - rightmost start cell
           for i = 0 to overlap
                  index = rightmost start cell + i
                  cell[index] = solid

for each cell                                           # find empty cells
    empty = true
    for each block
           if cell is within block's possible range
                  empty = false
                  exit block loop
    if empty
           set cell to empty
```

Figure 21: Pseudocode for the complete linesolver algorithm.

3.4 Building the leftmost and rightmost possible solutions

This section will detail how to build the leftmost possible solution of a line. Building the rightmost solution follows a very similar procedure but works from the other end of the line and with the blocks in reverse order. One way of creating the rightmost solution would be to reverse the line and the order of the blocks and then generate the 'leftmost' solution before reversing it back again.

Building the leftmost solution begins by placing each block in turn in its leftmost valid position. An invalid position is one which would cause the block to overlap a space or have solid cells immediately before or after the block (since there should be at least one space between blocks). The ordering of the blocks should of course match that of the clues describing their lengths.

In many cases this simple block-placing algorithm will indeed generate the leftmost solution. In some cases an additional step is required. This additional step is needed when there are existing cells in the line known to be solid but which aren't overlapped by any of the placed blocks; these are termed *unassigned solids*. This can occur if the entire simple leftmost solution is further left than an existing solid, or if known empty cells forced the block placement algorithm to skip over an existing solid. These situations are shown in Figure 22.

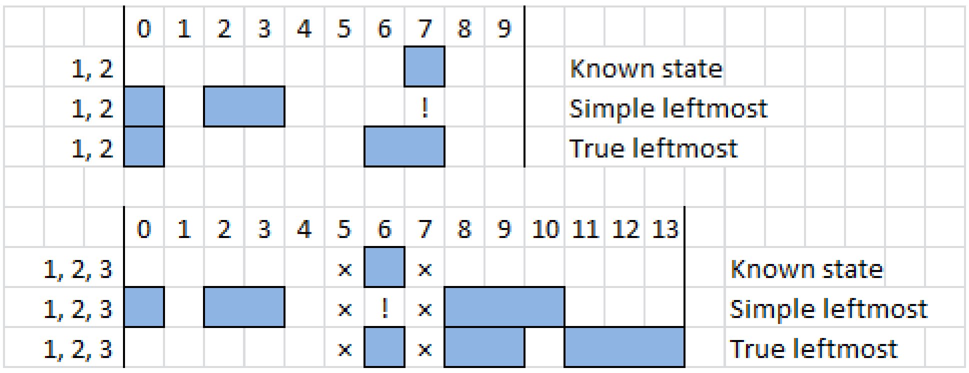


Figure 22: Problems with the simple leftmost block placement algorithm.

Figure 23 details the algorithm for handling these unassigned solids, while Figures 24-27 give various examples of the algorithm in action. Starting with the rightmost unassigned solid, the algorithm grabs each block left of the unassigned position (starting with the rightmost and working left) and tries to find the leftmost valid position for this block which includes the unassigned solid. If there is no valid position then the algorithm grabs the next block over and tries to position that one instead. If the first block to the left of the unassigned solid can be positioned to cover the solid then that block is moved there and the algorithm moves on to the next rightmost unassigned solid. If a block other than the one to the immediate left of the unassigned solid is the one to cover the unassigned solid, then all blocks to the right of that block must be repositioned to preserve block order. If no block can validly cover the unassigned solid then the line is unsolvable. Moving a block to cover an unassigned solid may create more unassigned solids and so this algorithm can end up repositioning blocks many times, but by always working with the rightmost unassigned solid it will eventually find the true leftmost solution (or show that there isn't one).

```
run simple block placement

while there are unassigned solids
    get the rightmost unassigned solid
    get the rightmost block to the left of this solid
    move the block so that its right edge overlaps the unassigned solid
    while the block overlaps the unassigned solid
            if this is a valid position
                    move on to next unassigned solid
            else
                    shift block right by one

    // if we get here then there is no valid way to overlap this block and solid
    do
            get the block to the left of the one we just tried
            move the block so that its right edge overlaps the unassigned solid
            while the block overlaps the unassigned solid
                    if this is a valid position
                            reposition all blocks to the right of this block
                            move on to next unassigned solid
                    else
                            shift block right by one
    until we run out of blocks to try (therefore unsolvable)
```

Figure 23: Pseudocode for handling unassigned solids after simple block placement.

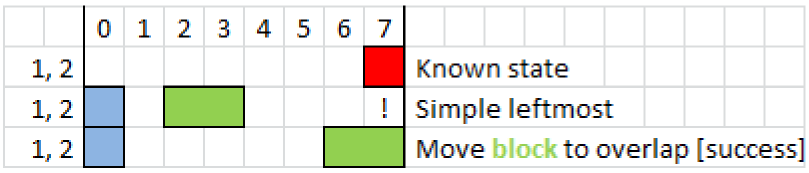


Figure 24: The simplest case when resolving unassigned solids.

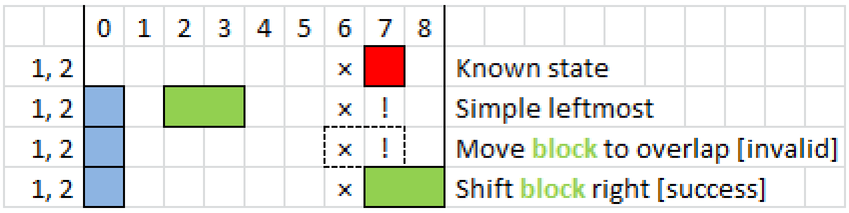


Figure 25: Having to shift the block after the initial reposition.

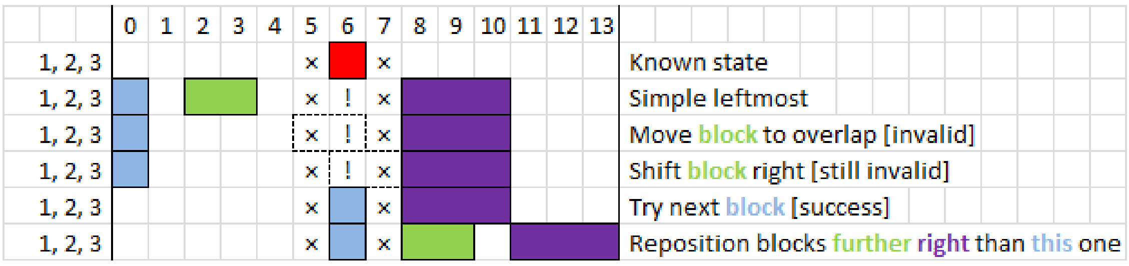


Figure 26: Moving on to the next block once all positions have been tried.

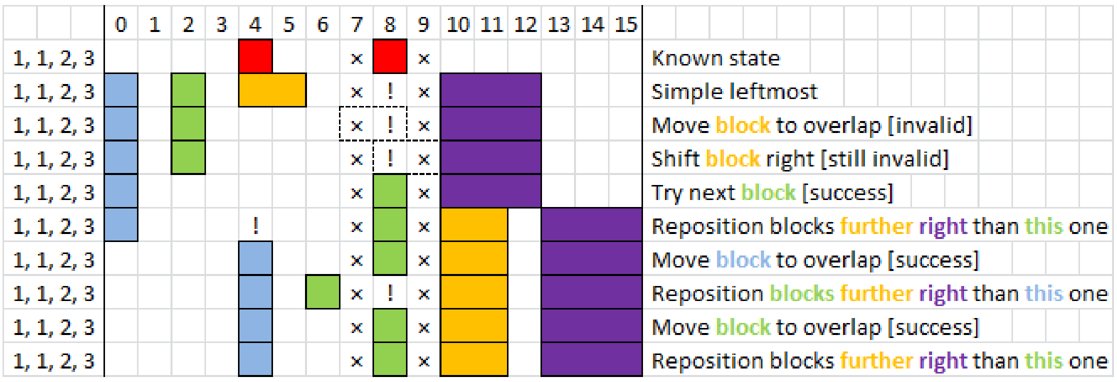

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
1, 1, 2, 3
Known state
Simple leftmost
Move block to overlap [invalid]
Shift block right [still invalid]
Try next block [success]
Reposition blocks further right than this one
Move block to overlap [success]
Reposition blocks further right than this one
Move block to overlap [success]
Reposition blocks further right than this one

Figure 27: Moving blocks may create more unassigned solids and require multiple repositions.

A mirrored version of this entire process must then be done to find the rightmost solution. A different (but much more computationally expensive) way to find the leftmost and rightmost solutions is to find every possible solution, eliminate the invalid ones, then measure the ‘leftness’ of each remaining solution and choose the two which are ‘most left’ and ‘least left’. This process is detailed in Appendices A and B.

3.5 Adapting 2D puzzle techniques to 3D

As mentioned in section 3.1.2 3D puzzles have different types of clues to 2D puzzles, but all of the solver techniques so far have dealt exclusively with the type of clues found in 2D puzzles. As a recap, there are three types of clues in a 3D puzzle:

- **Plain number** - Works the same way as a single number in a 2D puzzle: it means that there is one contiguous block of X solid cells in the line. This is termed a ***single*** clue.
- **Number with a circle** - This means that there are X solid cells in the line split into exactly two blocks of unknown lengths (the lengths sum to X). This is termed a ***double*** clue.
- **Number with a square** - This means that there are X solid cells split into three or more blocks of unknown lengths (the lengths sum to X). This is termed a ***triple+*** clue.

Solving a single clue works exactly the same as it would in a 2D puzzle. In solving a double or triple+ clue it must first be expanded into the set of all possible 2D clues and each of these must then be solved simultaneously. If the same deduction is made in every possible valid expansion of the clue then that deduction can safely be said to be true. As more information about the state of cells in the line is found, more of the possible expansions become invalidated as they have no possible solution which doesn’t conflict with the known state. In this way the set of possibilities slowly reduces down to the actual description of the line. The possible expansions of the first few 3D clues are given in Table 1.

	1	**2**	**3**	**4**	**5**	**6**
Single	1	2	3	4	5	6
Double	-	1, 1	1, 2 2, 1	1, 3 2, 2 3, 1	1, 4 2, 3 3, 2 4, 1	1, 5 2, 4 3, 3 4, 2 5, 1
Triple+	-	-	1, 1, 1	1, 1, 1, 1 2, 1, 1 1, 2, 1 1, 1, 2	1, 1, 1, 1, 1 2, 1, 1, 1 1, 2, 1, 1 1, 1, 2, 1 1, 1, 1, 2 3, 1, 1 1, 3, 1 1, 1, 3 2, 2, 1 2, 1, 2 1, 2, 2	1, 1, 1, 1, 1, 1 2, 1, 1, 1, 1 1, 2, 1, 1, 1 1, 1, 2, 1, 1 1, 1, 1, 2, 1 1, 1, 1, 1, 2 3, 1, 1, 1 1, 3, 1, 1 1, 1, 3, 1 1, 1, 1, 3 2, 2, 1, 1 2, 1, 2, 1 2, 1, 1, 2 1, 2, 2, 1 1, 2, 1, 2 1, 1, 2, 2 4, 1, 1 1, 4, 1 1, 1, 4 3, 2, 1 3, 1, 2 2, 1, 3 2, 3, 1 1, 3, 2 1, 2, 3 2, 2, 2

Table 1: Possible expansions of some 3D clues into sets of 2D clues.

Mathematically, writing a number as the sum of a set of positive integers is known as *composition*, from the field of combinatorics (Heubach and Mansour 2004). When expanding double clues only compositions with two elements are needed, while for triple+ clues only compositions with three or more elements are needed. In general there are 2^{n-1} distinct compositions of n where n is a natural number more than 0 ($n \in \mathbb{N}_1$). It can be seen from the table above that restricting this set of compositions to only those with two elements (i.e. a double expansion) results in $n - 1$ compositions, while for a triple+ expansion there are $2^{n-1} - n$ compositions, a sequence which follows the set of Eulerian numbers (On-Line Encyclopedia of Integer Sequences 1996).

See Appendix C for a discussion of the algorithm for finding these expansions.

3.6 Brute-force solver

The brute-force puzzle solver iterates over every line of the puzzle and applies the linesolver algorithm until either an iteration limit is reached or the puzzle is solved. It is the simplest way to apply the linesolver to complete a puzzle and serves as the benchmark against which the other solvers are compared.

```
for i = 0 to iteration limit
    for each line
            apply linesolver
            if puzzle complete
                    return true
return false
```

Figure 28: The brute-force solver in pseudocode.

3.7 Hierarchical solver

The hierarchical solver is an attempt to model one aspect of how human players tend to solve puzzles: look for information, then see if that new information can be used to find more. This is achieved by iterating over the puzzle in brute-force fashion until a linesolver manages to make some deductions; the lines which are perpendicular to these deductions (i.e. they share a newly-modified cell) are then pushed into a queue and linesolvers are applied in order until the queue is empty. Whenever deductions are made on lines from the queue, unsolved perpendicular lines are again pushed into the queue. In this way, one deduction leads to another which leads to another and information spreads outwards from an initial discovery.

Once a linesolver has been applied to a line, no more information can be deduced from that line until deductions in perpendicular lines result in one or more cells being modified. Since lines perpendicular to deductions are immediately added to the queue, the hierarchical solver need only check each line in the puzzle once. If the puzzle isn't solved after pushing each line in turn into the queue and processing that queue until it's empty, then the puzzle is unsolvable.

```
for each line
    push line into queue
    while queue not empty
        line = queue.pop()
        before = line
        apply linesolver
        for each cell in line
            if line[cell] != before[cell]
                push unsolved perpendicular lines into queue
    if puzzle is solved
        return true
return false
```

Figure 29: Pseudocode for the hierarchical solver.

3.8 Heuristic solver

The heuristic solver is an attempt to model a different strategy used by human players: look for the easiest/most obvious information. Every unsolved line in the puzzle is given a priority based on an estimate of how much information could be deduced by applying the full linesolver there. The linesolver is then applied to the line which is estimated to be most likely to give the most new information, i.e. the line with the highest priority. One of two things will then happen based on whether or not new information was indeed found:

- **No deductions made**. As well as a priority, each line also has a counter for how many times it has been a *bad guess*, i.e. how many times it has been the highest priority line but no deductions were made by applying a linesolver. Whenever a line comes up as a bad guess, its counter is increased by one and its priority is reduced by the bad guess count. So the first time a line comes up as a bad guess its priority is reduced by one, the second time it is reduced by two and so on.
- **Deductions made**. If the line is now solved then remove it from the priority queue, otherwise reduce its priority (since there is now less information to be found and we know we cannot deduce anything more until perpendicular deductions are made) and reinsert it. Increase the priority of all unsolved lines perpendicular to the deductions made, since those lines now have new information and are therefore more likely to result in more deductions.

This process repeats until the queue is empty (therefore the puzzle is solved) or no more deductions can be made on any of the remaining lines (it is unsolvable).

3.8.1 The Heuristic

This solver hinges on the effectiveness of the estimate for how much information is likely to be deduced from a given line. There are many potential factors to consider when calculating such an estimate, including:

- The length of the line
- The number of clues
- How many cells in the line should be solid (i.e. the sum of the clues)
- How many cells in the line are already known to be solid or empty
- Ratio between line length / number of cells already known / number of cells not yet known
- Tracking whether the line has changed since a linesolver was last applied there
- How many linesolvers have been applied elsewhere since one was last applied here (i.e. how long has it been since a linesolver was last applied here?)
- The total number of times a linesolver has been applied here
- The number of times a linesolver has been applied here without deducing anything (i.e. the bad guess count)

It is necessary to use a combination of these factors when estimating priority because there are underlying relationships between the factors which can affect how much potential information can be gained. For example, Figure 30 illustrates the relationship between the number of clues and the sum of the clues.

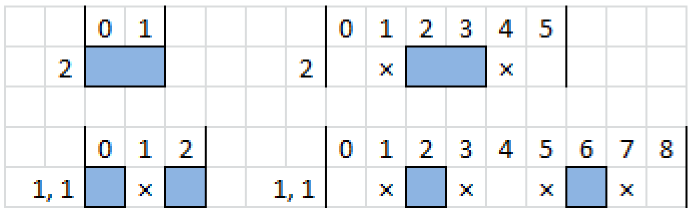


Figure 30: The difference in potential information between a block of length two and two blocks of length one.

Every block is guaranteed to be bounded by either empty cells or the boundaries of the puzzle. There is therefore a minimum and maximum amount of information to be gained by the placement of a block depending on where it is in the line. The top examples in Figure 30 show that the potential information that can be deduced by placing a single block of length two is between two and four cells (ignoring the solution of the rest of the arbitrarily-long line). However, the bottom examples show that the potential information from placing two blocks of length one is between three and six cells. In both cases the sum of the clues are the same. Thus the number of clues should be used in conjunction with the sum of the clues for a more accurate estimate of the potential information. This is just one such relationship between these potential factors.

Theoretically the 'optimum' heuristic would result in the minimum possible number of linesolver applications required to solve the puzzle. Every linesolver would be applied to the line which results in the most new information. However, because there are so many potentially-important factors with potentially-important relationships between them, the discovery or calculation of such a heuristic is a complex problem indeed. This problem is discussed further in section 6.2.2.

After a series of trial-and-error attempts based on simple reasoning, a reasonably-well-performing heuristic was found which takes into account three factors:

- The number of cells known to be solid
- The number of cells known to be empty
- The bad guess count for the line

As explained at the start of this section, the bad guess count is used to reduce the priority of a line when it has the highest priority but no deducible information. When deductions are made, the priority of the line is reduced by $2s + e$ where s is the number of new solid cells found and e is the number of new empty cells found. Relatedly, lines perpendicular to new solids have their priority increased by 2 while lines perpendicular to new empty cells have their priority increased by 1.

The reasoning for weighting solid cells as twice as important as empty cells is simple; given a line with no known state there can be no deduction of empty cells until at least one solid cell has been found. In general the solving of a puzzle is about finding solid cells; finding an empty cell is simply a helpful by-product which then imposes restrictions that allow for the deduction of more solids.

The reason that empty cells are included in the heuristic at all is so that longer lines are prioritised over shorter ones. Again the reasoning is simple; a long line has more potential information to give than a short one. Thus this simple heuristic prioritises in two ways:

- Finding solid cells is prioritised over finding empty cells
- Solving longer lines is prioritised over shorter lines

Although far more complex heuristics could no doubt be found which more effectively estimate where successful deductions can be made, this heuristic performs well enough for the purposes of this application.

3.8.2 Initial priority

It may seem reasonable to set the initial priority of each line as two times the number of required solids plus the number of required empty cells; however the algorithm actually performs significantly better by instead setting the initial priority for all lines to be zero. This is because the above heuristic isn't particularly good at guessing *where* information can be deduced, but it is very good at steering *towards* it. By giving all lines equal priority at the start, the algorithm very quickly begins to self-correct as lines with and without deducible information are encountered. The rate at which bad guesses are made reduces dramatically over time as the algorithm stabilises itself.

3.8.3 Exit clause

The brute force and hierarchical solvers offered natural exit clauses for when to consider a puzzle unsolvable: brute force simply stops after a given number of iterations, while the hierarchical solver need only push each line into the queue once and fully process the queue each time to know whether a puzzle is unsolvable. At first the heuristic solver doesn't seem to offer an obvious way to know when a puzzle cannot be solved, but in fact it is fairly straightforward.

During the operation of the heuristic solver, the number of consecutive bad guesses is tracked. Whenever deductions are made this tracker is reset to zero. When lines are fully solved they are removed from the priority queue. Lines with deducible information will have a higher priority than lines without deducible information and therefore will be encountered, solved and either reinserted or removed depending on whether it is fully solved. Therefore, if the number of consecutive bad guesses ever matches or exceeds the number of remaining lines in the queue the puzzle can be said to be unsolvable, as no more information can be deduced from any of the unsolved lines. The potential problems with this exit clause are discussed in section 4.3.

3.8.4 Pseudocode

```
set priority of all unsolved lines to 0
add all unsolved lines to priority queue
consecutive_bad_guesses = 0
while queue not empty
    line = queue.remove_highest_priority()
    apply linesolver
    if deductions were made
        num_solids_found = 0, num_empties_found = 0
        for each cell in line
            if cell is now solid but wasn't before
                increase priority of unsolved perpendicular lines by 2
                num_solids_found += 1
            else if cell is now empty but wasn't before
                increase priority of unsolved perpendicular lines by 1
                num_empties_found += 1
        if line is still unsolved
            line.priority -= 2 * num_solids_found + num_empties_found
            queue.insert(line)
        queue.sort()
        consecutive_bad_guesses = 0
    else
        consecutive_bad_guesses += 1
        if consecutive_bad_guesses >= queue.size()
            return false
        line.bad_guesses += 1
        line.priority -= line.bad_guesses
        queue.insert(line)
        queue.sort()
return true
```

Figure 31: Pseudocode for the heuristic solver.

3.9 Pre-pass

Both the hierarchical and heuristic solvers benefit greatly from a pre-pass procedure operating on every line in the puzzle exactly once. Rather than running the full linesolver in brute-force fashion, the pre-pass looks only for lines which are completely empty or which have a complete description and solves them accordingly.

3.10 Testing Methods

3.10.1 Puzzle generation

The effectiveness of voxelization as a method of generating 3D nonogram puzzles will be discussed qualitatively based on the results of voxelizing a set of test models at various resolutions. Solving these puzzles will then show how the solvers handle a variety of puzzles and show how grid resolution affects solver performance. The models are described in Table 2.

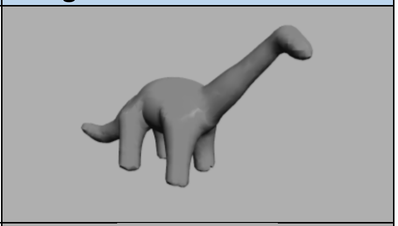
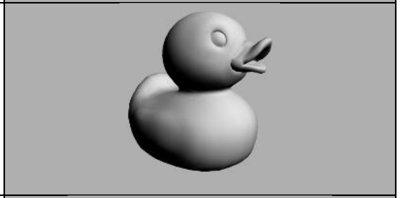
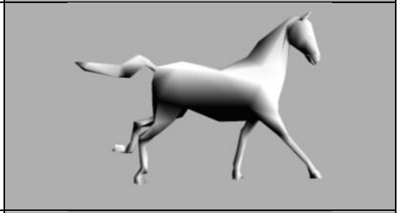
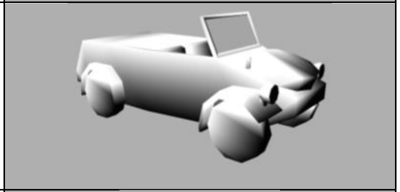
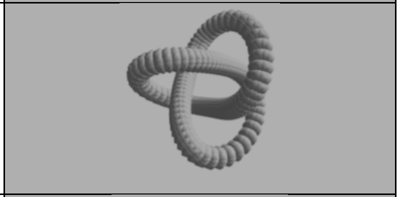
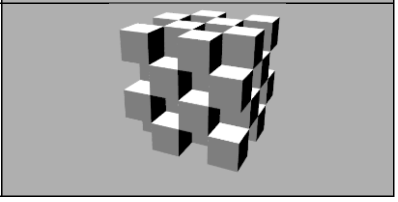
Model	Reasoning	Image
Dino	An ideal test model as it has many different features: larger along one axis; large solid area (torso); large empty areas (above tail and torso, below neck); long thin parts (neck, tail); lines with multiple blocks (legs, tail + underbelly)	
Duck	Mostly solid; roughly the same size in all axes	
Horse	Large solid area (torso) with very thin angled parts (legs)	
Jeep	Mostly rectangular; mostly hollow which results in many lines with large numbers of blocks	
Knot	Mostly empty; many lines with multiple small blocks	
Even-sided checkerboard	Unsolvable as it has two solutions. A 4×4×4 model is shown to the right but a voxel model is created at each resolution as required	

Table 2: Test models.

The set of test models will be voxelized at three different resolutions:

- 12×12×12 resulting in a maximum of 1728 cells in 432 lines
- 24×24×24 resulting in a maximum of 13824 cells in 1728 lines
- 36×36×36 resulting in a maximum of 46656 cells in 3888 lines

The 12^3 resolution was chosen as it is small enough to result in realistically-sized puzzles but large enough to still give recognizable voxelizations of some of the test models. The 24^3 resolution was chosen to show the upper bounds of what can reasonably be expected for the size of a 3D puzzle for an advanced player. The 36^3 resolution was a logical progression of this sequence and shows how the solvers deal with unreasonably large puzzles and how accurate the voxelization process can be at high resolutions.

Note that the cell and line numbers given above are maximums: when a model is voxelized the dimensions of the grid are shrunk to best fit the model, so in most cases only one axis of the puzzle will be the full original length and there will be far fewer cells and lines to solve.

This range of models and resolutions has been carefully chosen to provide a varied set of data so that conclusions can be drawn on the general effectiveness of this method for generating puzzles. The degree to which the voxelizations recognisably resemble the original models will show both the effect that grid resolution has on puzzle generation, as well as the strengths and weaknesses of the technique in terms of the types of models and model features it can and can't handle.

3.10.2 Puzzle solving

It is difficult to assess the effectiveness of the 3D solvers owing to the fact that no other 3D puzzle solvers exist. The applicability of the 3D techniques to 2D puzzles will be investigated by testing the solvers on the set of 2D puzzles used in an extensive comparison of 2D solvers (Wolter 2013), however direct comparison with the other solvers in the survey wouldn't be of much value. The main metrics the 3D solvers have been designed for are the amount of linesolvers needed to solve the puzzle and how many of those are considered bad guesses (i.e. they deduced nothing). The only quantitative comparison that could be made with other solvers is pure execution time, which has not been the focus.

As such, the only quantitative comparison that can be performed is between the 3D solvers themselves. This will be done by testing the brute force, hierarchical (with and without pre-pass) and heuristic (with and without pre-pass) solvers on each of the 18 puzzles generated as discussed above. The number of linesolvers and bad guesses can then be compared and the effectiveness of the different solvers can be assessed.

4. Results

4.1 Effectiveness of puzzle generation

The visual results of voxelizing the set of test models at 12^3, 24^3 and 36^3 can be seen in Table 3. These results were created by fine-tuning the position and size of each model in the voxel field to create the most recognisable voxelization possible at each resolution. In every case at least one axis still has the full resolution.

The Dino model voxelizes very well at low resolutions and best proves the potential of this technique. Even at 12^3 the individual legs, tail and neck are all clearly visible thanks to the chunky, simplistic style of the original model. At 24^3 the head gains a nice cartoonish amount of detail and at 36^3 the rounded surfaces of the body are more obvious.

The Duck also voxelizes well at low resolutions owing to its very simple shape; being mostly solid and having large smooth surfaces with minimal fine detail means that very little is lost by approximating the surface as a set of voxels. At higher resolutions what little detail there is becomes apparent: the open beak and the flick of the tail, as well as much smoother approximations of the curved areas.

As expected the Horse model fairs poorly. The thick torso and head are approximated reasonably well at all resolutions but the thin, angled geometry of the legs still can't be voxelized well even at 36^3. The tail and each leg are all aligned slightly differently and don't line up with each other, meaning detailed voxelizations of some of these parts lead to poor voxelizations of others.

The main blocky body of the Jeep model tends to voxelize fairly well with the exception of the car doors. The doors gently slope inwards towards the front of the vehicle which means that having the rear half of the sides aligned to the grid cause the doors to straddle the border between voxel lines and so get omitted. The finer detail such as the headlights, steering wheel and thin, sloped windshield are all missing at even the highest resolution, though the mostly axis-aligned seats do make reasonable appearances at both 24^3 and 36^3.

The constantly sloping nature of the Knot model makes it hard to line up with the grid. The 12^3 approximation does a reasonable job for the most part but entire sections are lost to the borders between voxels. This is mostly not a problem at 24^3 and is entirely solved by 36^3.

The only thing of note to say about the checkerboards is that they do of course use the full resolution along all axes resulting in the maximum number of lines and cells at each size, where exactly 50% of the cells are solid and 100% of the lines have the same very large triple+ clue which is half the size of the puzzle (e.g. every line in the 24^3 puzzle has the triple+ clue of ‘12’).

Table 4 gives the final dimensions of each puzzle, including a breakdown of how many empty, single, double and triple+ lines there are and what the minimum and maximum clues are for each. As a quick aside, a 2D puzzle with dimensions a by b has ab cells and $a + b$ lines. A 3D puzzle with dimensions a by b by c has abc cells and $ab + ac + bc$ lines.

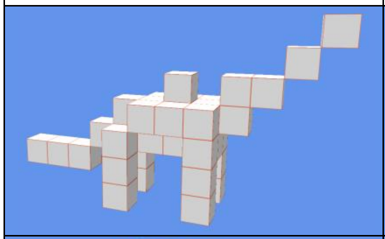
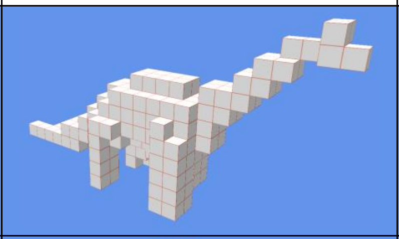
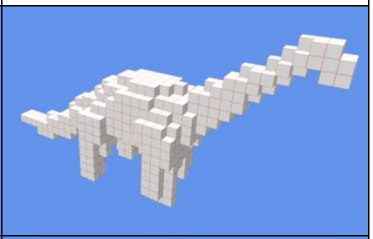
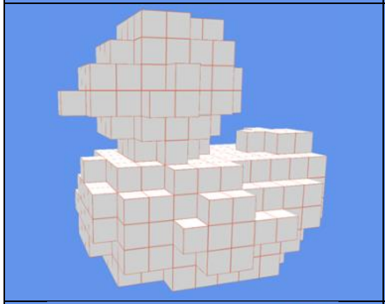
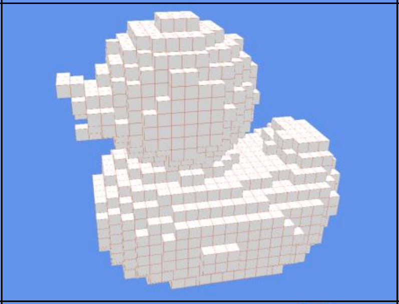
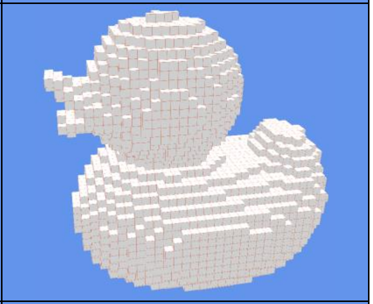
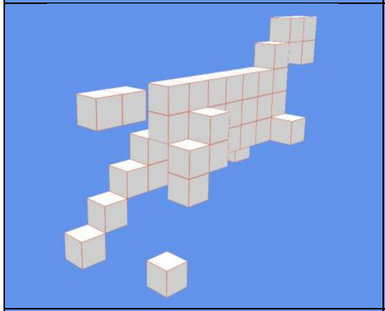
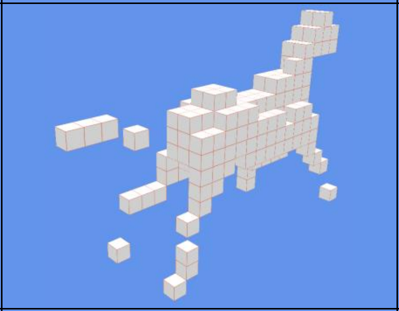
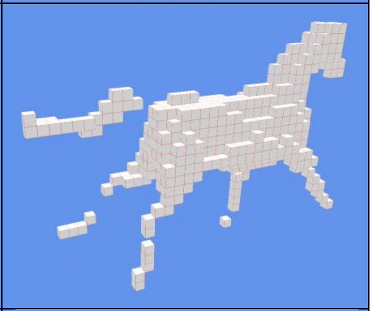
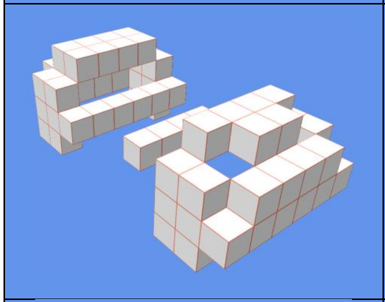
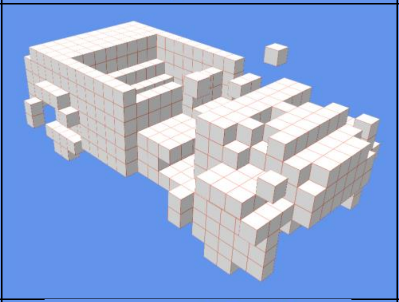
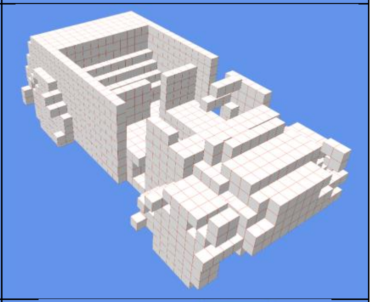
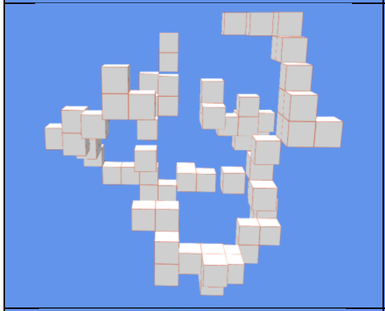
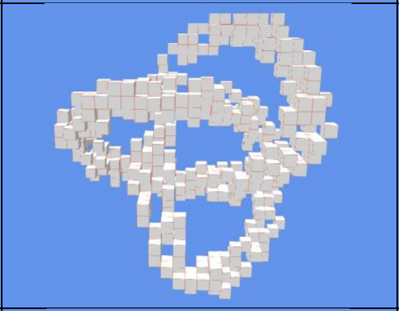
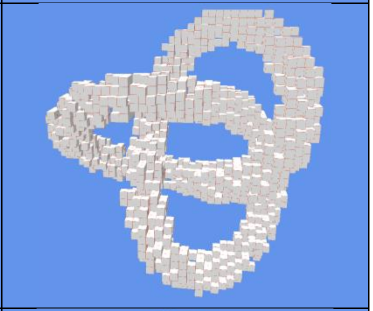
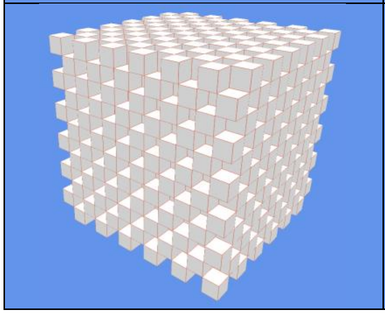
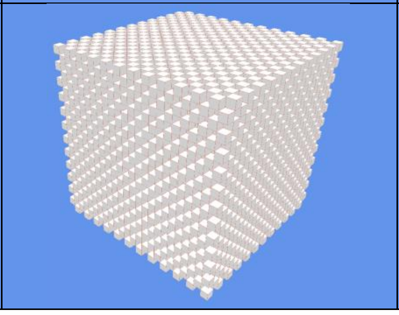
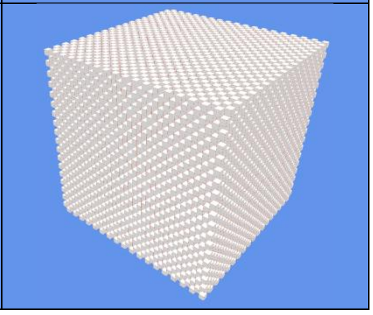
12×12×12	**24×24×24**	**36×36×36**
		
		
		
		
		
		

Table 3: Results of puzzle generation on the set of test models.

Puzzle	Size	Cells	Lines	Empty	Single	Double	Triple+
Dino	12×7×3	252	141	85	46 (1-5)	10 (2-3)	0
	24×12×6	1728	504	312	164 (1-10)	28 (2-7)	0
	36×18×9	5832	1134	694	377 (1-16)	61 (2-13)	2 (6-7)
Duck	12×11×9	1188	339	98	216 (1-11)	25 (4-10)	0
	24×23×17	9384	1351	386	850 (1-23)	113 (3-21)	2 (7-8)
	36×34×25	30600	2974	852	1865 (1-34)	246 (3-31)	11 (6-17)
Horse	12×9×3	324	171	99	65 (1-7)	7 (2-9)	0
	24×16×4	1536	544	348	176 (1-12)	19 (2-13)	1 (9)
	36×25×7	6300	1327	817	469 (1-19)	36 (2-23)	5 (9-24)
Jeep	12×4×6	288	144	61	53 (1-6)	26 (2-6)	4 (4-5)
	24×10×12	2880	648	188	221 (1-12)	184 (2-18)	55 (7-17)
	36×12×17	7344	1248	238	460 (1-19)	335 (2-22)	215 (3-27)
Knot	12×12×9	1296	360	208	134 (1-3)	17 (2-3)	1 (5)
	23×24×18	9936	1398	651	476 (1-5)	212 (2-9)	59 (3-8)
	34×36×26	31824	3044	1162	737 (1-6)	746 (2-9)	399 (3-13)
Check	12×12×12	1728	432	0	0	0	432 (6)
	24×24×24	13824	1728	0	0	0	1728 (12)
	36×36×36	46656	3888	0	0	0	3888 (18)

Table 4: Breakdown of line types and their ranges in the test puzzles.

Figure 32 shows the Horse 36^3 puzzle from a different angle and with the actual puzzle clues rendered. Clues are rendered as billboard sprites which rotate to always face the camera. Black clues are single lines, orange are double (i.e. circle clues) and red are triple+ (i.e. square clues). Note that there are actually clues for every line in the puzzle, but 'zero' clues aren't rendered. As a reminder there are two ways that 3D puzzle designers remove information to make puzzles difficult: clue combination and clue removal. This project includes clue combination as shown by the coloured clues, but does not feature clue removal. Every line has a clue. The clue 'A' has value 10, 'B' 11 and so on.

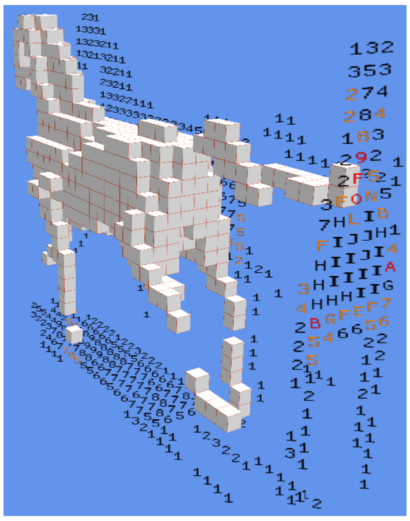


Figure 32: An example of a generated puzzle with the numeric clues rendered.

4.2 Effectiveness of puzzle solvers

Tests were performed on a PC running Windows 7 Professional with an Intel Core i5-2400 3.1GHz quad-core CPU, 8GB of RAM and an NVIDIA GeForce GT 440 graphics card. Full test data can be found in Appendix D. Table 5 summarizes the results, giving the number of linesolvers each solver needed for each puzzle and what percentage of those (to the nearest percentile) were bad guesses (i.e. resulted in no deductions).

Puzzle	Lines	Brute	Hier	Hier+Pre	Heur	Heur+Pre
Dino	141	153 (31%)	134 (13%)	23 (13%)	117 (34%)	24 (17%)
	504	712 (43%)	932 (51%)	128 (10%)	525 (31%)	114 (3%)
	1134	2292 (56%)	2792 (63%)	525 (31%)	1394 (34%)	359 (11%)
Duck	339	544 (47%)	396 (24%)	263 (30%)	455 (20%)	233 (19%)
	1351	3054 (57%)	3286 (54%)	1717 (45%)	2494 (31%)	1351 (25%)
	2974	8835 (62%)	10384 (64%)	7706 (62%)	6602 (36%)	4151 (34%)
Horse	171	228 (48%)	201 (33%)	37 (11%)	180 (44%)	32 (13%)
	544	1128 (64%)	980 (54%)	112 (11%)	638 (50%)	88 (14%)
	1327	2778 (57%)	3057 (59%)	590 (28%)	1821 (50%)	529 (26%)
Jeep	144	166 (53%)	176 (53%)	50 (2%)	199 (37%)	74 (19%)
	648	3957 (82%)	2598 (64%)	1344 (52%)	1916 (54%)	1148 (53%)
	1248	-	-	-	-	-
Knot	360	800 (64%)	871 (64%)	181 (38%)	634 (48%)	138 (22%)
	1398	13371 (87%)	12420 (85%)	4241 (66%)	5951 (70%)	2565 (64%)
	3044	-	-	-	-	-
Check	432	43200 (100%)	432 (100%)	432 (100%)	432 (100%)	432 (100%)
	1728	172800 (100%)	1728 (100%)	1728 (100%)	1728 (100%)	1728 (100%)
	3888	388800* (100%)	3888* (100%)	3888* (100%)	3888* (100%)	3888* (100%)

Table 5: Number of linesolvers and bad guess percentage for each solver type.

The reason there is no data for the Jeep puzzle at 36^3 is that the program failed because it ran out of memory. Referring back to Table 4, the largest triple+ line in the Jeep 36^3 puzzle has the clue 27. From section 3.5 the number of triple+ expansions of 27 is $2^{26} - 27$ = 67,108,837. When the solvers get to that line in the puzzle they attempt to compute all 67 million of these expansions (each consisting of 3-27 clues), solve the line with each of them and store all the results for eventual comparison. This is why the program runs out of memory on this puzzle. Ways to tackle this problem are discussed in section 6.2.2. It is interesting to note that in the Horse 36^3 puzzle there is a triple+ line with the clue 24 which has 8,388,584 expansions, and all of the solvers managed to solve that puzzle. It is simply a case of having enough memory.

The reason there is no data for the Knot 36^3 puzzle is because it proved too difficult. The brute force solver (the fastest in terms of execution time) was left running on the puzzle overnight but after 11 hours it still hadn't solved it (nor had it failed, or found the puzzle to be unsolvable). It is therefore not known whether this or the Jeep 36^3 puzzles are actually solvable purely using linesolvers, though it is expected that they are.

The brute force solver fails and considers a puzzle unsolvable after a certain number of iterations, in this case 100. This is why the brute force solver always uses 100 times as many linesolvers to fail on the checkerboard puzzles than the other solvers. The results given for the 36^3 checkerboard puzzle (marked with *) are predicted results as these tests took too long to actually complete. The checkerboard puzzle is an unsolvable puzzle where not a single cell can be deduced. Other puzzles may have some deducible information but be ultimately unsolvable; such puzzles would have distinct numbers of linesolvers for the hierarchical and heuristic solvers (as seen in section 4.3), but the brute force solver will always require n times the number of lines in the puzzle to fail, where n is the number of iterations to perform (here 100).

Aside from the checkerboard, Jeep 36^3 and Knot 36^3 puzzles, all of the solvers successfully found the solutions for all of the puzzles. The number of linesolvers, number of bad guesses and percentage of bad guesses all generally conform to the same pattern:

- For small puzzles: Heur+Pre < Hier+Pre < Heur < Hier < Brute
- For large puzzles: Heur+Pre < Hier+Pre < Heur < Brute < Hier

As expected the heuristic solver uses fewer linesolvers and makes fewer bad guesses than the hierarchical solver, which uses fewer than the brute force solver. Both the heuristic and hierarchical solvers have significant gains when the pre-pass procedure is used. Interestingly, without the pre-pass procedure the hierarchical solver tends to (but doesn't always) use more linesolvers and make more bad guesses than the brute force solver on large puzzles, but successfully uses fewer on small puzzles (with the pre-pass procedure it always uses significantly fewer). Even without the pre-pass the heuristic solver uses significantly fewer linesolvers than the brute-force solver on almost all of the puzzles. With pre-pass enabled both the hierarchical and heuristic solvers make significantly fewer bad guesses. At best only 2% of the linesolvers employed by the hierarchical solver made no deductions, with at best only 3% for the heuristic solver.

It is interesting to note that all of the solvers are capable of solving a puzzle using less linesolvers than there are lines (the brute force solver is also capable of this given the right puzzle, but doesn't exhibit this behaviour in the test data above). This is because a line can be incidentally solved by deducing the state of all of its cells through perpendicular lines.

Table 6 shows just how effective the hierarchical and heuristic solvers are by giving the number of linesolvers they need as a ratio of the number of line solvers needed by the brute-force solver. Across the set of solvable test puzzles, the hierarchical solver with pre-pass used on average 30.3% of the linesolvers that the brute-force solver did and at best 9.9%. The heuristic solver with pre-pass used on average just 25.6% of the linesolvers and at best just 7.8%.

Puzzle	Size	Brute	Hier	Hier+Pre	Heur	Heur+Pre
Dino	12×7×3	1.000	0.876	0.150	0.765	0.157
	24×12×6	1.000	1.309	0.180	0.737	0.160
	36×18×9	1.000	1.218	0.229	0.608	0.157
Duck	12×11×9	1.000	0.728	0.483	0.836	0.428
	24×23×17	1.000	1.076	0.562	0.817	0.442
	36×34×25	1.000	1.175	0.872	0.747	0.470
Horse	12×9×3	1.000	0.882	0.162	0.789	0.140
	24×16×4	1.000	0.869	0.099	0.566	0.078
	36×25×7	1.000	1.100	0.212	0.656	0.190
Jeep	12×4×6	1.000	1.060	0.301	1.199	0.446
	24×10×12	1.000	0.657	0.340	0.484	0.290
Knot	12×12×9	1.000	1.089	0.226	0.792	0.172
	23×24×18	1.000	0.929	0.317	0.445	0.192
	Average	1.000	0.998	0.303	0.726	0.256

Table 6: Amount of linesolvers required as a ratio compared to the brute-force solver.

The effect of puzzle size on the solvers can be seen in Figure 33, where the x-axis is the number of lines in each puzzle and the y-axis is the number of linesolvers required to solve it. Note that the y-axis is logarithmic here as it best shows the exponential relationship between puzzle size and number of linesolvers. This makes sense given the NP-Complete nature of solving nonograms. Only data for solvable puzzles has been plotted.

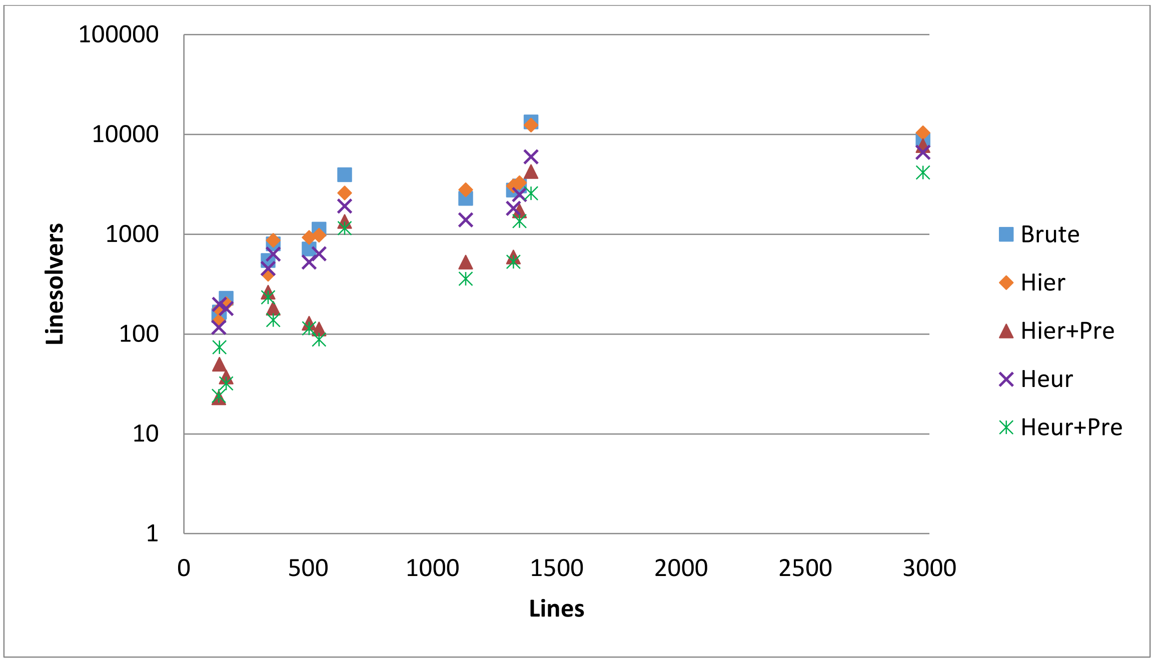


Figure 33: Graph of linesolvers required vs. number of lines.

The effect of puzzle complexity on the number of linesolvers can be seen in Figure 34, where the number of linesolvers has been divided by the number of lines to give the average number of linesolvers required per line. All of the solvers need more linesolvers per line as both the size and complexity increase. Puzzles with more 3D lines (i.e. double or triple+ lines) or 3D lines with larger clue values can be considered more complex and require significantly more linesolvers. This is particularly clear from the Jeep 24^3 and Knot 24^3 puzzles, with the best solver on the prior still having a 52% bad guess rate and the best solver on the latter having a 64% bad guess rate. These puzzles feature a great number of 3D lines often with very large values. It can also be seen that as the size and complexity of the puzzles increases, so too does the performance gap between the brute-force solver and the hierarchical and heuristic solvers with pre-pass enabled.

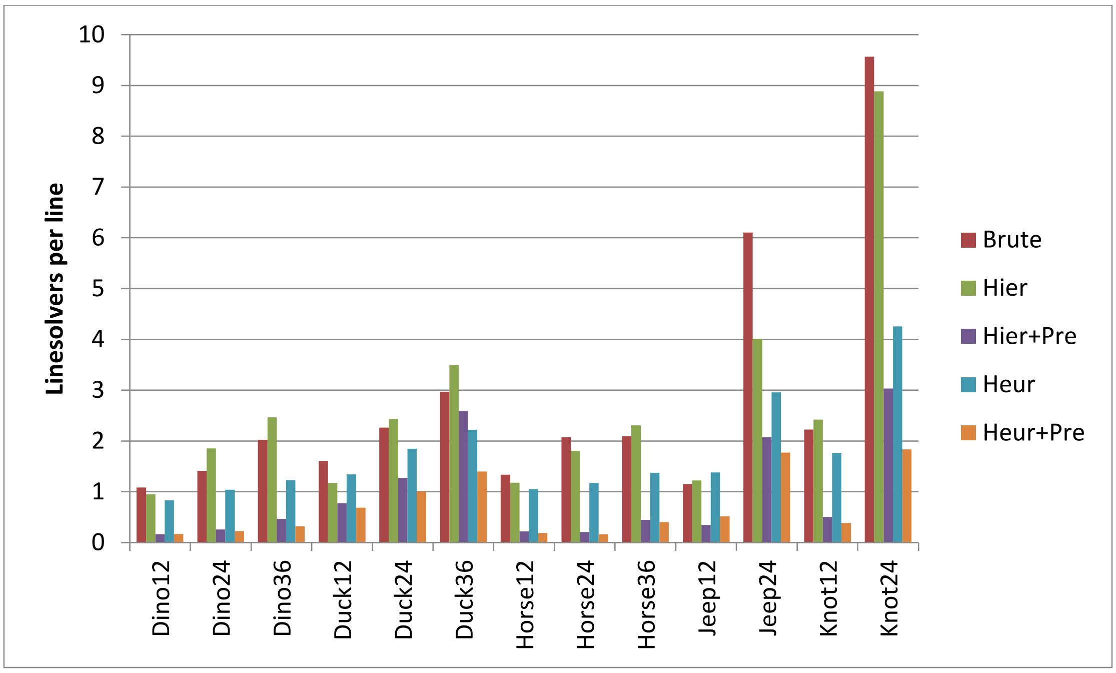


Figure 34: Graph of average number of linesolvers required per line.

Execution time (given in Appendix D) generally follows the pattern of:

- Brute < Hier+Pre < Heur+Pre < Hier < Heur

Theoretically the linesolver algorithm is the expensive part of solving a puzzle, so the expected pattern ought to match that of the linesolvers. In reality, the hierarchical solver maintains a very large queue and the heuristic solver maintains a very large sorted priority queue. The unoptimized overhead of maintaining these structures outweighs the cost of the additional linesolvers needed by the brute-force solver, resulting in these algorithms being slower in terms of execution time. The main metric that these solvers were developed for was reducing the number of linesolvers used and the percentage of those which are bad guesses, and in this regard both the hierarchical and heuristic solvers were a great success. With optimized queue handling it can be expected that these solvers will indeed out-perform the brute-force approach in terms of execution time as well.

4.3 Applicability of 3D solvers to 2D puzzles

A subset of the 31 puzzles used in the WebPBN survey of 2D solvers (Wolter 2013) was used to test how applicable the different solvers are to 2D puzzles. This subset consisted of 8 puzzles as described in Table 7, 3 of which are known to not be solvable purely using linesolvers.

Puzzle	Size	Cells	Lines	Empty	Single	Line-solvable?
Bucks	27×23	621	50	1	49 (1-23)	No
Cat	20×20	400	40	0	40 (1-17)	Yes
Dancer	5×10	50	15	0	15 (1-7)	Yes
Edge	10×11	110	21	0	21 (1-4)	No
Knot	34×34	1156	68	0	68 (1-26)	Yes
Skid	14×25	350	39	1	38 (1-13)	Yes
Smoke	20×20	400	40	0	40 (1-13)	No
Swing	45×45	2025	90	0	90 (1-30)	Yes

Table 7: Summary of the 2D puzzles.

The results of the solvers are summarized in Table 8. The full test data can be found in Appendix E. Of the three unsolvable puzzles, two were at least partly solvable and one was completely unsolvable (i.e. not a single cell could be deduced). All five of the puzzles that can be solved purely using linesolvers were solved successfully by all solvers.

Puzzle	Lines	Brute	Hier	Hier+Pre	Heur	Heur+Pre	Solved?
Bucks	50	5000 (84%)	541 (82%)	513 (84%)	309 (70%)	313 (72%)	Partly
Cat	40	363 (69%)	595 (77%)	595 (77%)	313 (62%)	313 (62%)	Yes
Dancer	15	64 (64%)	35 (43%)	35 (43%)	47* (49%)	47* (49%)	Yes
Edge	21	2100 (100%)	21 (100%)	21 (100%)	21 (100%)	21 (100%)	No
Knot	68	435 (76%)	1012 (90%)	1012 (90%)	696** (81%)	696** (81%)	Yes
Skid	39	291 (72%)	373 (76%)	335 (76%)	249 (59%)	246 (61%)	Yes
Smoke	40	4000 (99%)	169 (80%)	169 (80%)	130 (84%)	130 (84%)	Partly
Swing	90	1392 (67%)	4299 (89%)	4299 (89%)	1165 (55%)	1165 (55%)	Yes

Table 8: Number of linesolvers and the percentage of bad guesses for the 2D puzzles.

The most important results here are for the heuristic solver on the Dancer and Knot puzzles, marked with * and ** respectively. The exit clause for the heuristic solver is to consider a puzzle unsolvable if the number of consecutive bad guesses ever reaches or exceeds the number of unsolved lines left in the queue. The assumption here is that a bad guess will always cause a line to be reinserted in the queue after any line that has actually deducible information. In these two puzzles this assumption failed. The heuristic solver was only able to solve the Dancer puzzle when the exit clause was modified to fail when the consecutive bad guesses reached **two times** the number of lines in the queue, and for the Knot puzzle it had to be increased to **four times**. It is clear from this that the heuristic solver's current exit clause is insufficient, though it has yet to cause a generated 3D puzzle to fail early. Whether this is because of the 3D nature of the generated puzzles or the expert design of the 2D puzzles is unclear.

The pre-pass procedure generally had very little effect on these puzzles as there were almost no empty or completely described lines. Empty lines are generally rare in 2D puzzles and many 2D solvers do not support them, but they are a key part of 3D puzzles and feature in every single puzzle in *Picross 3D* (2009). *Picross 3D 2* (2015) even introduced a mechanic for automatically clearing all empty lines at the start of a puzzle, because of how common they are. Without an effective pre-pass procedure the hierarchical solver often uses far more linesolvers than the brute force solver, though the heuristic solver still generally uses less than either the brute force or hierarchical.

The Edge puzzle was completely unsolvable and the results here follow the same pattern as for the 3D checkerboard puzzles which are also completely unsolvable. The Bucks and Smoke puzzles were at least partly solvable and here it can be seen that the heuristic solver uses fewer linesolvers than the hierarchical one to find what can be found and then fail, considering the puzzle unsolvable.

5. Discussion

5.1 Generation of 3D Puzzles

The results shown in Table 3 are an effective proof-of-concept for this technique. Even at low resolutions it is possible to create recognisable voxel approximations of models and use these as the basis for 3D nonogram puzzles. The Dino model in particular gives encouraging results; the 12^3 voxelization results in a 12×7×3 puzzle with just 141 lines, comparable in size to puzzles in the 'normal' difficulty level of *Picross 3D* (2009). Advanced 2D puzzle players regularly solve puzzles upwards of 40×40 resulting in 1600 cells, again comparable to the number of cells in even the 24^3 Dino puzzle.

As seen in the Dino and Duck models a chunky, cartoonish style seems to have the best low resolution voxelizations. Large features and minimal fine detail means little is lost in the approximation process. It is therefore likely that non-photorealistic models such as those from children's games would work well as the basis for puzzles thanks to their over-emphasized features. Low-poly models would also work well due to the lack of fine detail; assets from games with a low-poly aesthetic, older games, or even low-poly versions of models created for level-of-detail systems could all serve well. The Horse, Jeep and Knot models show how thin, small or angled geometry invariably gets lost in the voxelization process; anything smaller than the size of the voxels themselves is likely to end up between lines and therefore not be represented in the voxel model.

Indeed, as it is a similar discretization of a continuous surface onto a uniform grid, voxelization suffers the same problems that rasterization does and can benefit from the same solutions. In their paper detailing the raycasting-based voxelization approach used in this application Thon, Gesquière and Raffin (2004) detail how to extend the system to support a 3D version of anti-aliasing. Instead of casting only one ray for each line of voxels, the lines are oversampled by casting multiple rays with either a uniform or random distribution. Rather than the simple binary inside/outside determination currently used, this would allow for measurements of the *amount* of each voxel that lies in or out of the model and a tolerance value for what is considered 'in' can be adapted at run-time to get the best results. Other techniques such as using a secondary grid at a higher resolution (known as local refinement) could also help reduce the number and seriousness of the approximation errors (Szucki and Suchy 2012).

It is plain to see that larger and more frequent 3D lines result in increasingly-difficult puzzles. This also follows from the difficulty progression of puzzles in *Picross 3D*, where later levels are larger and have more 3D clues. With this in mind, models with interiors such as the Jeep or with many overlapping parts such as the Knot will generally produce extremely difficult puzzles, as shown by the inability of the solvers to find the solutions for the 36^3 puzzles generated from these models.

5.2 3D Puzzle Solvers

The work done here in creating various 3D puzzle solvers is an important first step in extending existing solver techniques to support 3D. Specifically, the concept of a linesolver algorithm operating on one line of the puzzle at a time and using logic to deduce the state of cells was shown to be very applicable to 3D. By expanding 3D lines into simultaneous sets of 2D lines it should be possible to take many other, more advanced 2D techniques such as boundary logic, smile logic and summing (Wolter [no date]) and apply them to 3D puzzles. Chronological backtracking (Yu, Lee and Chen 2011) could also be used to further develop the solvers and support puzzles that aren't line-solvable. As a tool for puzzle designers the solvers would be more useful if they could validate that there is a single unique solution for a puzzle, rather than simply finding one of possibly many.

The ideas presented here on applying linesolvers to a puzzle in different ways have also proven effective. Both the hierarchical and heuristic solvers use dramatically fewer linesolvers than the brute-force solver for 3D puzzles when the pre-pass procedure is used. Even without the pre-pass the heuristic solver still consistently out-performs the brute-force solver and on smaller puzzles the hierarchical solver does as well. The prevalence of empty lines in 3D puzzles proves to be a strength for the solvers. With empty lines being rare in 2D puzzles, the hierarchical solver suffers when applied to 2D and tends to use more linesolvers and have a higher bad guess rate than the brute-force solver. Even in 2D the heuristic solver manages to generally use less than the brute-force solver, though the margins are smaller. With a better heuristic taking into account different parameters than the one developed here, this approach in particular could lead to a solver that approaches the ideal minimum possible number of linesolvers required to solve any puzzle.

Though the actual execution times for the hierarchical and heuristic solvers are greater than those of the brute-force solver, with optimized queue handling the overhead of queue management would be alleviated and the hierarchical and heuristic solvers could well out-perform the brute-force solver in execution time as well.

In terms of linesolvers and bad guesses the brute-force solver proved to be a good benchmark. In particular the brute-force solver seems to struggle when there are many 3D lines in the puzzle. These are by far the most computationally expensive part of the solvers and the brute-force approach attempts to solve them with as much regularity as any other line, with very little progress made each iteration for these lines. This results in the huge spikes in the average number of linesolvers per line for the Jeep 24^3 and Knot 24^3 puzzles seen in Figure 34. Animating the solver process shows the ineffectiveness of the brute-force solver once the majority of the puzzle is solved. When there are only a few unknown cells left the brute-force solver will often take one iteration per cell, because although the remaining cells may be perpendicular those lines were already attempted this iteration, so the new information isn't available until the next iteration. The rate at which bad guesses are made increases hugely once the majority of the puzzle is solved.

This contrasts very nicely with the performance of the hierarchical and heuristic solvers with the pre-pass procedure enabled. Because of the nature of the pre-pass these solvers tend to have the most effective and consistent results on puzzles with large numbers of empty lines such as the Dino and Horse puzzles, around 60% of whose lines are empty. Animating the solvers shows just how much more effective they are at focusing on where the most information can be gained. Conversely to the brute-force solver, the rate at which these solvers make bad guesses reduces as more of the puzzle is solved. The animations also show just how different all of the approaches are, with lines being solved in drastically different orders.

All of the solvers take exponentially longer as the number and size of 3D lines increases. If the heuristic solver were to be optimized for execution time rather than linesolvers or bad guesses, the priority of 3D lines could be lowered so that they are encountered less often early on and only have to be dealt with once a lot of perpendicular information has already been deduced.

The brute-force solver is by far the worst at determining that a puzzle is unsolvable, using *n* times the number of lines in the puzzle to come to that conclusion, where *n* is the number of iterations to perform. Even then a puzzle could still be solvable and simply require more than *n* iterations. Although it generally uses less linesolvers than the hierarchical solver to determine that a puzzle is unsolvable, the heuristic solver can exit early and consider solvable puzzles a false positive. Clearly work is needed in improving the heuristic exit clause. By far the most effective exit clause is that of the hierarchical solver. Because of the nature of how it processes lines and adds them to the queue, it is guaranteed that a puzzle is not line-solvable once every line has been added in turn to the queue and the queue has been fully processed. This solver is therefore the only one that can safely guarantee that a puzzle is or isn't line-solvable.

5.3 Answering the Research Question

How suitable is voxelizing existing polygonal meshes as a method of generating solvable 3D nonogram puzzles and how can 2D solving techniques be applied to 3D puzzles?

The findings of this research have demonstrated that voxelization can indeed be used to create solvable 3D nonograms and that multiple puzzles of varying difficulty can be generated from a single model by changing the resolution of the voxel grid. The results will likely need further tweaking by puzzle designers, but this method can be used to quickly create numerous puzzles very quickly and easily and can even make use of existing assets from other games. This project serves as an effective proof-of-concept for the technique and with further development it could become a valuable tool in the 3D puzzle creation pipeline. The creation of automatic 3D solvers also serves to benefit designers as it allows them to immediately know whether or not a puzzle is solvable. Extensions to the solvers could ensure that the solution is unique and even give a measure of the expected difficulty level. By treating 3D puzzle lines as simultaneous sets of 2D lines it has been shown that 2D puzzle solving techniques are directly applicable to the 3D puzzle space. In particular the 2D linesolver technique has been adapted to support 3D puzzles and it is therefore clear that many other 2D solver techniques could be adapted to support 3D as well.

6. Conclusions and Future Work

6.1 Conclusion

This project has covered a lot of new ground and involved the creation of many algorithms for tackling the sub-problems within the larger goals of generating and solving 3D puzzles. These include:

- How voxelization can be used to create 3D puzzles from existing polygonal models.
- How to construct the leftmost and rightmost solutions for a 2D line by first finding the naïve solutions and then iteratively shifting blocks until all existing state is accounted for.
- Finding all possible solutions for a 2D line by recursively finding the set of *spreads* for a line then moving these along to find the set of *shifts*.
- How to measure the 'leftness' of a given solution to extract the leftmost and rightmost solutions from the set of all solutions.
- A simplified approach to the linesolver algorithm which uses just two deduction techniques: one to find solid cells and one to find empty cells. The situations where these techniques could lead to false positives are described and handled appropriately. It was also shown how empty lines and lines with a *complete description* can be detected and solved immediately as early-outs.
- How 2D solving techniques can be applied to 3D puzzles by treating 3D clues as simultaneous sets of 2D clues.
- How to expand a 3D clue into a set of 2D clues, known as finding the *compositions of n*, where *n* is the block size. Specific algorithms for finding the double expansion and triple+ expansion of *n* were described.
- A discussion of different approaches to applying linesolvers to the puzzle space with the goal of using fewer linesolvers and having a lower bad guess rate. A performance comparison of the brute-force, hierarchical and heuristic solvers was conducted and the results discussed.

The ultimate aims for this project were to show how voxelization and 3D puzzle solvers could benefit the 3D puzzle creation pipeline. Voxelization has been shown to be a useful tool in the beginning stages of puzzle design as it allows for rapid prototyping of a subject model at various resolutions. Generated puzzles can then be refined to better represent the original subject or to adjust the difficulty of the puzzle by making modifications in tandem with the 3D solvers. Being able to know whether a change has rendered a puzzle unsolvable without having to spend the time solving it by hand could greatly improve the 3D puzzle designer's workflow.

6.2 Future Work

Although a lot of new ground has been covered in this project, it only goes to show how much more could be done. There are many avenues for improvement and many directions to take this further.

6.2.1 Puzzle Generation

The obvious improvement for the puzzle generation process would be to add anti-aliasing support to the voxelization algorithm so that higher quality voxel approximations can be made, which would also then support much finer detail geometry. Oversampling each line of voxels with multiple raycasts would give a ratio of how much of that voxel is within the model, not just a binary in/out (Thon, Gesquière and Raffin 2004). Voxels above a certain threshold value can then be considered solid and those below empty. Varying this threshold value can be used to create a number of puzzles from the same voxelization in a very similar way to how Ortiz-García et al. (2007a) vary the greyscale threshold in their 2D puzzle generation algorithm. An example of this can be seen in Figure 35. The puzzle with the best balance between recognisability and difficulty can then be chosen. Thresholding and variation could also be used to generate nonograms of varying difficulty from the same input, continuing the work by Batenberg et al. (2009).

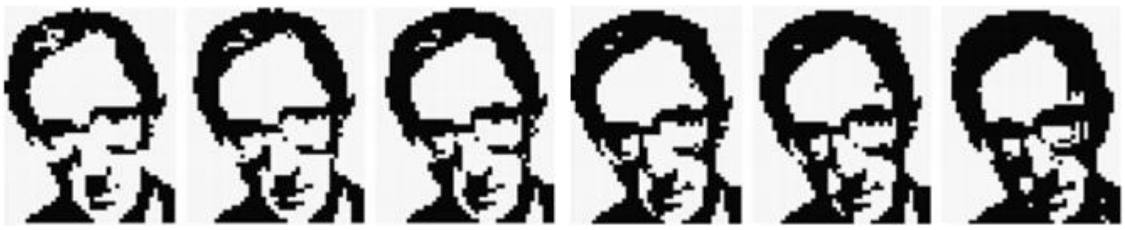


Figure 35: Varying the threshold value in 2D puzzle generation (Ortiz-García et al. 2007a).

The major lacking feature of the 3D puzzles generated in this project is the removal of clues. A 2D puzzle can be considered to consist of two axes of information and the player makes deductions by cross-referencing these axes. In a 3D puzzle there are of course three axes of information, which is enough to make solving the puzzle through cross-referencing trivial. This is why there are many lines in 3D puzzles which don't have clues at all: information has been removed so that there are fewer axes of information for any given cell, making the deduction process more difficult and interesting for the player. Working out how to remove clues while leaving the puzzle solvable and moderating its difficulty (and even the techniques required by the player to solve it) could be an entire project in itself. As such this would likely be the main initial focus for further development as there are many unanswered questions here. For example, is a 3D puzzle guaranteed to be line-solvable if no clues are removed? Is a 3D puzzle still guaranteed to be line-solvable if every cell still has at least two axes of information? A lot of work could be done here.

Another very interesting area for further development is that of colour puzzles. Only 3D black-and-white puzzles were tackled in this project. The colour reduction followed by iterative modification used by Ortiz-García et al. (2007a) could be well-suited to the generation of 3D colour puzzles as well as 2D. The colour of voxels can be set during the voxelization process using a process such as that used by Silva, Pamplona and Comba (2009), then reduced down to however many colours are required for the puzzle. Adding multi-colour support to the solvers would require significant work, though colour puzzles are generally easier to solve (Wolter 2013a).

6.2.2 3D Puzzle Solvers

Out of the brute-force, hierarchical and heuristic solvers the heuristic approach shows by far the most potential. With a good enough heuristic taking into account the right set of parameters, this approach could eventually lead to an algorithm which uses the minimum possible number of linesolvers to solve a puzzle. One potential way to find this heuristic could be to use an evolutionary algorithm or neural network to find permutations of parameters and optimize toward the heuristic that requires the least number of linesolvers and/or has the lowest bad guess rate.

Automatic solvers aren't just useful for algorithmic puzzle generation, they are also extremely useful for human puzzle designers. Automatic solvers allow them to iteratively change their puzzle without having to manually solve it each time and can even give a measure of the expected difficulty. For this reason it would be very useful to integrate the 3D solvers with voxel modelling software such as *Magica Voxel* (ephtracy 2016). Extending the solvers to output some measure of difficulty, as well as validate that there is only one solution rather than simply find one, would also make for a more useful puzzle design tool.

In terms of performance there are several techniques which could give substantial improvements. Linesolvers can safely be applied concurrently on lines which aren't perpendicular (i.e. they share no common cells). Multi-threading the application of linesolvers could therefore drastically improve the performance of all of the solvers, particularly the brute-force solver as there is a consistent order in which linesolvers are applied and there is no queue dictating which lines to solve and therefore minimal resource sharing between threads. It could also massively speed up solving large double or triple+ lines since every expansion must be solved simultaneously and the results then compared; with multi-threading it would literally be simultaneous.

Caching could also be used to drastically improve solver performance. During the solution process a line with a length and state that has already been encountered can come up very often; caching is a way to capitalise on this and avoid duplicate calculations. First introduced in the BGU solver (Pomeranz, Raziel and Rabani 2010), whenever a line is given to a linesolver the result is stored in a hash table keyed by the clues and initial line state. The table is checked before running any linesolver and the pre-calculated solution can then be used if present, avoiding duplicate work. There is a surprisingly high hit-rate on the hash table with upwards of 90% of the solutions being found there, often resulting in more than a 50% improvement in execution time (Wolter 2013c). This could be particularly beneficial in avoiding very costly double or triple+ line calculations that have been encountered before.

By far the biggest performance drain in the current implementation is the double and especially the triple+ lines. When these lines are encountered every possible expansion of the 3D clue into sets of 2D clues is calculated and all of these lines are solved simultaneously and their results stored, which are then compared. As more of the line is solved more of the expansions become invalid as they no longer have any possible solution that doesn’t contradict the existing state. Eventually the set of expansions reduces down to the actual description of the line. The problem is that this uses a huge amount of memory. For example the current implementation crashes after running out of memory when trying to solve the Jeep 36^3 puzzle because there is a triple+ line with the clue 27. The program has to calculate the roughly 67 million expansions of 27 (each consisting of 3-27 clues), then solve the line with each, storing all 67 million resulting lines and comparing them all. There are a few different approaches that could be taken to handle this huge memory overhead:

1. **Calculate and forget.** Calculate all expansions for a line when it comes up, solve all of them (many won’t have solutions), compare the results then throw everything away. This is the current method and is the simplest to implement. It’s very computationally expensive and will run out of memory for very large clues. Once an expansion has been found to not have a solution it can safely be ignored, but this approach will recalculate and test every expansion every time.

2. **Calculate and remember.** Calculate every expansion for every line once and keep them all in memory. Lines with the same clue will have the same initial set of expansions but must have their own working copy. When an expansion is found to not have a solution for a particular line, remove it. This would give optimal performance as it avoids redundant tests, but would use an incredible amount of memory.
3. **Calculate and store.** Calculate every expansion for every line once and store them all to disk. The set of expansions for a line are loaded in when needed and those without solutions are removed from the file. This could even be taken one step further, with only one expansion being loaded at a time rather than the whole set. This is obviously very slow, but it would allow the solver to handle a clue of *any* size: the only limitation is disk space.
4. **Pre-calculate.** Calculate the expansions for all the numbers up to a reasonable limit and compile them into the program or read them from disk. All lines with the same clue use the same full set of expansions. There would be duplicate work in trying to find a solution for an expansion that has previously been shown not to have one for the current line, but it completely avoids the cost of calculating the expansions themselves at run-time and uses less memory than remembering or storing only the valid expansions for each line.

One of the great benefits of the 'calculate and store' approach is that it would allow the solver to support puzzles of any size, since only one line is dealt with at a time. The raycasting-based voxelization method used in this project has a similar benefit and indeed the authors point out that their algorithm can handle voxelization at any resolution by storing the results immediately to disk (Thon, Gesquière and Raffin 2004). There is a great symmetry here between the puzzle generation and puzzle solving methods in operating on one line at a time, allowing for 3D puzzles of any size to be created and solved.

Appendix A: Finding every possible solution of a line

Finding every possible solution of a line is achieved by first finding every unique combination of blocks and the spaces between them (these solutions are termed the *spreads*), then shifting these solutions rightwards as much as possible one cell at a time (these are the *shifts*). Figure A1 gives an example of the spreads and shifts when trying to place three blocks of two into a line of ten cells.

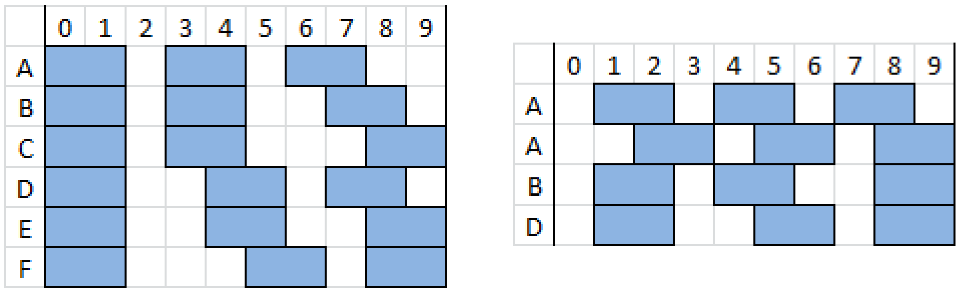


Figure A1: The spreads (left) and shifts (right), with cell indices along the top. The spreads are labelled A-F while the shifts are labelled with the spread they are derived from.

Every possible solution of the line is contained in the set of spreads or the set of shifts. These can be treated as one collection of all solutions, as there is no distinction between the spreads and shifts other than how they are found. Note that if existing state has already been deduced for this line (i.e. certain cells are already known to definitely be solid or empty), many of these solutions will be invalid as they contradict this state. This is the naïve set of *every* possible way of ordering a set of blocks in a line of a certain size. Invalid solutions can easily be removed from the set afterwards, or tested for as each solution is found (though care must be taken here as an invalid spread may still result in valid shifts).

Spreads are found through a recursive function. First the naïve leftmost possible solution is calculated and stored in the set of solutions. The recursive function is then called on the second block in this solution (note that every spread has the first block at the far left of the line; solutions where this is not the case are always in the set of shifts). The function attempts to shift each block in turn rightwards one cell at a time, starting with the rightmost block. When a block cannot move rightwards anymore, the recursion backtracks and moves the preceding block rightwards by one then proceeds forwards again. See Figure A2 for pseudocode of the complete algorithm.

This algorithm can handle any size of line or block and any number of blocks (one or more, zero being trivial). The set of spreads and shifts in Figure A1 are given in the order that this algorithm finds them and so serve as a good example when walking through this pseudocode. A larger example can help to show the underlying pattern of this algorithm; see Figure A3 for an example of placing four blocks of two into a line of fourteen cells.

```
func FindAllSolutions()
    store leftmost line                            #spread A
    if line has more than one block
            FindSpreads(second block)              #spreads B onwards

    for each spread                                #shifts
            while can shift right
                    shift right by one
                    store line
end func

func FindSpreads()                         #recursive spread-finding function
    if is last block
            while can shift right
                    shift right by one
                    store line
    else
            while can shift right
                    FindSpreads(next block)
                    shift right by one
                    store line
end func
```

Figure A2: Pseudocode for finding every possible solution of a line.

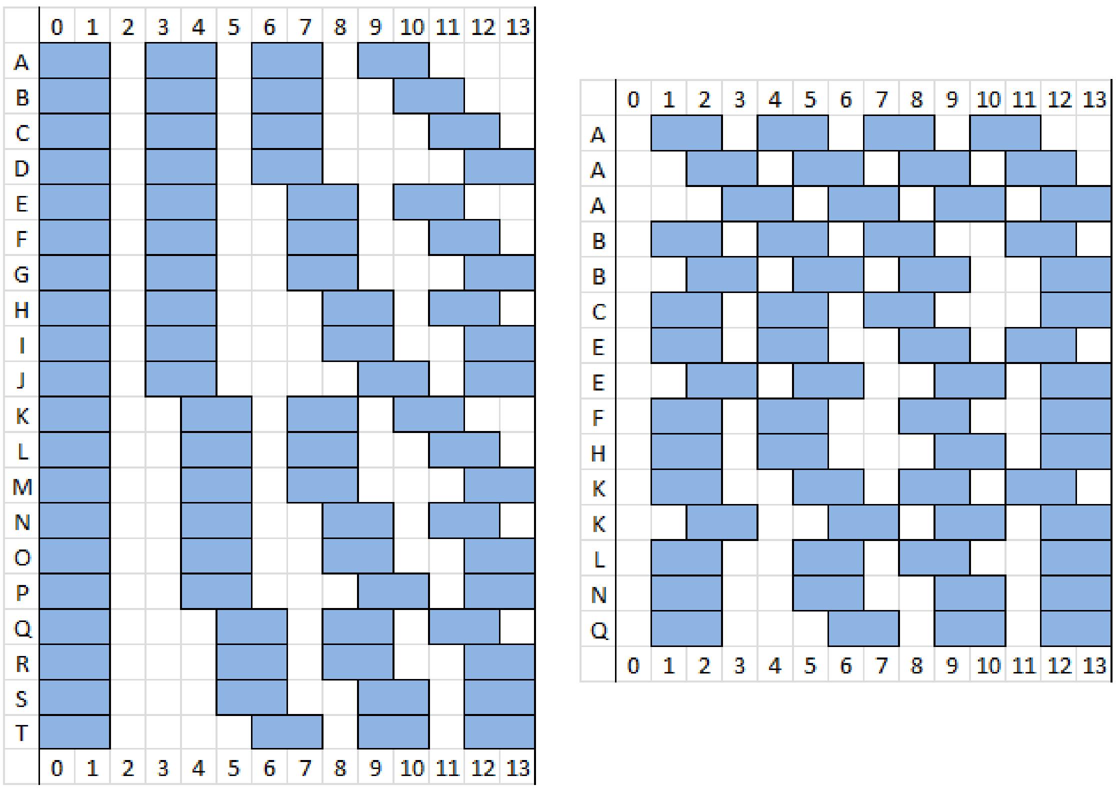


Figure A3: Finding the spreads (left) and shifts (right) for a large line.

Appendix B: Measuring the 'leftness' of a solution

The 'leftness' of a solution can be measured by summing the indices of the leftmost cell in each block. This is shown to the right of the solutions in Figure A4, where blocks of length four and three are being placed in a line of ten cells.

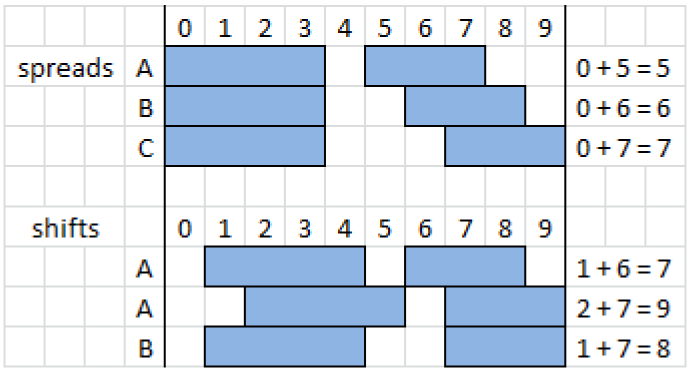


Figure A4: Summing the leftmost indices to measure 'leftness'.

The leftmost solution is the solution with the lowest index sum, while the rightmost solution is the one with the highest index sum. Multiple solutions may have the same index sum; all but one of these will be invalidated by existing state in the line (if there is no existing state then the naïve leftmost and rightmost solutions would have been chosen and it is guaranteed that these solutions will have unique index sums).

An example of finding the leftmost and rightmost solutions while taking into account existing state can be seen in Figure A5, where cell nine from the above example is known to be solid and so invalid solutions have been discarded. The top row has the lowest index sum and is therefore the leftmost solution, while the middle row has the highest index sum and is therefore the rightmost solution.

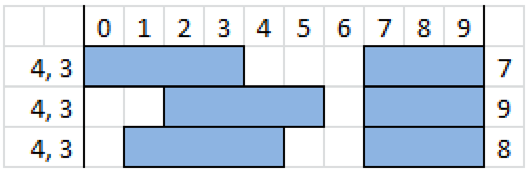


Figure A5: Valid solutions when cell nine is known to be solid.

Appendix C: Finding the compositions of n

The algorithm for generating these sequences is as follows:

```
func FindCompositions(all_comps, clue_sum, triple_or_more)
    if triple_or_more                                          ## triple+ expansion
            comp = {}
            for i = 0 to clue_sum
                    comp.add_element(1)             # start with all 1s
            all_comps.add_element(comp)
            FindSets(all_comps, comp, 0)
    else                                                       ## double expansion
            left = clue_sum - 1                     # left counts down
            right = 1                               # right counts up
            while left > 0
                    all_comps.add_element( { left, right } )
                    left -= 1
                    right += 1
end func

func FindSets(all_sets, set, start_index)
    if set.last_element() != 1 or set.size() <= 3
            return
    set.remove_last_element()
    for i = start_index to set.size()
            set[i] += 1
            all_sets.add_element(set)
            FindSets(all_sets, set, i)              # recurse
            set[i] -= 1
end func
```

Figure A6: Pseudocode for performing double or triple+ expansion.

This algorithm starts with the set of n 1s. It then removes the last 1 from the set and adds it to each element in turn, storing these sets also. Successive calls of the function start adding this 1 from later on in the set in order to avoid creating duplicate solutions. This is illustrated in Figure A7, where the triple+ expansion of 6 is found. Each box represents the set of solutions generated by one call of the recursive function. The arrows show the recursive calls and what number in the sequence to start adding the 1 to. The order that the solutions are found is shown top-to-bottom on the right side of the diagram. Finding a double expansion is a much simpler process of counting down with one number and up with another.

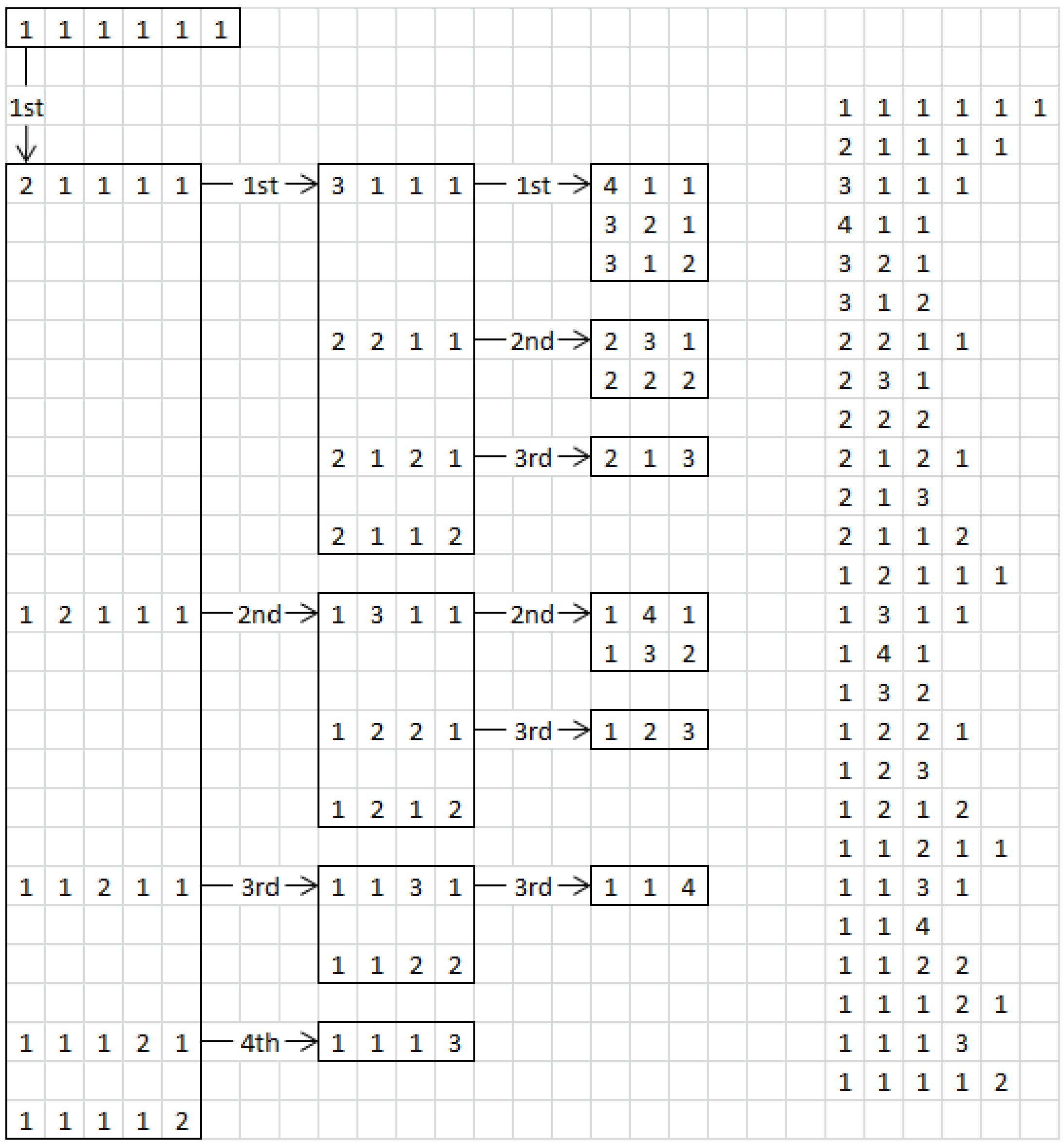


Figure A7: Finding the triple+ expansion of 6.

Appendix D: Full Results from 3D Puzzle Testing

The times given here are the average of 100 results, unless an individual time was very large in which case the test was only performed once. The number of linesolvers and bad guesses is the same every time for a given puzzle and solver.

12×7×3 (141 lines, 252 cells)	Linesolvers	Bad guesses	%	Time
Brute	153	47	30.72	<0.001s
Hier	134	17	12.69	<0.001s
Hier+Pre	23	3	13.04	<0.001s
Heur	117	40	34.19	0.003s
Heur+Pre	24	4	16.67	<0.001s
24×12×6 (504 lines, 1728 cells)				
Brute	712	303	42.56	0.002s
Hier	932	478	51.29	0.012s
Hier+Pre	128	13	10.16	0.003s
Heur	525	165	31.43	0.019s
Heur+Pre	114	3	2.63	0.006s
36×18×9 (1134 lines, 5832 cells)				
Brute	2292	1278	55.76	0.011s
Hier	2792	1755	62.86	0.154s
Hier+Pre	525	162	30.86	0.025s
Heur	1394	469	33.64	0.077s
Heur+Pre	359	39	10.86	0.023s

Table A1: Dino test results.

12×11×9 (339 lines, 1188 cells)	Linesolvers	Bad guesses	%	Time
Brute	544	257	47.24	0.001s
Hier	396	94	23.74	0.003s
Hier+Pre	263	78	29.66	0.002s
Heur	455	90	19.78	0.015s
Heur+Pre	233	44	18.88	0.008s
24×23×17 (1351 lines, 9384 cells)				
Brute	3054	1733	56.75	0.024s
Hier	3286	1773	53.96	0.511s
Hier+Pre	1717	765	44.55	0.208s
Heur	2494	768	30.79	0.180s
Heur+Pre	1351	334	24.72	0.118s
36×34×25 (2974 lines, 30600 cells)				
Brute	8835	5516	62.43	7.135s
Hier	10384	6668	64.21	24.585s
Hier+Pre	7706	4808	62.39	63.424s
Heur	6602	2383	36.10	34.235s
Heur+Pre	4151	1410	33.97	25.424s

Table A2: Duck test results.

12×9×3 (171 lines, 324 cells)	Linesolvers	Bad guesses	%	Time
Brute	228	110	48.25	<0.001s
Hier	201	67	33.33	0.001s
Hier+Pre	37	4	10.81	<0.001s
Heur	180	80	44.44	0.007s
Heur+Pre	32	4	12.50	0.002s
24×16×4 (544 lines, 1536 cells)				
Brute	1128	723	64.10	0.003s
Hier	980	533	54.39	0.018s
Hier+Pre	112	12	10.71	0.003s
Heur	638	317	49.69	0.029s
Heur+Pre	88	12	13.64	0.005s
36×25×7 (1327 lines, 6300 cells)				
Brute	2778	1588	57.16	450s (7.5m)
Hier	3057	1808	59.14	1406s (23.4m)
Hier+Pre	590	167	28.31	1240s (20.7m)
Heur	1821	907	49.81	1584s (26.4m)
Heur+Pre	529	140	26.47	1218s (20.3m)

Table A3: Horse test results.

12×4×6 (144 lines, 288 cells)	Linesolvers	Bad guesses	%	Time
Brute	166	88	53.01	<0.001s
Hier	176	94	53.41	0.001s
Hier+Pre	50	1	2.00	<0.001s
Heur	199	74	37.19	0.006s
Heur+Pre	74	14	18.92	0.002s
24×10×12 (648 lines, 2880 cells)				
Brute	3957	3261	82.41	16.827s
Hier	2598	1650	63.51	76.856s
Hier+Pre	1344	698	51.93	62.325s
Heur	1916	1026	53.55	42.756s
Heur+Pre	1148	605	52.70	37.268s
36×12×17 (1248 lines, 7344 cells)				
Brute	Out of memory	-	-	-
Hier	-	-	-	-
Hier+Pre	-	-	-	-
Heur	-	-	-	-
Heur+Pre	-	-	-	-

Table A4: Jeep test results.

12×12×9 (360 lines, 1296 cells)	Linesolvers	Bad guesses	%	Time
Brute	800	511	63.88	0.001s
Hier	871	561	64.41	0.004s
Hier+Pre	181	68	37.57	0.001s
Heur	634	307	48.42	0.015s
Heur+Pre	138	30	21.74	0.004s
23×24×18 (1398 lines, 9936 cells)				
Brute	13371	11624	86.93	0.047s
Hier	12420	10583	85.21	0.856s
Hier+Pre	4241	2781	65.57	0.192s
Heur	5951	4175	70.16	0.264s
Heur+Pre	2565	1631	63.59	0.359s
34×36×26 (3044 lines, 31824 cells)				
Brute	-	-	-	>11 hours
Hier	-	-	-	-
Hier+Pre	-	-	-	-
Heur	-	-	-	-
Heur+Pre	-	-	-	-

Table A5: Knot test results.

12×12×12 (432 lines, 1728 cells)	Linesolvers	Bad guesses	%	Time
Brute	43200	43200	100	2.122s
Hier	432	432	100	0.024s
Hier+Pre	432	432	100	0.024s
Heur	432	432	100	0.055s
Heur+Pre	432	432	100	0.055s
24×24×24 (1728 lines, 13824 cells)				
Brute	172800	172800	100	6013s (100m)
Hier	1728	1728	100	32.676s
Hier+Pre	1728	1728	100	47.318s
Heur	1728	1728	100	99.249
Heur+Pre	1728	1728	100	102.821
36×36×36 (3888 lines, 46656 cells)				
Brute	388800*	388800*	100*	-
Hier	3888*	3888*	100*	-
Hier+Pre	3888*	3888*	100*	-
Heur	3888*	3888*	100*	-
Heur+Pre	3888*	3888*	100*	-

Table A6: Checkerboard test results.

*These tests were not completed, these are the expected results.

Appendix E: Full Results from 2D Puzzle Testing

The times given here are the average of 100 results. The number of linesolvers and bad guesses is the same every time for a given puzzle and solver.

Bucks	Linesolvers	Bad guesses	%	Time	Solved?
Brute	5000	4213	84.26	0.008s	Partly
Hier	541	446	82.44	0.011s	Partly
Hier+Pre	513	430	83.82	0.008s	Partly
Heur	309	216	69.90	0.014s	Partly
Heur+Pre	313	225	71.88	0.014s	Partly
Cat					
Brute	363	249	68.60	0.001s	Yes
Hier, Hier+Pre	595	458	76.97	0.003s	Yes
Heur, Heur+Pre	313	193	61.66	0.007s	Yes
Dancer					
Brute	64	41	64.06	<0.001s	Yes
Hier, Hier+Pre	35	15	42.86	<0.001s	Yes
Heur, Heur+Pre	47	23	48.94	0.002s	Yes*
Edge					
Brute	2100	2100	100	0.003s	No
Hier, Hier+Pre	21	21	100	<0.001s	No
Heur, Heur+Pre	21	21	100	<0.001s	No
Knot					
Brute	435	329	75.63	0.001s	Yes
Hier, Hier+Pre	1012	907	89.62	0.006s	Yes
Heur, Heur+Pre	696	564	81.03	0.032s	Yes**
Skid					
Brute	291	209	71.82	<0.001s	Yes
Hier	373	282	75.60	0.002s	Yes
Hier+Pre	335	254	75.82	0.002s	Yes
Heur	249	148	59.44	0.007s	Yes
Heur+Pre	246	151	61.38	0.006s	Yes
Smoke					
Brute	4000	3969	99.22	0.011s	Partly
Hier, Hier+Pre	169	136	80.47	<0.001s	Partly
Heur, Heur+Pre	130	109	83.85	0.003s	Partly
Swing					
Brute	1392	939	67.46	0.005s	Yes
Hier, Hier+Pre	4299	3846	89.46	0.034s	Yes
Heur, Heur+Pre	1165	643	55.19	0.035s	Yes

Table A7: 2D puzzle test results. Results are listed together if the pre-pass procedure made no difference.

* The heuristic exit clause had to be modified so that instead of failing when the consecutive bad guess count reaches the size of the queue, it fails when it reaches **2×** the size of the queue.

** The heuristic exit clause had to be modified so that instead of failing when the consecutive bad guess count reaches the size of the queue, it fails when it reaches **4×** the size of the queue.

References

Literature and Online

Batenberg, K.J. and Kosters, W.A. 2004. A discrete tomography approach to Japanese puzzles. *Proceedings of the 16th Belgian-Dutch Conference on Artificial Intelligence*. Groningen, October 21-22. pp. 243-250.

Batenburg, K.J. and Kosters, W.A. 2009. Solving nonograms by combining relaxations. *Pattern Recognition*. 42(8): pp. 1672-1683.

Batenburg, K.J., et al. 2009. Constructing simple nonograms of varying difficulty. *Pure Mathematics and Applications*. 20(1): pp. 1-15.

Cepero, M. 2013. *Voxel Physics*. [online]. Available from: http://procworld.blogspot.co.uk/2013/12/voxel-physics.html [Accessed 19 April 2016].

Eisemann, E. and Décoret, X. 2006. Fast scene voxelization and applications. *Proceedings of the 2006 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. Redwood City, CA: March 14-17. pp. 71-78.

Evans, A. and Kirczenow, A. 2011. Two uses of voxels in LittleBigPlanet 2's graphics engine. [presentation slides]. *Advances in Real-Time Rendering in 3D Graphics and Games SIGGRAPH 2011*. Vancouver, August 7-11.

Goldner, K., Kimura, R. and Schwartz, A. [no date]. *Nonokenken: solving puzzles using integer programs*. University research. University of Washington.

Henstra, S. 2011. *From image to nonogram: construction, quality and switching graphs*. Master's thesis. Leiden University.

Heubach, S. and Mansour, T. 2004. Compositions of n with parts in a set. *Congressus Numerantium*. 168(1): pp. 127-143.

Huang, J. et al. 1998. An accurate method for voxelizing polygon meshes. *Proceedings of the IEEE 1998 Symposium on Volume Visualization*. Research Triangle Park, NC: October 19-20. pp.119-126.

Johnston, E. 2014. *Picross-3D-Solver*. [online]. Available from: https://github.com/epmjohnston/Picross-3D-Solver [Accessed 18 April 2016].

Kaufman, A. and Shimony, E. 1986. 3D scan-conversion algorithms for voxel-based graphics. *Proceedings of the 1986 Workshop on Interactive 3D Graphics*. New York, NY: October 23-24. pp. 45-75.

On-Line Encyclopedia of Integer Sequences. 1996. *A000295 - Eulerian numbers*. [online]. Available from: http://oeis.org/A000295 [Accessed 26 March 2016].

Ortiz-García, E.G., et al. 2007a. Automated generation and visualization of picture-logic puzzles. *Computers & Graphics*. 31(5): pp. 750-760.

Ortiz-García, E.G., et al. 2007b. Solving Japanese puzzles with heuristics. *Proceedings of the 2007 IEEE Symposium on Computational Intelligence and Games*. Honolulu, April 2-5. pp. 224-231.

Pestana, A. 2014. *Volumetric lights*. [online]. Available from: http://www.alexandre-pestana.com/volumetric-lights/ [Accessed 17 April 2016].

Pomeranz, D., Raziel, B. and Rabani, R. 2010. *The BGU Nonograms Project*. [online]. Available from: https://www.cs.bgu.ac.il/~benr/nonograms/ [Accessed 18 April 2016].

Silva, L.F.M.S., Pamplona, V.F. and Comba, J.L.D. 2009. Legolizer: a real-time system for modelling and rendering LEGO representations of boundary models. *25th SIBGRAPI Conference on Graphics, Patterns and Images*. Rio de Janeiro, October 11-15. pp. 17-23.

Simpson, S. 2000. *How it works*. [online]. Available from: http://www.lancs.ac.uk/~simpsons/nonogram/howitworks [Accessed 31 October 2015].

Szucki, M. and Suchy, J. 2012. A voxelization based mesh generation algorithm for numerical models used in foundry engineering. *Metallurgy and Foundry Engineering*. 38(1): pp. 43-54.

Takeshige, M. 2015. *The Basics of GPU Voxelization*. [online]. Available from: https://developer.nvidia.com/content/basics-gpu-voxelization [Accessed 19 April 2016].

Thon, S., Gesquière, G. and Raffin, R. 2004. A low cost antialiased space filled voxelization of polygonal objects. *GraphiCon 2004 Proceedings*. Moscow, September 6-10. pp. 71-78.

Ueda, N. and Nagao, T. 1996. *NP-completeness results for nonogram via parsimonious reductions*. Department of Computer Science, Tokyo Institute of Technology.

van Rijn, J.N. 2012. *Playing games: the complexity of Klondike, Mahjong, Nonograms and Animal Chess*. Master's thesis. Leiden University.

Wang, W. and Tang, M. 2014. Simulated annealing approach to solve nonogram puzzles with multiple solutions. *Procedia Computer Science*. 36(1): pp. 541-548.

Westerdiep, A. 2010. *Voxel*. [online image]. Available from: http://www.drububu.com/miscellaneous/voxelizer/voxel.gif [Accessed 30 October 2015].

Wiggers, W. 2004. A comparison of a genetic algorithm and a depth first search algorithm applied to Japanese nonograms. *1st Student Conference on IT, University of Twente*. Enschede, June 14.

Wolter, J. 2012. *The 'pbnsolve' Paint-by-Number Puzzle Solver*. [online]. Available from: http://webpbn.com/pbnsolve.html [Accessed 18 April 2016].

Wolter, J. 2013a. *Survey of Paint-by-Number Puzzle Solvers*. [online]. Available from: http://webpbn.com/survey/ [Accessed 07 April 2016].

Wolter, J. 2013b. *Comparison of Solvers on the n-Dom Puzzles*. [online]. Available from: http://webpbn.com/survey/dom.html [Accessed 19 April 2016].

Wolter, J. 2013c. *Effect of Line Solution Caching on Pbnsolve Run-times*. [online]. Available from: http://webpbn.com/survey/caching.html [Accessed 20 April 2016].

Wolter, J. [no date]. *Advanced Puzzle Solving Techniques*. [online]. Available from: http://webpbn.com/index.cgi?page=solving.html [Accessed 15 April 2016].

Yu, C., Lee, H. and Chen, L. 2011. An efficient algorithm for solving nonograms. *Applied Intelligence*. 35(1): pp. 18-31.

Software and Third-Party Libraries

Assimp Development Team. 2014. *Assimp - Open Asset Import Library*. (Version 3.1.1). [source code]. Available from: http://assimp.sourceforge.net/ [Accessed 17 November 2015].

Barrett, S. 2003. *IProf*. (Version 0.2). [source code]. Available from: http://silverspaceship.com/src/iprof/ [Accessed 20 April 2016].

Cornut, O. 2015. *ImGui*. (Version 1.46). [source code]. Available from: https://github.com/ocornut/imgui [Accessed 20 April 2016].

ephtracy. 2016. *Magica Voxel*. [software]. Version 0.97.2. Available from: https://ephtracy.github.io/index.html?page=mv_main [Accessed 20 April 2016].

Microsoft Corporation. 2016. *DirectX Tool Kit*. (Version January 5th 2016). [source code]. Available from: https://github.com/Microsoft/DirectXTK [Accessed 20 April 2016].

Vauter, A. 2014. *Nonogram Solver*. [software]. Version 9. Available from: http://www.ticalc.org/archives/files/fileinfo/461/46105.html [Accessed 17 April 2016].

Wootten, J. 2012. *Gnonograms*. [software]. Version 0.9.8-beta. Available from: https://github.com/jeremypw/gnonograms [Accessed 17 April 2016].

Games

Mario's Picross. 1995. [computer game]. Nintendo Game Boy. Jupiter.

NonoBlock. 2014. [computer game]. Android, iOS, Linux, Mac and Windows PC. Dansl.

Picross DS. 2007. [computer game]. Nintendo DS. Jupiter.

Picross 3D. 2009. [computer game]. Nintendo DS. HAL Laboratory.

Picross 3D 2. 2015. [computer game]. Nintendo DS. HAL Laboratory.

Resogun. 2013. [computer game]. Sony PlayStation 4. Housemarque.

Voxatron. 2011 (public alpha). [computer game]. Linux, Mac and Windows PC. Lexaloffle Games.

Models and Puzzles

Dino was distributed with Assimp as 'fromtruespace_bin32.x' and was originally from the discontinued modelling software TrueSpace. *Duck* was distributed with Assimp as 'duck.dae'. *Knot* was distributed with Assimp as 'pyramob.3DS' and was created by Virlux. The *Horse* and *Jeep* models were also distributed with Assimp but their original sources are given here.

Beyer, K. 2005. *WW2 German Kubalwagon* (renamed to *Jeep*). [3d model]. Available from: http://www.katsbits.com/download/models/ww2-german-kubalwagon.php [Accessed 07 April 2016].

Chugunny, K. 2016a. *Japanese crossword <<Brigantine>>*. [online puzzle]. Available from: http://www.nonograms.org/nonograms/i/166 [Accessed 15 April 2016].

Chugunny, K. 2016b. *Japanese crossword <<Tiny goat>>*. [online puzzle]. Available from: http://www.nonograms.org/nonograms/i/4353 [Accessed 22 March 2016].

Chugunny, K. 2016c. *Japanese crossword <<Titmouse>>*. [online puzzle]. Available from: http://www.nonograms.org/nonograms2/i/6130 [Accessed 18 April 2016].

Tsiantas, E. [date unknown]. *Horse*. [3d model]. Available from: http://telias.free.fr/models_md2_menu.html [Accessed 07 April 2016].