Scaling the Tiles: A Puzzle Game

Report

[redacted] Supervised by Vincent van Oostrom

BSc. Computer Science

School of Informatics and Engineering, University of Sussex

2025/26

Statement of Originality

This report is submitted as part requirement for the degree of Computer Science at the University of Sussex. It is the product of my own labour except where indicated in the text. The report may be freely copied and distributed provided the source is acknowledged. I hereby withhold permission for a copy of this report to be loaned out to students in future years.

Signed:

01/05/2026

Acknowledgements

I would like to thank Dr Vincent van Oostrom for his support and guidance throughout this project.

Summary

This report presents the design and implementation of a novel edge-matching puzzle game inspired by Wang Tiles. Unlike traditional edge-matching puzzles, the game introduces a scaling mechanic that allows players to resize tiles, enabling connections between edges made up of sequences of different colours. The report covers the full development process, including initial research, system design, implementation, and evaluation of the final product.

Table of Contents

1 Introduction

This project aims to create a novel edge-matching puzzle inspired by the concept of Wang Tiles/McMahon squares. Puzzle pieces will have sequences of one or more colours on each side, and to accommodate this, the player will be able to arbitrarily scale the tiles to fit each other.

This results in a complex game with theoretically numerous possible solutions and outcomes. In some cases, an infinite tiling pattern may arise where tiles are recursively scaled to fit together. To simplify the game, we could rule out recursion completely by forcing finite use-counts. However, an interesting challenge would be to find a way to visualise these infinite tilings by converging to a solution.

1.1 Objectives

1.1.1 Primary Objectives

- Research how puzzles are created and solved
- Design a graphical user interface that allows players to drag and drop puzzle pieces onto a grid
- Create an intuitive mechanism for scaling pieces to fit together
- Allow for the player to adjust the sequence length and use-count of puzzle pieces.

1.1.2 Extension Objectives

- Create an automatic puzzle solver
- Create an automatic puzzle generator
- Create a library of pre-built puzzles with varying difficulty levels
- Visualise infinite tiling patterns by converging to a recursive solution
- Conduct testing and evaluation of the game
- Investigate the complexity of the game

1.2 Motivation

As an enthusiast of puzzle games, I enjoy exploring original ideas that bring a fun twist to classic formats and challenge players to think in new ways. In preparing for this project, I researched many puzzles but found no instance of an edge-matching puzzle with a resizing mechanic, or with tiles containing more than one colour on each side. Such a mechanic does not seem to exist in current commercial or academic puzzles, making the project both original and of research interest.

Allowing the player a degree of freedom by manipulating puzzle pieces leads to more complex and engaging gameplay, allowing them to experiment with solutions resulting in satisfying or, just as likely, frustrating outcomes.

Additionally, the nature of this project will allow me to exercise a number of skills I have gained throughout my course, providing me with the beneficial experience of managing a project through the full software development lifecycle. This includes applying theoretical knowledge from algorithms and complexity theory to analyse and classify the computational hardness of the puzzle. On the practical side, the project will develop my skills in software design and architecture, GUI development and programming. Throughout the course of the project, I aim to employ best practice in software development.

1.3 Report Structure

Section 1 of the report contains the introduction, outlining the aims and motivation of the project.

Section 2 covers the professional and ethical considerations of the project, including adherence to the BCS Code of Conduct and the ethical framework for user testing.

Section 3 presents background research, examining the theoretical foundations of edge-matching puzzles, related games, and the computational complexity of the problem domain.

Section 4 contains the requirements analysis, defining the functional and non-functional requirements of the system.

Section 5 details the design of the system, covering the system architecture, data model, core mechanics, user interface and accessibility.

Section 6 describes the implementation of the system, including the choice of programming environment, the realisation of the design, and challenges encountered during development.

Section 7 presents the evaluation of the project, including complexity analysis, system testing, and user testing.

Section 8 concludes the report, reflecting on the outcomes of the project and suggesting directions for future work.

2 Professional Considerations

2.1 Ethical Considerations

The project contains little risk as a mostly research-based undertaking, however, to evaluate the playability of the game, user testing will be carried out towards the end of the project. At all points ethical practice will be followed and the process will align with the BCS Code of Conduct [1] and the University of Sussex Research Ethics Guidance for projects, ensuring professionalism and respect for participants.

All participants will receive clear information about the study and provide informed consent before testing. Participation is voluntary, and individuals may withdraw at any time without consequence. No personally identifiable data will be collected, and responses will be anonymised and stored securely, accessible only to the student and supervisor.

As the project does not involve experiments beyond basic usability testing, it is considered low risk, therefore no formal ethical approval is required. A signed copy of the user testing compliance form is included in Appendix A – Ethical Compliance Form

2.2 BCS Code of Conduct

The relevant sections of the BCS Code Of Conduct are:

1. **Public Interest**

1.1. have due regard for public health, privacy, security and wellbeing of others and the environment;

The game contains no distressing content, and no information is collected about users.

1.2. have due regard for the legitimate rights of third parties;

Godot is distributed under the MIT licence, which permits unrestricted use, modification, and redistribution, making it appropriate for academic and commercial development.

1.3. conduct your professional activities without discrimination on the grounds of sex, sexual orientation, marital status, nationality, colour, race, ethnic origin, religion, age or disability, or of any other condition or requirement;

1.4. promote equal access to the benefits of IT and seek to promote the inclusion of all sectors in society wherever opportunities arise.

In line with the above two sections, there will be no discrimination against participants in the user study, and participation will be entirely voluntary. Throughout development, reasonable steps will be taken to accommodate users with disabilities where possible, and inclusion will be prioritised to ensure equal access and accessibility for all participants.

2. **Professional Competence and Integrity**

2.1. only undertake to do work or provide a service that is within your professional competence;

2.2. NOT claim any level of competence that you do not possess;

2.3. develop your professional knowledge, skills and competence on a continuing basis, maintaining awareness of technological developments, procedures, and standards that are relevant to your field;

Through discussions about the project with the supervisor, it is agreed that the objectives are within my technical capabilities, and I will continue to develop relevant knowledge throughout the project lifecycle,

2.4. *ensure that you have the knowledge and understanding of legislation and that you comply with such legislation, in carrying out your professional responsibilities;*

I have familiarised myself with relevant legislation and will ensure compliance with it throughout the project.

2.7. *reject and will not make any offer of bribery or unethical inducement.*

Participation in the study is voluntary and no incentive will be provided.

3 Background Research

3.1 Theory

Two-dimensional puzzles can be split into two categories: pictorial and apictorial, as shown in Figure 1. Pictorial puzzles consider visual aspects of the pieces, such as images or colours, in the construction of the puzzle, whereas apictorial puzzles are those that only consider geometric shape of pieces [2].

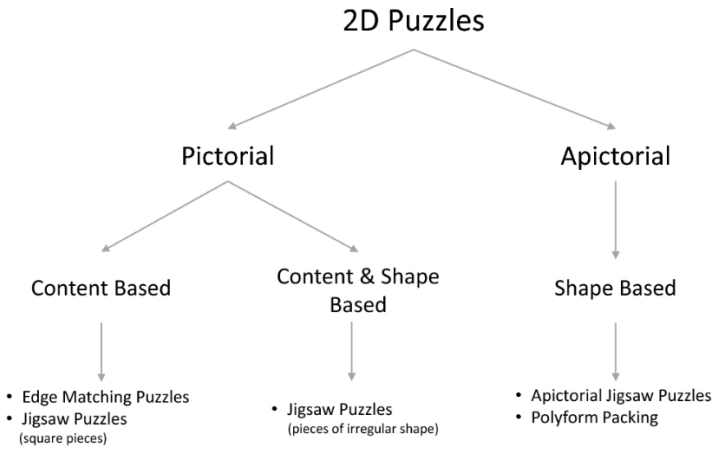

FIGURE 1: MAIN CATEGORIES OF TWO-DIMENSIONAL PUZZLES [3]

Pictorial content-based puzzles, which don't take into account the shape of the tiles, include jigsaw puzzles with square pieces as well as a subtype of puzzles known as edge matching puzzles. This variation involves arranging a given set of tiles so that the coloured edges of adjacent tiles match.

Wang Tiles, introduced by Hao Wang in 1961 as a model for studying decidability in logic [4], are square tiles with coloured edges that are placed edge-to-edge without rotation such that adjacent edges match. While Wang proposed that any set of tiles capable of tiling the infinite plane could do so periodically, this was disproved by Berger in 1966, who constructed an aperiodic set of Wang Tiles and proved that the domino problem is undecidable [5]. This means no algorithm can exist that correctly answers this question for all possible tile sets.

Berger's proof of undecidability has significant implications for the complexity of edge-matching puzzles. Determining whether a given set of Wang Tiles can tile the infinite plane is undecidable — no algorithm can correctly answer this question for all possible tile sets [6]. For finite puzzles, however, the problem is decidable, and has been shown to be NP-complete. The computational complexity of the proposed game is discussed further in Section 7.1.

3.2 Similar Games

To support the design process, similar games were examined and evaluated for both their visual presentation and game design.

3.2.1 Tetravex

Tetravex was a game originally created in a digital format for the Windows Entertainment Pack 3 by Scott Ferguson. It involves arranging a set of squares in a grid (defaulted to 3x3) so that the edges of adjacent pieces match.

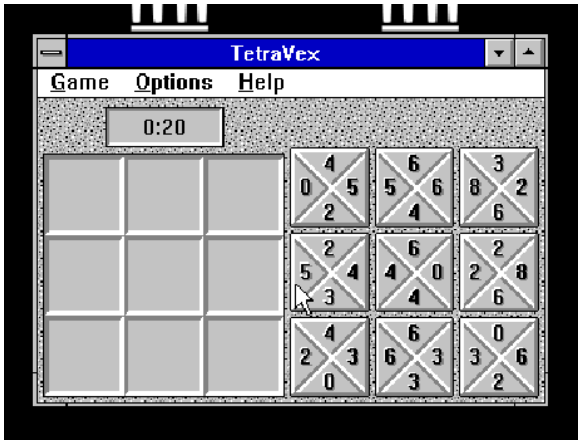

FIGURE 2: THE ORIGINAL TETRAVEX VIA AN EMULATOR ON ARCHIVE.ORG

The original version of the game, pictured in Figure 2, used numbers for each edge, but more modern versions include corresponding colours for each number. The original version is visually simple, with almost no colours used in the interface, and menus typical of the operating system and interface standards at the time [6].

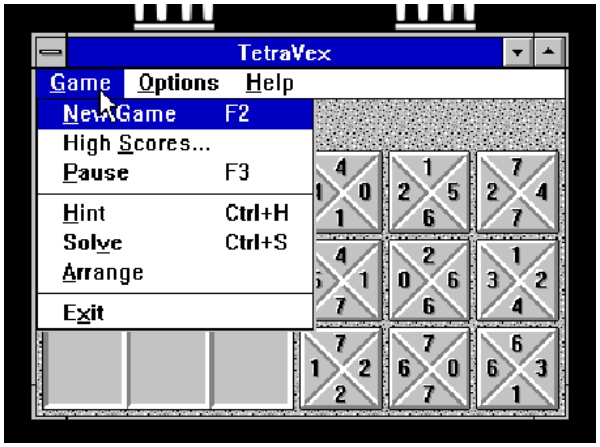

FIGURE 3: TETRAVEX GAME MENU

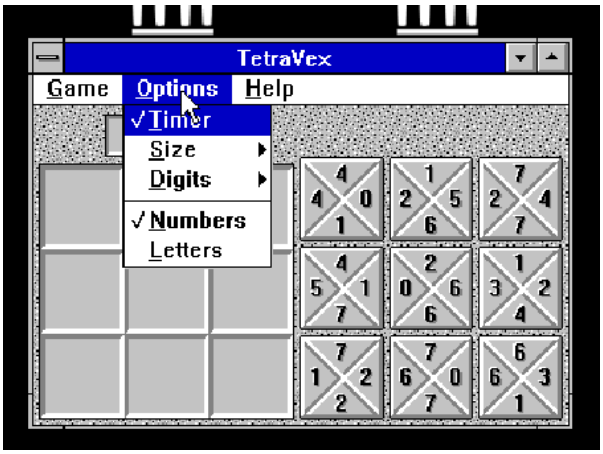

FIGURE 4: TETRAVEX OPTIONS MENU

The game menu (Figure 3) features options to receive a hint and to solve a puzzle. The "hint" option places one tile its correct place. A similar hint mechanic could be implemented to help the player without revealing the full solution.

Figure 4 shows that the size of the grid and number of digits used in the puzzle are adjustable settings, both of which are factors that affect the complexity of the puzzle, thereby adjusting the difficulty.

Increasing the number of digits in the game allows for fewer possible matches for each tile, making it possibly easier to reach a solution. A larger grid-size, however, makes it more difficult due to an increased number of tiles and therefore more false matches. These factors present a good basis for balancing the difficulty in the proposed game.

FIGURE 5: A MODERN VERSION OF TETRAVEX, VIA SMART-GAMES.ORG

Figure 5 shows a newer version of the same game [7] , which is very similar to the original but lacks some of the more complex features and options, with a bigger focus on visuals. This version uses colours for each of the edges as well as the numbers, making it more visually pleasing and easier to spot possible matches.

The original idea for this project was to use matching coloured edges for the tiles, but adding an option for displaying numbers would make the game more accessible for those who cannot distinguish between the colours as well.

3.2.2 Jigsaw Explorer

Jigsawexplorer.com [8] is one of many online versions of the traditional jigsaw puzzle, where the player must reconstruct an image using pieces with uniquely shaped edges.

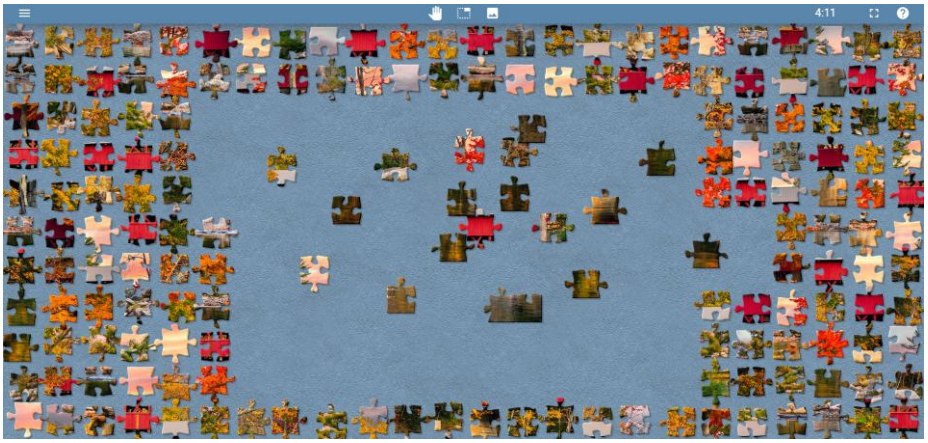

FIGURE 6: A SCREENSHOT OF THE WEB GAME JIGSAWEXPLORER.COM

The game loads a puzzle with pieces scrambled and arranged in a border-like style around a blank space in the middle, as shown in Figure 6. Pieces can be dragged and dropped wherever the player pleases, and snap together when one is placed close enough to an adjacent piece. There is audio feedback when pieces snap together.

Pieces snapping into place when dragged close enough provides satisfying feedback. In the proposed game, tiles could snap to valid positions when released nearby rather than requiring pixel-perfect placement.

Notably, connected pieces can be dragged around together as a group, allowing the player arrange pieces in the play area. However, once connected, pieces cannot be disconnected without resetting the puzzle. In a jigsaw puzzle, this is not a problem, as pieces can only connect with in the confirmed solution order. In this project, that is not the case, so grouped tiles will require a way to be separated if the player wishes to do so.

The scrambled pieces arranged around the border of the play area is an intuitive starting layout. Tiles could similarly be displayed in a sidebar or holding area rather than overlapping the board.

4 Requirements Analysis

Based on the background research a set of requirements for the game was created.

4.1 Functional Requirements

TABLE 1: FUNCTIONAL REQUIREMENTS

Requirement	Description
Drag and drop tiles	Users can drag and drop tiles onto any empty space within the play area
Tiles snap together	Tiles automatically snap to neighbours if their edges match when aligned and scaled.
Tile groups	Tiles snapped together are in a group that can be moved around.
Tile scaling	Users can resize tiles along horizontal and/or vertical axes
Edge matching validation	The game checks that adjacent tile edges contain matching colours in the correct scale.
Completed puzzle detection	The game detects when all tiles are connected with valid edges and no gaps remain
Reset puzzle	Users can reset the puzzle to its original unsolved state.
Tile deletion	Users can remove placed tiles and return them to the tile selection area.
Default puzzle	The game launches with a simple puzzle using single-colour edges to introduce gameplay.
Multiple solutions support	The game accepts any valid completed layout; it does not assume only one solution.
Zoom & pan camera	Users can zoom in/out and pan around the puzzle board.
Adjustable sequence length	Users can configure the number of colours per edge before starting a puzzle
Puzzle difficulty presets	Users can choose preset difficulty levels that adjust puzzle size and colour variation.
Timer	Players can optionally enable a timer while solving.
Puzzle generator	Users can generate new puzzles by defining the size, colour sequence, and use count constraints.
Puzzle solver	Users can select an option to show the solution to the current puzzle
Infinite tiling visualisation	The game can visualise a recursively scaling infinite tiling by converging tile sizes toward zero

4.2 Non-functional Requirements

TABLE 2: NON-FUNCTIONAL REQUIREMENTS

Requirement	Description
Usability	The interface must be intuitive, allowing players to drag, scale, and place tiles without extensive instructions.
Performance	The game must remain smooth and responsive when generating, displaying, and solving puzzles.
Reliability	The puzzle generator must always produce at least one valid solution. The game must correctly detect completion.
Accessibility	Colour sequences should include optional numbers to support colour-blind users.
Maintainability	Code must be modular and well-documented, for future updates or extensions.
Error-handling	The game should correctly handle invalid inputs, such as placing a tile outside the play area or extreme scaling attempts.

5 Design

As Sommerville writes in Software Engineering [9] p.197, software design is a vital stage of development in which software components and their relationships are identified, based on a set of requirements. It is very closely linked to the implementation stage, laying the foundation for the programmatic process. The following sections document the design decisions made for this project.

5.1 System architecture

The first step of the design process is to define the components of the game and how they interact with one another. Breaking the system down modularly helps us to understand the function of each part, identify dependencies between components, and makes it easy to map the distinct responsibilities onto the structure of the chosen programming environment.

The core feature of the system is the player's ability to move and resize tiles. There will need to be a component that manages these tiles and their connections, and one that tracks the overall state of the puzzle as it evolves.

Figure 7 shows a high-level components diagram illustrating these isolated components and how data flows between them during the core gameplay loop.

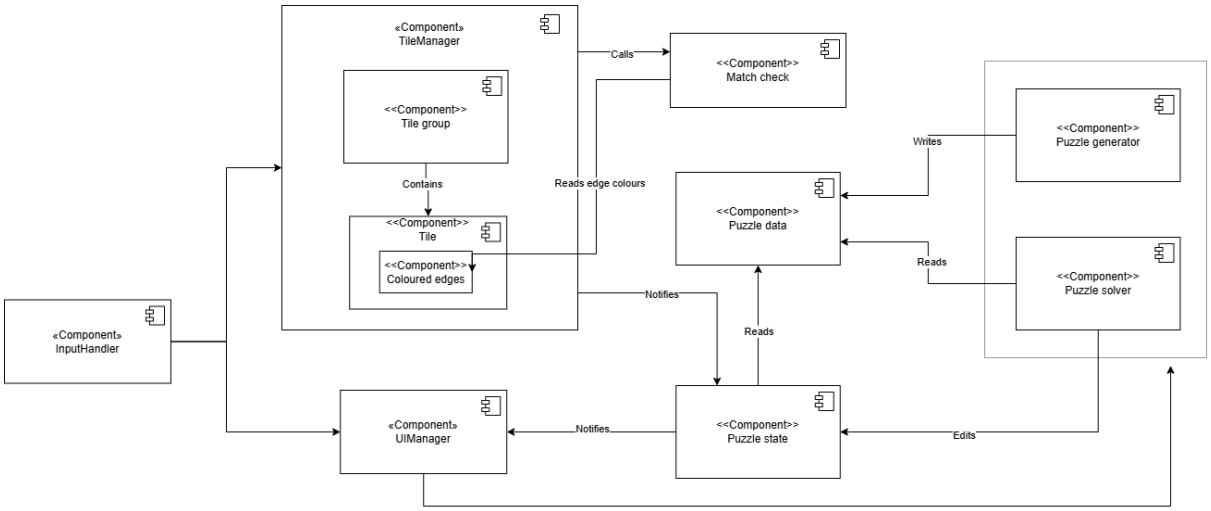

FIGURE 7: HIGH-LEVEL COMPONENTS OF THE SYSTEM

The arrows in the Figure 7 highlight how certain components will reference or call others. Interaction with the system is initiated through the *InputHandler*, which represents the system's ability to detect when the player interacts with UI or click on tiles.

The *TileManager* is responsible for managing all tile instances on the board. Tiles are composed of *edges* with sequences of colours and can be attached to one another to form *TileGroups* that are moveable as a unit. Snapping of tiles is managed by the *MatchChecker*, which is invoked by the *TileManager* when a tile is moved close to another. It verifies if the connection is valid by reading the edge sequences of each tile.

A *PuzzleState* component acts as a state machine and tracks the position, scale and connections of all tiles on the board, and communicates this information to the UI manager in order to trigger the win screen when the puzzle is completed.

PuzzleData is the data structure that represents a particular puzzle and its configuration, defining the set of available tiles and their edge sequences. The *PuzzleGenerator* writes to

the *PuzzleData* when given parameters by the player via the UI. Puzzle*Solver* reads from *PuzzleData* and modifies the *PuzzleState* to produce a solution.

The *UIManager* is responsible for rendering all interface elements such as menus, the tile selection area, and overlays such as the completion screen, as well as translating UI interactions into commands for the rest of the system.

5.2 Data model

With an overarching view of the system, we can begin to look more closely at the distinct units within it and map out their specific structures and functions. Figure 8 shows a class diagram of the core data model.

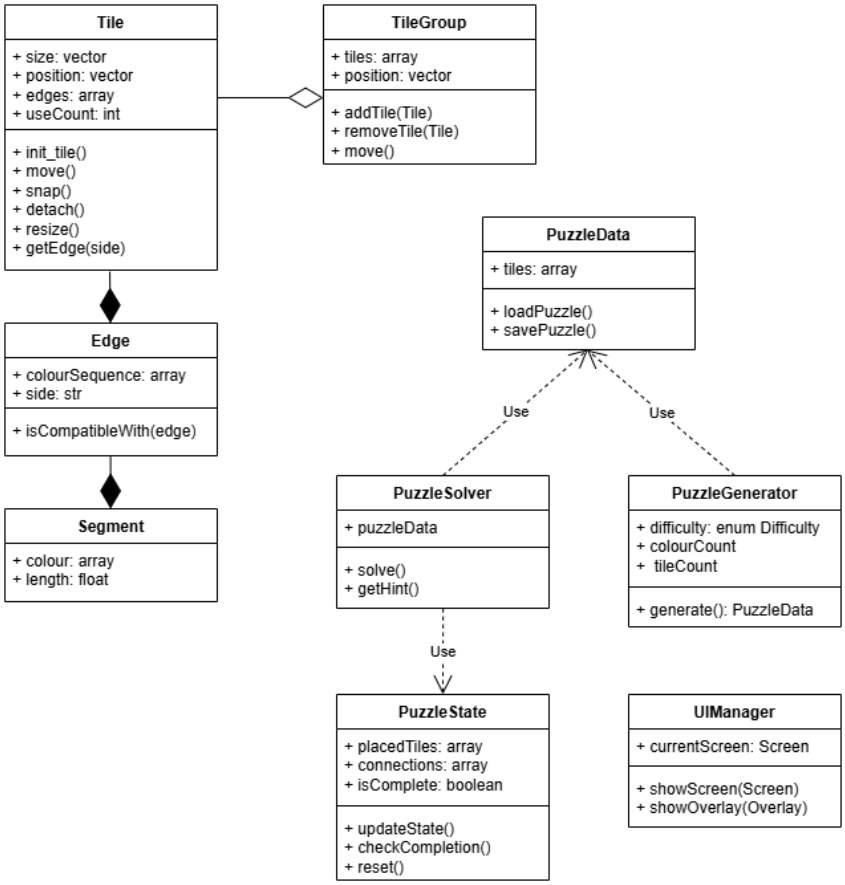

FIGURE 8: CLASS DIAGRAM

The following entities make up the data model:

Segment - The smallest unit. Each segment has a colour and a physical length. Segments are owned by an edge and cannot exist independently.

Edge - Each tile has four edges, one per side. An edge is composed of an ordered sequence of one or more segments and has a direction.

Tile - A tile is a square game object defined by its four edges, its position in the play area and its scale. Tiles are the primary objects the player interacts with.

TileGroup - When two or more tiles are snapped together, they form a TileGroup. The group maintains references to its member tiles and allows them to be moved as a unit. This abstraction is necessary because once tiles are connected, dragging one should move all connected tiles together.

PuzzleData - a static data structure representing a particular puzzle configuration. It defines the set of available tiles, their edge sequences, and use-count constraints. It is written to by the PuzzleGenerator and read by the game and PuzzleSolver. It represents the puzzle, not the current state of the player's progress.

PuzzleState - the dynamic version of PuzzleData. It tracks everything that changes during play: the current position and scale of each tile, which tiles are connected to which, and whether the puzzle has been completed. It is updated continuously as the player interacts with the board and is monitored by the UIManager to trigger state changes such as the completion screen.

5.3 Snapping Logic

The system must follow a set of rules when deciding whether to allow two tiles to connect. Since tiles can have multiple segments on each edge, it must be ensured that all segments adjacent to each other upon connection are the same colour and same length.

Figure 9 shows a flow diagram of checks that must be performed when determining whether a potential 'snap' should be allowed. It essentially checks four different factors when a tile is dropped, and if one of them is false, the snap is rejected.

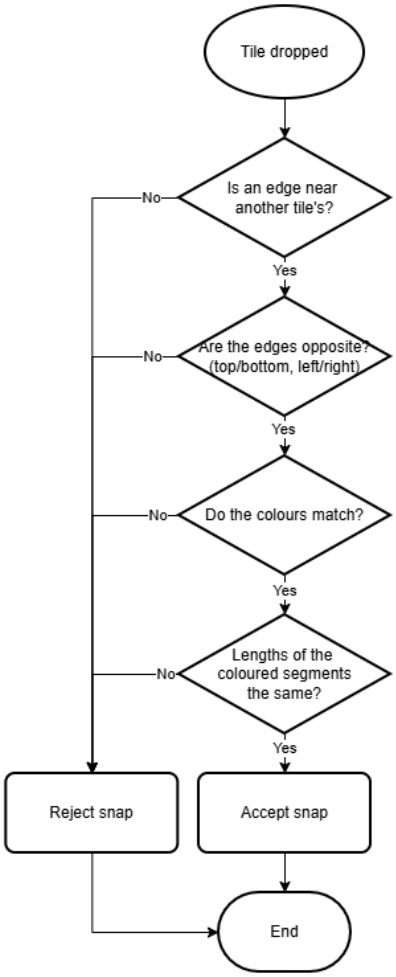

FIGURE 9: FLOW OF SNAPPING LOGIC

The rules are as follows:

- **The segment must be near a segment from another tile.** In this case, 'near' refers to whether the edge of the tile is within a certain threshold distance of an edge of another tile. This proximity check exists because the player is allowed to move tiles freely, meaning there is no grid to define adjacency.
- **The edges must be facing opposite directions.** A segment at the top of a tile must only be able to snap to a segment at the bottom of another tiles. The opposite-edge check prevents errors like a tile's top snapping to another tile's top.
- **The colours of the segments must match.** This is the core matching rule inherited from traditional edge-matching puzzles.
- **The lengths of the segments must be approximately the same.** This enforces the scaling constraint, ensuring that only segments of equal physical size are considered compatible.

With respect to the final rule, the allowance of segments that are 'approximately' the same length to snap together is a consequence of the decision not to use a grid. Since players will be able to resize tiles freely to any scale they like, it will be virtually impossible to resize a tile to the exact same length or width as another. The solution is for the system to allow segments within a certain length threshold to snap together, assuming that the players intention was for them to connect, and to automatically adjust the length of the edge to make an exact match upon snapping.

These checks are performed when a tile is being dragged, and visual feedback should be present to show the player the possible snaps. Compatibility will also have to be checked with all surrounding tiles that a tile will be touching after the snap is applied.

When tiles are snapped together, they form *TileGroups*, which can be treated as a singular unit that the player can move and connect with other *TileGroups*.

5.4 Resizing

In this game, the function of resizing tiles exists to allow players to match segment lengths across tiles with different sequence lengths. For example, a tile with 2 segments on its right edge must be scaled to twice the height of a tile with 1 segment on its left edge so that they may connect. Combined with the ability to use a tile more than once, this allows for players to fit tiles together in interesting ways. For example, one tile could be used repeatedly to tile an area by being resized, like in Figure 10.

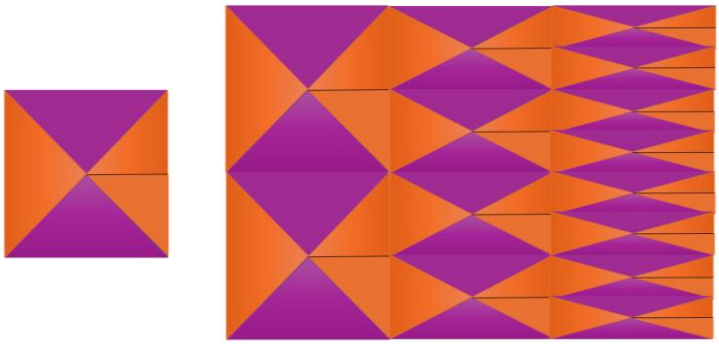

FIGURE 10: AN EXAMPLE OF ONE TILE USED REPEATEDLY

In some cases, an infinite tiling pattern may arise, where tiles are recursively scaled to fit together, as seen in Figure 11.

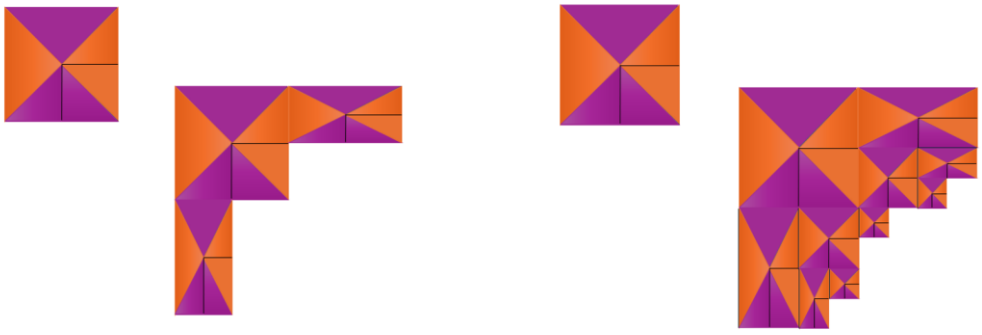

FIGURE 11: EXAMPLE OF POSSIBLE INFINITE TILING

In such cases, it would be interesting to find a way to visualise the concept of *converging* to a solution as theoretical recursion could extend indefinitely, effectively covering the plane. Implementing a method to approximate or represent this recursion could enhance the gameplay.

There must be some constraints on resizing, like a minimum and maximum scale, to avoid cases of extreme scaling. To allow more freedom in puzzle creation, tiles can be resized on either axis or both.

5.5 Puzzle Generator

Ansótegui et al. propose a method for generating edge-matching puzzles that are guaranteed to have at least one solution [10] p.12. It randomly assigns colours to edges of puzzle pieces and builds tokens from those colour assignments. This algorithm is based on a common approach used in many puzzles, where the solution is constructed first and puzzle pieces are derived directly from it. A similar approach can be used in the generation of puzzles for the project, with important modifications.

This implementation deviates as tiles are not guaranteed to be the same size. Therefore, instead of having a grid with tiles assigned to each partitioned space, a grid of 'units' will be produced. This grid exists only during generation and is not visible to the player. Each internal boundary is assigned a single random colour from the available colour set.

Tiles are then created by partitioning the unit grid into non-overlapping rectangles using a rectangle partitioning algorithm. Each tile's edges are derived from the unit boundaries it crosses. As such, a tile edge that spans multiple unit boundaries will have multiple colour segments. Since adjacent tiles share the same underlying unit boundaries, their touching edges are guaranteed to reference the same colours, ensuring compatibility. Solvability is therefore guaranteed without any additional validation step.

5.6 User Interface

This section focuses on the visual elements of the game that the player interacts with, including menus, the play area, and in-game overlays.

5.6.1 Navigation Flow

The game is organised into a small number of distinct screens, each corresponding to a different state of the application. Figure 12 shows the navigation flow between these screens.

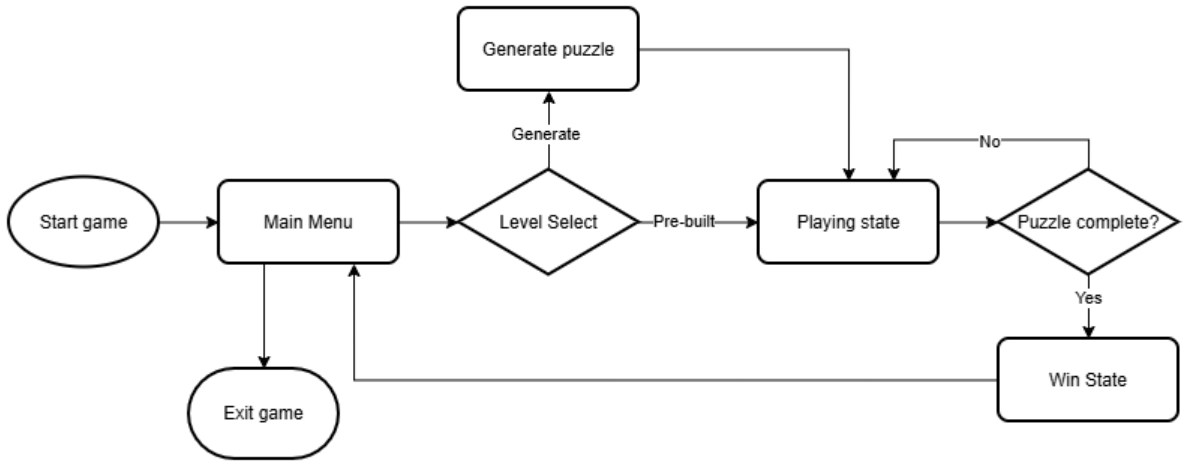

FIGURE 12: NAVIGATION FLOW

The player begins at the Main Menu, which provides options to start the game, access settings, or exit. From here they proceed to the Level Select screen, where they can choose from pre-built puzzles of varying difficulty or opt to generate a new puzzle using custom parameters. Selecting a puzzle transitions the player into the Play screen, which is the primary game view containing the tile area, the board, and in-game controls such as reset, hint, and zoom. Upon successfully completing a puzzle, a Level Complete overlay is displayed, showing a completion message and optionally the time taken. From here the player is returned to Main screen to choose another puzzle or exit.

5.6.2 Mock-ups

It was decided that menu screens should be minimal, presenting only the essential navigation options. The Level Select screen will display puzzle options simply in a list format. A separate option will be available to open the puzzle generator with configurable parameters.

Most of the Play screen will be occupied by the canvas, on which tiles can be freely moved, scaled, and connected. A panel on one side houses the available tiles not yet placed on the board. A minimal toolbar provides access to reset, hint, and a duplicate button. If the timer is enabled, it is displayed unobtrusively in a corner of the screen.

Figure 13 and Figure 14 show rough mock-ups that were created before implementation for the play screen.

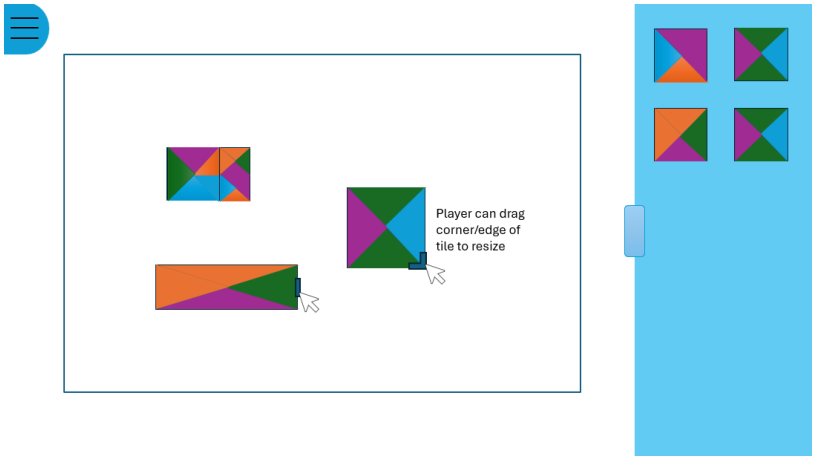

FIGURE 13: MOCK-UP OF RESIZING FEATURE

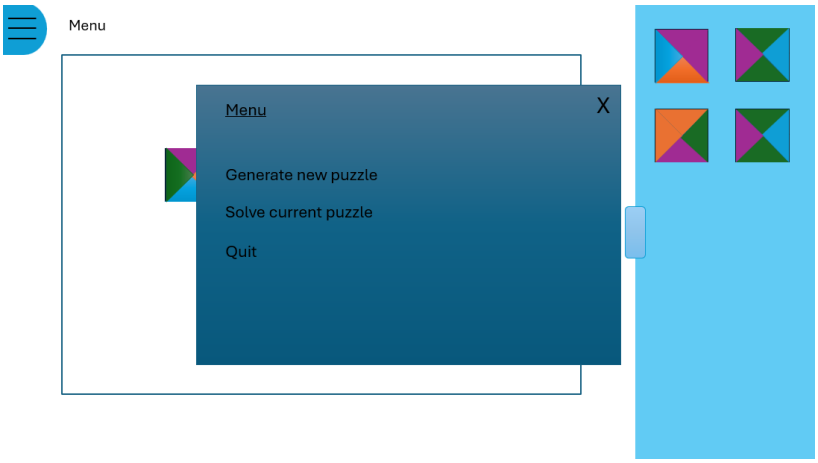

FIGURE 14: MOCK-UP OF IN-GAME MENU

5.7 Accessibility

The Equality Act (2010) defines a disability as 'a physical or mental impairment that has a 'substantial' and 'long-term' negative effect on your ability to do normal daily activities' [11]. The World Health Organization estimates that 16% of the world's population is disabled [12]. As a prominent demographic, it is important to dedicate resources during development towards ensuring that disabled people can enjoy video games, without having to compromise an aspect of the experience.

A core feature of the proposed game centres around the colours of the tiles. This creates a considerable barrier for people who are colour-blind. Colour-blindness is the reduced ability to see certain colours, or the differences between them. The condition can range from complete colour deficiency to more common types like red-green colour-blindness.

Molina-Lopez and Medina-Medina [13] propose a set of design patterns for improving game accessibility for colour-blind player. The most relevant ones include avoiding reliance on colour alone, or allowing options to configure alternative colours and associating colours with a certain icon or symbol.

In line with the recommended measures, three alternative colour palettes were designed: one for protanopia (red-blind), one for deuteranopia (green-blind), and one for tritanopia (blue-yellow blind). Each palette was designed and verified using the colour blindness simulation tool at davidmathlogic.com [14], which allows designers to preview how colours appear under different types of colour vision deficiency and adjust them accordingly.

Additionally, a high-contrast palette was designed for players with low vision or sensitivity to colour, using colours with high luminance difference. Table 3 shows the chosen colour palettes and their hex codes, as well as the justification for each choice.

TABLE 3: COLOUR PALETTES

Default palette	#FF6B6B	#FFD93D	#6BCB77	#4D96FF
Protanopia-friendly palette	#FF7F50	#FFE066	#2DB27D	#1F77D0
Justification	This palette reduces reliance on red–green contrast by shifting red tones towards orange. Colours are chosen to differ in both hue and brightness, making them easier to distinguish even when red perception is limited.			
Deuteranopia-friendly palette	#D55E00	#F0E442	#009E73	#0072B2
Justification	This palette avoids combinations that rely on red–green differences. Instead, colours are selected with strong contrast in brightness and distinct hues, making them more easily distinguishable.			
Tritanopia-friendly palette	#E4572E	#F3A712	#59C3C3	#3B4CC0
Justification	This palette uses a wider spread of hues and incorporates contrast in both saturation and luminance to ensure colours remain distinguishable even when blue–yellow differentiation is impaired.			
High-contrast palette	#D7263D	#F4D35E	#2EC4B6	#3A86FF
Justification	This palette prioritises strong luminance contrast between colours, making it suitable for users with low vision or general visual sensitivity.			

6 Implementation

This section covers how the proposed design was implemented within the chosen programming environment, highlighting the key functions of the system and how they map onto the above designs.

Throughout development, maintaining clear and manageable code was a consistent focus, in line with the maintainability requirement outlined in 4.2. Functions and variables were given descriptive name, so their purpose is generally clear without needing excessive inline comments. Comments were still included where the logic is less obvious.

Each script was designed to handle a specific responsibility, following the modular structure described in Section 5.1. This helps limit the impact of changes, as modifications to one part of the system are less likely to affect others. Communication between components was mainly handled using Godot's signal system, which keeps scripts loosely coupled and easier to work with.

6.1 Programming environment

The engine chosen for the project is Godot Engine 4.4, a free and open-source game engine distributed under the MIT licence. Godot was selected for several reasons. Its built-in scene and node system provides a natural way to represent game objects such as tiles, edges, and segments, mapping closely onto the data model defined in Section 5.2. The engine includes a dedicated 2D framework with support for input handling and collision detection, which are relevant to the core mechanics of the game. Additionally, Godot's scripting language, GDScript, is syntactically similar to Python and well suited to rapid prototyping.

Alternative engines were considered, including Unity, which is a more widely used engine with a larger community and more extensive documentation. However, Godot was preferred for its lighter weight, more straightforward 2D workflow, and prior experience using it in coursework.

6.2 Tiles

As outlined in Section 5.2, each edge of a tile is represented as an ordered sequence of discrete Segment objects, each occupying a physical region along the edge of a tile. In Godot, Segments are implemented using Area2d/CollisionShape2D objects, allowing the engine's built-in overlap detection to determine when two segments are occupying the same physical region.

This maps onto the snapping logic described in Section 5.3. When a tile is dragged, Godot reports which collision shapes are overlapping, and the match checker reads the colour and size properties of the overlapping segments to determine compatibility.

Each tile is implemented as an Area2D node, containing a CollisionShape2D that defines its physical boundary and a Button node that handles player interaction such as clicking and dragging.

The tile's size and shape are defined programmatically at initialisation through the *init_tile* function. The segments are produced from a dictionary *side_colours,* which contains a list of colours corresponding to each cardinal edge of the tile. Segments are then created as coloured wedges of equally proportioned lengths along each side. This makes it easier to implement compatibility checking as opposed to tiles with segments of varying proportions.

Tile data is stored in a *TileInfo* resource, a custom Resource that saves tile properties for saving and loading puzzles. This was introduced later in development once puzzle persistence became a requirement. The resource script is pictured in Figure 15.

```
1   extends Resource
2   class_name TileInfo
3
4   @export var tile_size: Vector2
5   @export var start_position: Vector2
6   @export var side_colours: Dictionary = {
7       "north" = ["red", "green"],
8       "east" = ["red", "green"],
9       "south" = ["red"],
10      "west" = ["red"]
11      }
```

FIGURE 15: THE TILEINFO RESOURCE SCRIPT

At initialisation, segments are generated programmatically by the *draw_triangles* function. For each of the four edges, the function divides the edge length equally among the colours in that direction's sequence, computing a start position and step vector based on the tile's dimensions.

Metadata is attached to each segment Area2D including its direction, colour, parent tile reference, length, and midpoint position, allowing other tiles to read all necessary compatibility information without needing direct access to the tile's internal structure.

The result is a tile bearing a resemblance to the classic Wang tile, but with multiple coloured wedges along each edge, as seen in Figure 16.

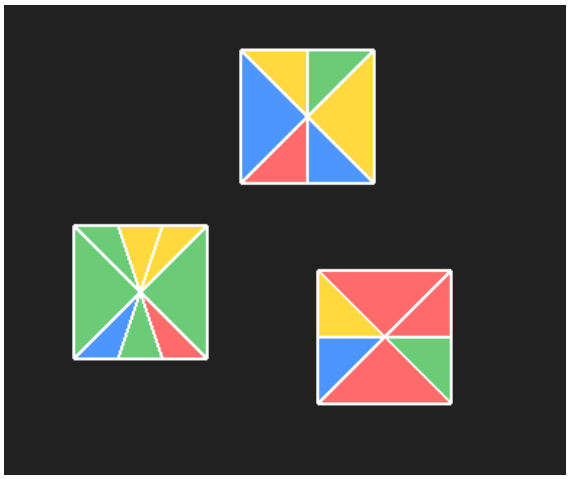

FIGURE 16: EXAMPLE OF TILES WITH MULTIPLE COLOURED WEDGES

6.2.1 Movement and Interaction

The drag-and-drop mechanic for the tiles was implemented using Godot's built-in Button node, which provides mouse input callbacks without requiring manual input polling. Tile interaction is implemented using a simple state machine with three states: idle, dragging, and resizing. This abstraction separates interaction modes cleanly and prevents conflicting behaviours (e.g. resizing while dragging), improving maintainability and reducing input ambiguity (see Figure 17).

Continuous position updates are handled per frame to ensure smooth motion. A drag offset is calculated at the start of interaction so that the tile maintains its relative position to the cursor, avoiding unintuitive snapping of the tile's centre to the mouse position. The cursor shape changes to visually indicate that the tile is being dragged. If the tile belongs to a *TileGroup*, the whole group is moved as a unit.

This approach prioritises responsiveness and user control, which directly supports the usability requirement defined in Section 4.2.

Holding Shift while clicking a tile removes it from its *TileGroup* without entering the dragging state, allowing the player to separate a tile from a connected group.

```
68  func _process(_delta: float) -> void:
69      # dragging or resizing happens here
70      match state:
71          State.DRAGGING:
72              if get_parent().is_in_group("tile_group"):
73                  get_parent().global_position = get_global_mouse_position() - drag_offset
74              else:
75                  position = get_parent().to_local(get_global_mouse_position()) - drag_offset
76              _update_highlights()
77          State.RESIZING:
78              if handle_active != null:
79                  _resize_tile_from_mouse(handle_active)
80
81      _clamp_to_screen()
```

FIGURE 17: THE _PROCESS FUNCTION, WHICH IS CALLED EVERY FRAME

A single tile can be selected by right-clicking, which highlights it in gold and enables the duplicate and delete buttons in the toolbar. Duplicating a tile creates a copy. Tile groups can also be duplicated as a unit, preserving the relative positions of all member tiles. Deselection occurs automatically when the player clicks anywhere other than a tile.

The ability to delete and duplicate is important in cases where puzzles require a tile to be used more than once in the solution.

6.2.2 Snapping

When a tile is dropped, it checks for a valid snap by iterating over all tiles registered with the puzzle manager and comparing every segment pair between the dragged tile and each other tile. A snap candidate is accepted if the segments match (see section 5.3 for matching conditions).

Where multiple valid candidates exist, the closest by distance is selected as the best snap. This ensures predictable behaviour from the user's perspective, as the visually closest match is prioritised.

Before the snap is finalised, *_is_position_valid* checks that the tile's new position does not overlap any other tile and that no colour conflicts exist with adjacent tiles that would not be part of the snap. This prevents the player from snapping a tile into a position that would create an invalid adjacency with a third tile.

If the matched segments are close in length but not exactly equal, a resize is triggered via *_snap_resize*, which adjusts the tile's width or height so that its segment length exactly matches the target, ensuring consistent geometry without requiring players to perfectly resize tiles.

```gdscript
func _on_button_button_up() -> void:
    # when player drops tile, set state to idle, and apply snap if one found
    state = State.IDLE
    set_cursor()
    _clear_all_highlights()
    var snap = _find_best_snap()
    if snap == null:
        return

    # check if bounding rect doesn't overlap other tiles in target group
    if not _is_position_valid(snap["target_position"], snap["other_tile"]):
        return

    _apply_snap_position(snap["target_position"])

    if snap["other_tile"] != null:
        _handle_tile_connection(snap["other_tile"])
    else:
        connectSound.play()

    if snap["resize_needed"]:
        _snap_resize(snap["resize_length"], snap["resize_dir"])
```

FIGURE 18: THE _ON_BUTTON_BUTTON_UP FUNCTION, WHICH IS CALLED WHEN A TILE IS DROPPED

6.2.2.1 Snap Preview

While a tile is being dragged, *_update_highlights* runs each frame, identifying all segment pairs that would form a valid snap at the current position. Matching segment outlines on both tiles are highlighted in cyan with an increased line width, giving the player real-time visual feedback about potential connections before committing to a drop.

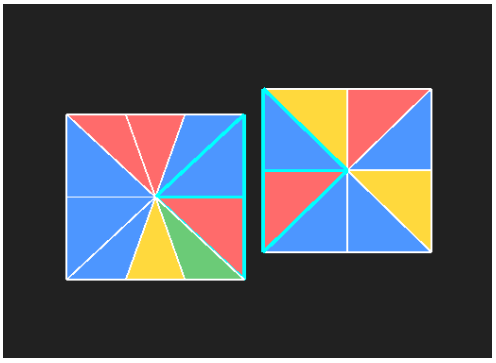

FIGURE 19: TILES WITH HIGHLIGHTED SEGMENTS TO SHOW VALID SNAP

6.2.3 Tile groups

When two tiles snap together, *_handle_tile_connection* manages the transfer of tiles into groups. Based on the existing status of tiles, it either creates a new group to place both tiles into, or transfers tiles into one group.

This ensures that connected components are always represented as a single *TileGroup* node regardless of the order in which connections are made. An example is pictured in Figure 20.

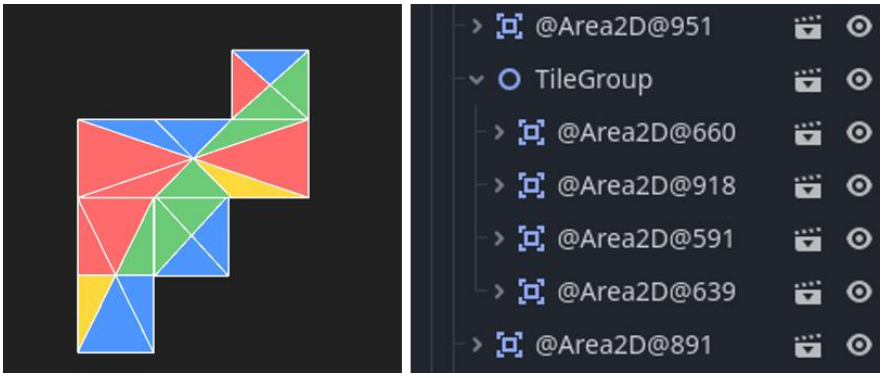

FIGURE 20: A TILE GROUP

6.2.4 Resizing

Eight resize handles are created around each tile at initialisation-- one on each edge and corner. Handles respond to mouse hover by changing the cursor shape to the appropriate directional resize cursor.

When a handle is clicked, the state transitions to RESIZING and the active handle is recorded. The new tile size is calculated based on the mouse position relative to the opposite anchor point, which is the point on the tile that remains fixed during the resize. For edge handles, only the relevant axis is resized; for corner handles, both axes are adjusted simultaneously. A minimum size of 50 units is enforced on each axis to prevent tiles from becoming too small to interact with. After resizing, the various components are all updated to reflect the new dimensions.

6.3 Puzzle Manager

The *PuzzleManager* is the central coordinating component of the system, responsible for managing the lifecycle of puzzles, tiles, and UI states.

6.3.1 Loading and Spawning

Puzzles are represented as custom *PuzzleData* resources (Figure 21) containing a list of *TileInfo* objects. When a puzzle is loaded, *load_puzzle* clears any existing tiles and iterates over the puzzle's tile data, spawning a tile scene for each entry. In play mode, tiles are positioned in a tray area below the board rather than at their solution positions, laid out in a grid calculated from the viewport width and tile size. Duplicate tiles are filtered out using a signature string before spawning, ensuring the player receives only unique tiles.

```
1   extends Resource
2   class_name PuzzleData
3
4   @export var puzzle_name: String
5   @export var tiles: Array
6   @export var unit_size: Vector2 = Vector2(100, 100)
7   @export var board_size: Vector2
8   @export var board_colours: Dictionary = {
9       "north": ["red", "green"],
10      "south": ["red", "blue"],
11      "east":  ["red", "green"],
12      "west":  ["red", "blue"],
13  }
```

FIGURE 21: THE PUZZLEDATA RESOURCE

6.3.2 Puzzle Selection and Generation

The player can choose from a set of pre-built puzzles or generate a new one by selecting a difficulty level. Selecting a difficulty invokes the puzzle generator with a predefined set of parameters tuned for each level.

6.3.3 Puzzle Authoring Tool

To support the creation of pre-built puzzles, an authoring mode was developed for use during development. This tool displays controls for manually placing and configuring tiles, editing the board frame's size and colour sequences, and saving the resulting layout as a *PuzzleData* resource file. It is not accessible to the player and was used exclusively to produce the pre-built puzzle library included with the game. However, with further development, this tool could be made accessible to players as a dedicated puzzle creation mode, increasing replayability.

6.4 Pre-built puzzles

Three puzzles were constructed using the authoring tool. The puzzles were designed to be of lower difficulty, providing an easy introduction to the novel mechanics of the game, while exemplifying some of the interesting cases that can be expressed within the context of the game.

Puzzle 1 introduces the concept of multi-segmented tiles that need to be resized by presenting a simple board with two tiles that must be scaled correctly before being placed (Figure 22). Puzzle 2 (Figure 23) builds on this by introducing tile duplication, requiring the player to duplicate one of the provided tiles to complete the solution. Finally, puzzle 3 (Figure 24) combines the mechanics by providing a board and a single tile which must be repeatedly duplicated and resized to fill the space, encouraging the use of the group duplication feature.

FIGURE 22: PRE-BUILT PUZZLE 1

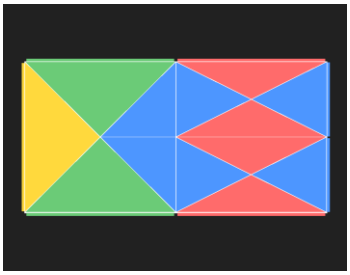

FIGURE 23: PRE-BUILT PUZZLE 2

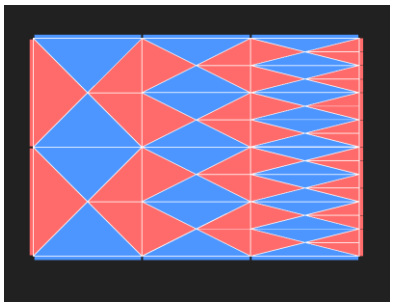

FIGURE 24: PRE-BUILT PUZZLE 3

6.5 Puzzle Generator

The puzzle generator produces valid puzzles by deriving tiles from a pre-coloured grid, guaranteeing solvability by construction. Generation happens in three stages: assigning colours to the unit grid boundaries, partitioning the grid into tile regions, and building tile data from those regions.

Initially, a greedy rectangle-packing algorithm was used to partition the unit grid. Starting from the top-left, the algorithm grew rectangles by expanding right and down randomly. Whilst simple to implement, this approach produced heavily biased layouts. Tiles in the top-left tended to be large, with the remaining space fragmented into thin strips along the bottom and right edges. This made puzzles feel unbalanced and reduced the variety of tile sizes, which is important for making the scaling mechanic feel meaningful. BSP (Binary Space Partitioning) was adopted as a replacement because it subdivides the entire space simultaneously rather than greedily consuming it, producing a more natural and varied distribution of tile shapes and sizes.

Figure 25 illustrates the process of deriving tiles from a unit grid. Colours are randomly assigned to each edge in step 1, then the BSP algorithm divides the grid into rectangles

(steps 2 and 3). In the example pictured, the grid was split into eight sections of varying sizes which fit together. Finally, the generator derives tiles and their colour sequences from the grid.

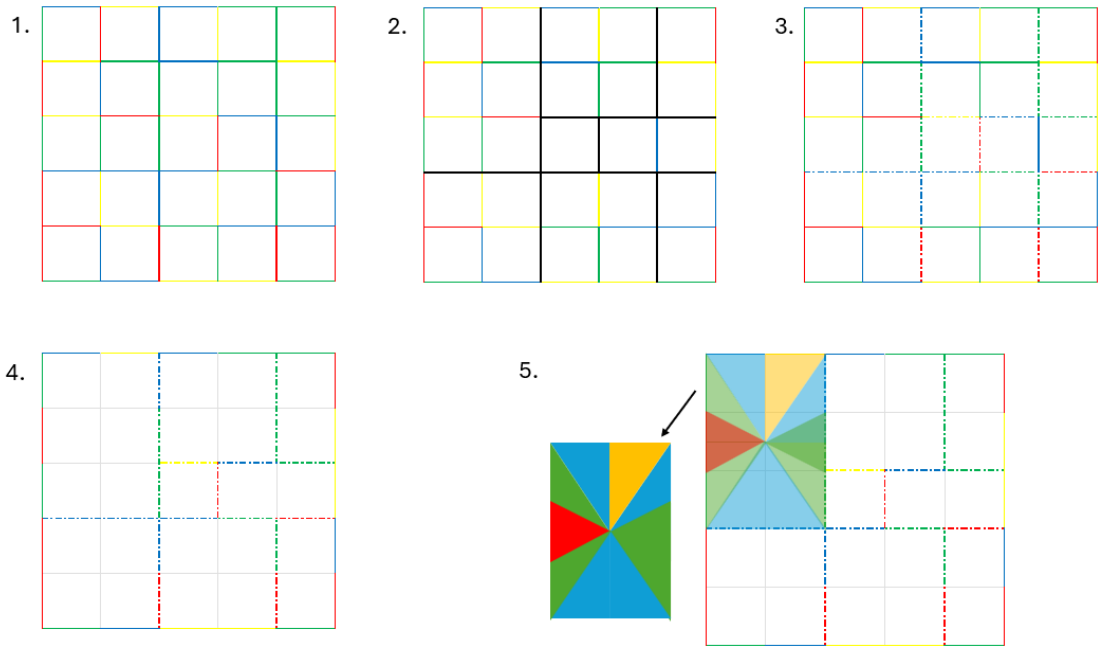

FIGURE 25: HOW TILES ARE FORMED FROM THE UNIT GRID

To increase potential for ambiguity in puzzles (see section 6.5.1), a repeating motif pattern is used when assigning colours to the unit, rather than randomly assigning colours. This results in colours repeating across the grid in a structured pattern and therefore, a higher chance of multiple tiles sharing the same edge colour sequences, which increases ambiguity in the puzzle and makes it harder for the player to determine the correct connections to make.

The unit grid is partitioned into rectangular tile regions using a recursive BSP algorithm. Starting from the full grid, *_bsp_split* repeatedly divides each region into two until a stopping condition is met. The algorithm favours splitting along the longer dimension, with the split point chosen randomly within the valid range. This means that wider regions are more likely to be split vertically and taller regions are more likely to be split horizontally, resulting in ideally more equal distribution of tile sizes.

Splitting stops when the maximum recursion depth is reached, when the region is too small to split further, or probabilistically based on the region's area relative to the total grid. Larger regions are less likely to stop early, encouraging finer subdivision where there is space while letting small regions stay intact. These conditions produce a natural-looking variety of tile sizes.

```gdscript
func _partition_into_tiles():
    # partition the grid into rectangles using a recursive bsp algorithm
    tile_regions.clear()
    var root = Rect2i(0, 0, unit_grid_size.x, unit_grid_size.y)
    _bsp_split(root, 0)

func _bsp_split(region: Rect2i, depth: int):
    # recursive function splits grid into two repeatedly
    if _should_stop(region, depth):
        tile_regions.append(region)
        return

    var split = _choose_split(region)
    if split == null:
        tile_regions.append(region)
        return

    print("SPLIT ", depth, " COMPLETE, MOVING TO NEXT")
    _bsp_split(split.a, depth + 1)
    _bsp_split(split.b, depth + 1)
```

FIGURE 26: THE RECURSIVE _BSP_SPLIT FUNCTION

Each region is converted into *TileInfo* by *_build_tile_from_region*. Edge colour sequences are read directly from the boundary arrays. The outermost boundary entries define the board frame's colour sequences. These are stored in the *PuzzleData* and used by the board to render its edge segments, which the player must satisfy when placing tiles.

The finished puzzle is saved into a *PuzzleData* resource and returned to the puzzle manager for loading.

6.5.1 Difficulty

The difficulty of generated puzzles is controlled by tuning five parameters passed into *generate_puzzle*: the unit grid size, unit world size, number of colours, BSP depth, and motif period. The three difficulty presets-- easy, medium, and hard-- were determined empirically by generating puzzles with various parameter combinations and evaluating the results by hand.

A larger unit grid produces more tiles and longer edge sequences, increasing the number of possible placements the player must consider. Increasing the number of colours reduces the likelihood of false matches, generally resulting in an easier puzzle with less possible configurations to try. A higher BSP depth allows finer subdivision, producing more tiles of smaller and more varied sizes, which increases the complexity of the scaling mechanic since more resizing is required to align segments.

The motif period controls how colours are distributed across the boundary grid. A smaller motif period means colours repeat more frequently across the grid, increasing the number of segments with matching colours and therefore the number of plausible but incorrect tile placements, increasing ambiguity.

The three presets were configured as follows:

TABLE 4: PARAMETERS FOR THE THREE DIFFICULTY PRESETS

Parameter	Easy	Medium	Hard
Unit grid size	5x5	7x7	10x10
Unit world size	130x130	100x100	70x70
Colours	4	4	5
BSP depth	3	5	7
Motif period	4	3	3
Max segments	3	3	2

6.6 User Interface

The user interface was implemented in line with the minimalist design outlined in 5.6. The main menu (Figure 27) displays the title of the game prominently and presents three options: Play, Settings, and Exit. The background across all menu screens makes use of Godot's Particles2D node to generate a calm ambient particle effect, adding subtle motion to the scene without distracting from the interface elements.

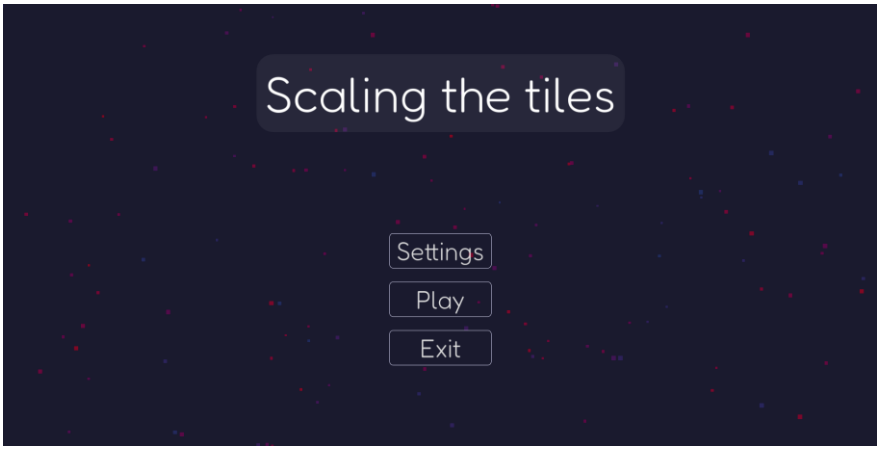

FIGURE 27: THE MAIN MENU

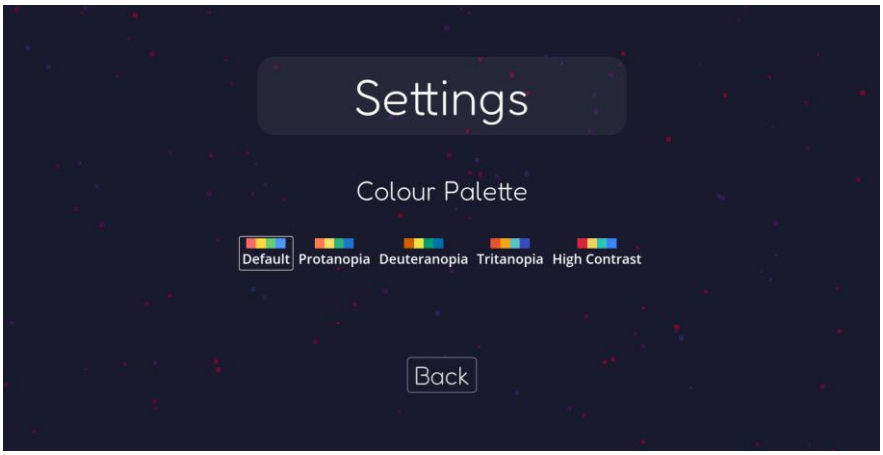

FIGURE 28: THE SETTINGS MENU

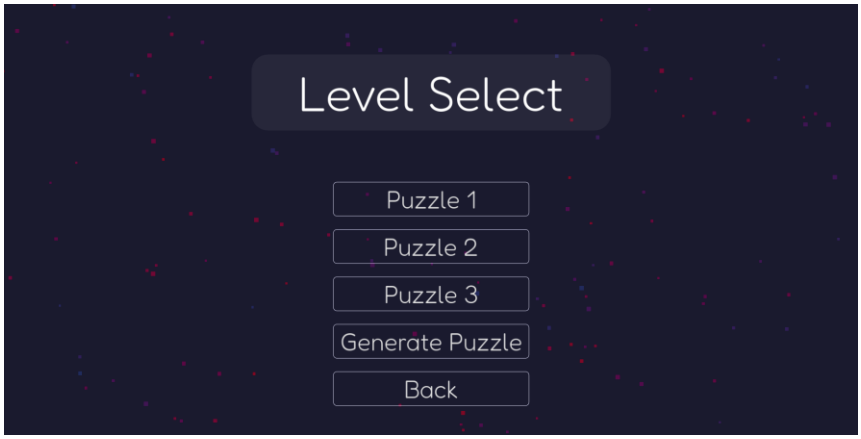

FIGURE 29: THE LEVEL SELECT MENU

The Settings and Level Select menu screens are similar in appearance, with minimal distractions. The Settings menu (Figure 28) allows the player to select a colour palette from the five options designed in Section 5.7. Switching palette updates a global *ColourPalette* singleton.

The Level Select screen (Figure 29) presents the player with the choice of three pre-built puzzles or the option to generate a new puzzle. Selecting a difficulty level opens a popup confirming the selection before invoking the puzzle generator with the corresponding parameters.

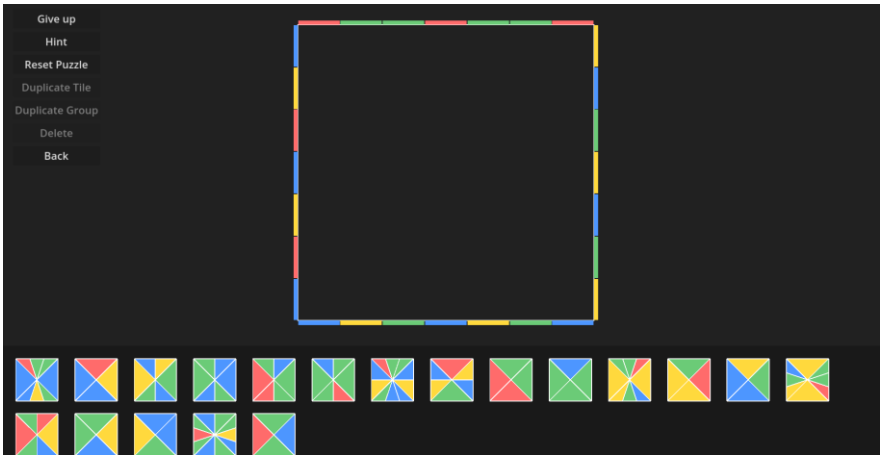

FIGURE 30: THE PLAY SCREEN

The play screen (Figure 30) is the primary game view. The board occupies the centre of the screen, framed by coloured edge segments that define the puzzle's boundary constraints. Below the board, a tray displays all available tiles arranged in a grid, allowing the player to survey the full tile set before beginning.

A minimal toolbar on the left provides access to the puzzle controls. The Delete, Duplicate and Duplicate Group buttons are greyed out unless a tile is selected.

Upon pressing the Give Up button, the solution is loaded and a level failed screen is displayed (Figure 31). A 'View Puzzle' button dismisses the overlay, leaving the completed board visible, which this allows the player to study the solution and understand how the tiles fit together, which is useful for learning. An equivalent win screen is displayed upon successful puzzle completion, accompanied by a confetti particle effect to reward the player and provide a satisfying sense of completion.

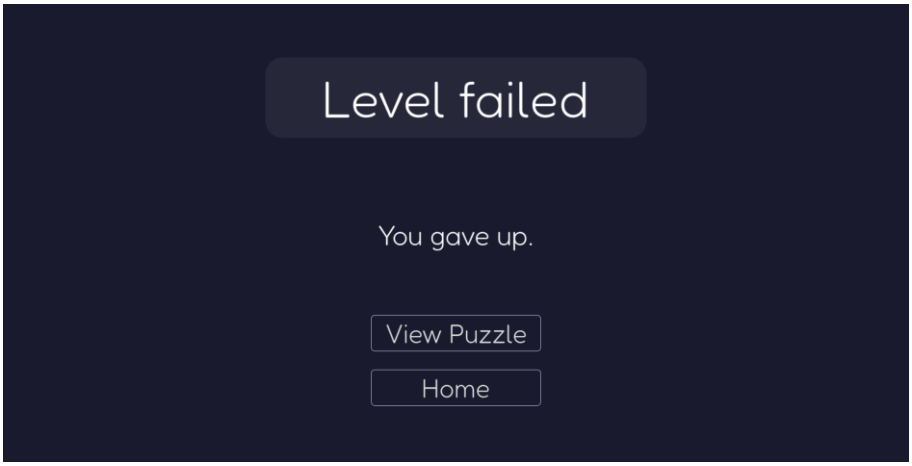

FIGURE 31: LEVEL FAILED SCREEN

FIGURE 32: WIN SCREEN

6.7 Sound design

Some sound effects were used to subtly enhance the gameplay. A short 'plop' sound effect was used to indicate when tiles successfully snap together, reinforcing the action with immediate audio feedback. The same sound was reused with a modified pitch for tile disconnections, creating a related but distinct cue while maintaining consistency in the audio design.

A music track described as 'chill' loops in the background of the game. The track was chosen for its calm and unobtrusive tone, helping to create a relaxed atmosphere that supports concentration during puzzle solving. It was sourced from Pixabay [15], which has a free to use license for its tracks.

6.8 Challenges and Design Changes

Throughout development, several challenges were encountered that required changes to the original design. In many cases, these changes improved the usability and robustness of the system, even if they deviated from the initial plan.

6.8.1 Tuning puzzle difficulty

One of the main challenges was tuning the difficulty of generated puzzles. While the generator guarantees solvability, it does not guarantee that puzzles are interesting or appropriately challenging. Early versions of the generator often produced puzzles that were either trivial, due to highly distinct edge patterns, or overly difficult, due to excessive ambiguity.

This made it necessary to introduce additional parameters, such as motif period and BSP depth, to better control how colours are distributed and how tiles are partitioned. Difficulty presets were then defined empirically by generating multiple puzzles and evaluating them manually.

Despite these improvements, difficulty balancing remains an imperfect process. The results from user testing suggest that the difference between some difficulty levels is not always clear, indicating that a more systematic approach may be needed.

6.8.2 Tile Partitioning Strategy

The initial implementation of the puzzle generator used a greedy rectangle-packing approach to divide the grid into tiles. While simple to implement, this method produced highly uneven layouts, with large tiles concentrated in one area and smaller, fragmented tiles elsewhere. This reduced the effectiveness of the scaling mechanic, as there was less variation in tile shapes and sizes.

To address this, the partitioning algorithm was replaced with a Binary Space Partitioning (BSP) approach. This allowed the grid to be subdivided more evenly, producing a wider variety of tile sizes and more balanced layouts overall.

This change significantly improved the quality of generated puzzles and made the scaling mechanic more meaningful in gameplay.

6.8.3 User Interaction and Controls

Some interaction issues became apparent during development and testing. For example, resizing tiles using multiple handles could occasionally lead to unintended behaviour, such as triggering multiple interactions at once. Additionally, some users found it unclear how to separate tiles from groups or use duplication features.

To address this, interaction states were more clearly defined, and input handling was refined to reduce conflicts between actions. Visual and audio feedback were also added to make interactions more explicit.

The player was initially only given the option to duplicate the tile they had selected, but the option to duplicate the whole tile group was added to allow for more efficient completion of some more creative puzzles.

6.8.4 Scope

Due to time constraints, several planned features were not implemented. These include a general puzzle solver, infinite tiling visualisation, camera controls, and a timer.

In prioritising development effort, focus was placed on ensuring that the core mechanics-- tile manipulation, snapping, grouping, and puzzle generation-- were fully functional and robust. This decision resulted in a more complete and stable core system, at the cost of some advanced features.

These unimplemented features are discussed further in section 8 as areas for future work.

7 Evaluation and Testing

7.1 Computational Complexity

7.1.1 Classifying the complexity

The computational complexity of the puzzle can be analysed by framing it as a decision problem: given a set of tiles with coloured edge sequences, does a valid tiling exist? This section examines where this problem sits within the complexity hierarchy, and how the scaling mechanic affects that classification.

We classify a problem as NP-complete if it is in NP, which means a proposed solution can be verified in polynomial time, and is NP-hard, meaning every problem in NP can be reduced to it. Standard edge-matching puzzles satisfy both conditions. Verifying a solution is straightforward: for each pair of adjacent tiles, check that their touching edges have matching colours. This requires checking at most O(n) edge pairs for n tiles, which is polynomial. Finding a solution, however, requires searching an exponentially large space of possible arrangements, and has been proven NP-complete [16].

Standard edge-matching puzzles can be seen as a special case of this game in which all tiles have identical size and use-count of 1. Any instance of a standard edge-matching puzzle can therefore be expressed as an instance of this game with no scaling required. Since the standard problem is NP-complete, this game is at least NP-complete by polynomial-time reduction.

The scaling mechanic introduces additional complexity beyond the standard case. Two sub-cases can be identified:

Constrained scaling. If use-counts are finite and scale factors are restricted to a discrete set of rational values (such as x1, x2), the search space remains finite, as there are a bounded number of possible placements and scales for each tile. The problem remains NP-complete despite the search space growing significantly compared to the unscaled case. While still in the same complexity class, puzzles are much harder in practice.

Unconstrained scaling. If use-counts are infinite and scaling is continuous, as is the case in this game, the search space becomes infinite, as there are uncountably many possible scale ratios for each tile placement. This is analogous to the Wang Tile tiling problem on the infinite plane, which Berger proved to be undecidable [5], meaning that algorithm can determine for all possible tile sets whether a valid tiling exists. While the scaling mechanic in this game similarly introduces an unbounded and continuous search space, this analogy alone is not sufficient to establish undecidability. Therefore, although the problem appears to be at least as hard as known undecidable tiling problems, a formal classification would require a reduction from an established undecidable problem. Establishing such a reduction is beyond the scope of this project and is left as future work.

7.1.2 Solver

The complexity analysis explains why a general solver was not implemented. In the unconstrained case the problem is significantly more difficult than standard puzzles and possibly undecidable. Even in the constrained case, NP-completeness means no polynomial-time algorithm is known, and the additional branching from scaling makes practical solving significantly harder than for standard edge-matching puzzles.

Given more time, a constrained solver could more feasibly be implemented. An effective algorithm would involve backtracking with constraint propagation: placing tiles one at a time, pruning branches where a partial placement creates an unsatisfiable constraint, and

backtracking when no valid placement exists. This is the approach used in SAT-based solvers for standard edge-matching puzzles [10].

A hint system was implemented as a practical alternative, reading from the generator's known solution to place one tile at a time without requiring a general solver.

7.2 System Testing

To evaluate the success of the project, each functional requirement defined in Section 4.1 will be tested against the implemented system to determine whether it has been met.

TABLE 5: EVALUATION OF FUNCTIONAL REQUIREMENTS

Requirement	Test	Result
Drag and drop tiles	Drag tiles to different positions on the board	Pass
	Try to drag a tile out of the screen boundary	Pass- tile clamped to screen
Tiles snap together	Place two tiles with matching colour and length within snap threshold	Pass- snapped together
	Try to snap tiles with mismatched colours or segment lengths	Pass – snap rejected
Tile groups	Snap three tiles together and drag the group around	Pass
	Shift click to remove a tile from a group	Pass
Tile scaling	Resize a tile by dragging from its corner	Pass
	Resize a tile by dragging from its edge	Pass
	Try to resize a tile as small as possible	Pass- size clamped
Edge matching validation	Snap a tile adjacent to a third tile that would create a colour conflict	Pass- snap rejected
Completed puzzle detection	Construct a valid solution matching all board edge segments	Pass- win screen appears
Reset puzzle	Place several tiles then press reset	Pass- tiles return to tray
Tile deletion	Select and delete a tile	Pass
Default puzzle	Launch the game and verify there are preset puzzles ready to load	Pass
Multiple solutions support	Complete the same puzzle in two different valid arrangements	Pass
Zoom & pan camera		Not implemented
Adjustable sequence length		Pass- incorporated into the difficulty presets
Puzzle difficulty presets	Generate a puzzle at each difficulty level	Pass
Timer		Not implemented
Puzzle generator	Verify generated puzzles are solvable	Pass
Puzzle solver		Not implemented
Infinite tiling visualisation		Not implemented

7.2.1 Requirements not implemented

Zoom and pan camera – the ability to zoom in and out and pan around the board was planned to improve usability on larger puzzles. This was deprioritised in favour of completing the core tile mechanics and puzzle generator. It was decided that the camera shall remain static instead.

Timer – an optional timer was planned as a way for players to track their solve time and add a competitive element to the game. This was considered a low-priority feature and was not implemented due to time constraints. It remains a straightforward addition for future development.

Puzzle solver – a general solver was not implemented. As discussed above, the unconstrained case of the puzzle is at least NP-complete, making a general solver theoretically infeasible. A hint system was implemented as a practical alternative.

Infinite tiling visualisation – visualising the convergence of a recursively scaling infinite tiling was identified in the introduction as a theoretically interesting extension. This was not implemented due to its complexity and the time required to develop a robust approach. It remains an open and interesting direction for future work.

7.3 User Testing

Time constraints and resulted in the user testing not being as extensive as would have been ideal for a thorough evaluation. The primary aim of user testing was to test the system against the non-functional requirements outlined in section 4.2, particularly the usability, error-handling and accessibility of the game. In the end, 4 participants were interviewed, which is too small a sample size to assert results conclusively, but provides an acceptable qualitative basis for evaluation of a project of this scope.

In section 5.7, focus was placed on creating an accessible experience particularly for colourblind users. Ideal testing would involve garnering feedback from people who have different types of colour-blindness, noting the effectiveness of the various colour palettes and whether they impact gameplay at all. Since no colourblind participants were recruited, this aspect is deferred to future work.

Interviews were conducted with a small number of anonymous, voluntary participants. They were debriefed with the scripts in Appendix A – Ethical Compliance Form. The questionnaire in Appendix B – Interview was used to collect the participants' opinions and some observations were made by the interviewer.

7.3.1 Results

The results of the user interviews are summarised below.

TABLE 6: USER INTERVIEW RESULTS

Participant 1:	
Aspect	**Opinion**
Difficulty	Pre-built puzzle: fairly easy, good as an introduction to the game
	Generated puzzles: Easy and medium were of similar difficulty, hard was too hard
Liked Features	The visual indication of segments that are valid to snap to is helpful
Issues or missing features	User sometimes resized multiple tiles with one click without meaning to
UI navigation	Menus were very simple and easy to navigate
Overall enjoyment	Enjoyed the game, but probably would need more variation in puzzles to play it more
Participant 2:	
Aspect	**Opinion**

Difficulty	Pre-built puzzle: took some time to grasp the tile re-using/duplication but it was fun
	Generated puzzles: medium difficulty was nice, not too hard or easy
Liked Features	The music and sound effects were nice ambience
Issues or missing features	A help section or tutorial would be nice
UI navigation	Very simple and easy to understand
Overall enjoyment	The user stated they would play the game casually for fun as a short, time-filling activity
Participant 3:	
Aspect	**Opinion**
Difficulty	Pre-built puzzle: easy
	Generated puzzles: Easy was too easy, medium felt like a good step up
Liked Features	The colour palette was nice, menus were pretty
Issues or missing features	A way to save and share puzzles, or a multiplayer mode, would be fun
UI navigation	No problems, liked the background design
Overall enjoyment	The user enjoyed the game
Participant 4:	
Aspect	**Opinion**
Difficulty	Pre-built puzzle: easy to understand mechanics
	Generated puzzles: hard difficulty felt overwhelming and difficult to approach
Liked Features	The snapping mechanic felt satisfying when tiles connected correctly. The puzzles felt tedious at times.
Issues or missing features	The valid snaps could be more visually obvious- the blue highlighting gets lost in all the colour
UI navigation	Generally clear, but some controls were not immediately obvious
Overall enjoyment	Found the concept interesting but game is not expansive enough

Overall, the user testing suggests that the core mechanics of the game are intuitive and engaging, with most participants able to understand and interact with the system without significant difficulty. The drag-and-drop interaction and snapping feedback were consistently identified as effective features, indicating that the core usability requirement has largely been met.

However, several recurring issues were identified. One of the most prominent was difficulty balancing. Multiple participants noted that the distinction between easy and medium puzzles was not always clear, while hard puzzles were often perceived as disproportionately difficult. This suggests that the current parameter tuning for the puzzle generator does not yet produce a consistent progression in challenge.

Another common theme was the lack of onboarding or guidance. Participants who were unfamiliar with mechanics such as tile duplication or resizing required some time to understand how these features worked. This indicates that the system would benefit from the inclusion of a tutorial or help system to improve initial usability.

Visual clarity was also highlighted as an area for improvement. While snapping feedback was generally well received, one participant noted that the highlighting could be difficult to distinguish. This suggests that the visual design of feedback elements may need to be adjusted to ensure they remain clear under all conditions.

In terms of engagement, participants generally reported that the game was enjoyable, particularly as a short, casual experience. However, some users indicated that additional content or features, such as more puzzles or the ability to share them, would be needed to maintain long-term interest.

It is important to note that these findings are based on a small sample size and are therefore not conclusive. However, they provide useful qualitative insights into the strengths and weaknesses of the system and highlight clear areas for future improvement

7.3.2 Feedback during implementation

Special thanks to Participant 1, who provided some feedback during the process of implementation via a short interview which can be found in.

7.4 Non-functional Requirements

Based on the results of user and system testing, the non-functional requirements were also evaluated based on whether they were fulfilled, partially fulfilled, or not fulfilled.

Requirement	Fulfilled?
Usability	Partially fulfilled – Users generally found the interface intuitive and easy to navigate, with positive feedback on layout and clarity. However, minor issues such as initial confusion with certain controls and lack of guidance (e.g., tutorial or hints) indicate room for improvement.
Performance	Fulfilled – The system responded quickly during testing, with no noticeable delays when loading, generating, or interacting with puzzles.
Reliability	Fulfilled – No crashes or major bugs were reported during user or system testing. Core functionality operated consistently across all test scenarios.
Accessibility	Partially fulfilled – Accessibility features such as colourblind-friendly design were considered during development, but could not be fully validated due to lack of testing with colourblind participants.
Maintainability	Fulfilled – The system was developed with a clear structure and modular design, making it easier to update or extend.
Error-handling	Partially fulfilled – Basic error handling was present.

8 Conclusion

8.1 Evaluation

Overall, the project was successful in achieving its primary objectives. The final system allows players to drag, scale, and connect tiles within a freeform play area, with snapping logic ensuring that only valid connections are made. The puzzle generator was also successfully implemented, producing puzzles that are guaranteed to be solvable by constructing them from an underlying grid.

Most of the extension objectives were partially achieved. The puzzle generator and difficulty presets were implemented, along with a small set of pre-built puzzles. However, more advanced features such as a general puzzle solver and infinite tiling visualisation were not completed within the available time.

The solver was addressed in section 7.1, where its limitations and the reasons for not fully integrating it into the final system are outlined. Infinite tiling was not implemented due to time constraints and the added complexity it would introduce to both the generation and rendering systems.

User testing, although limited, provided useful feedback. Participants generally found the game easy to understand and the interface straightforward to use. However, some issues were identified, such as unclear controls and difficulty levels not always feeling well balanced. These results suggest that while the core mechanics are strong, there is still room to improve usability and onboarding.

From a theoretical perspective, the project only partially achieved its aim of exploring the complexity of the problem. The analysis shows that the puzzle generalises known NP-complete problems, and that allowing continuous scaling introduces a much larger (potentially unbounded) search space.

8.2 Future work

There are several clear areas for future development. One of the most valuable additions would be a constrained puzzle solver, using techniques such as backtracking and constraint propagation. This goes hand in hand with extension task of properly classifying the complexity of the puzzles.

Another interesting direction would be to explore ways of visualising infinite tiling patterns by approximating recursive scaling. This would build on the theoretical ideas discussed earlier and add a unique feature to the game.

The puzzle generator could also be improved further, particularly in how it controls difficulty and ambiguity. Expanding the set of pre-built puzzles and allowing players to save or share puzzles would also increase replayability.

Finally, more extensive user testing would be beneficial, especially involving players with accessibility needs. This would allow the usability and accessibility features to be properly evaluated and refined.

8.3 Final remarks

In summary, the project demonstrates that extending edge-matching puzzles with scaling and multi-segment edges creates a system that is both technically interesting and engaging

to play. The core objectives were achieved, and the final system provides a strong foundation for further development, both as a game and as a topic for further study.

9 References

[1] 'BCS Code of Conduct for members - Ethics for IT professionals | BCS'. Accessed: Apr. 14, 2026. [Online]. Available: https://www.bcs.org/membership-and-registrations/become-a-member/bcs-code-of-conduct/

[2] P. Harel, O. I. Shahar, and O. Ben-Shahar, 'Pictorial and Apictorial Polygonal Jigsaw Puzzles from Arbitrary Number of Crossing Cuts', *Int. J. Comput. Vis.*, vol. 132, no. 9, pp. 3428–3462, Sep. 2024, doi: 10.1007/s11263-024-02033-7.

[3] S. Markaki and C. Panagiotakis, 'Jigsaw puzzle solving techniques and applications: a survey', *Vis. Comput.*, vol. 39, no. 10, pp. 4405–4421, Oct. 2023, doi: 10.1007/s00371-022-02598-9.

[4] H. Wang, 'Proving theorems by pattern recognition - II', *Bell Syst. Tech. J.*, vol. 40, no. 1, pp. 1–41, 1961, doi: 10.1002/j.1538-7305.1961.tb03975.x.

[5] R. Berger, *The undecidability of the domino problem*. in Memoirs of the American Mathematical Society ; number 66. Providence: American Mathematical Society, 1966.

[6] Scott Ferguson, *TetraVex*. (1991). Accessed: Apr. 14, 2026. [Online]. Available: http://archive.org/details/win3_TetraVex

[7] 'Tetravex - online game'. Accessed: Apr. 14, 2026. [Online]. Available: https://smart-games.org/en/tetravex/game/

[8] 'Jigsaw Explorer – Online Jigsaw Puzzles'. Accessed: Apr. 14, 2026. [Online]. Available: https://www.jigsawexplorer.com/

[9] I. Sommerville, *Software engineering*, Tenth edition. Boston: Pearson, 2016.

[10] C. Ansótegui, R. Béjar, C. Fernández, and C. Mateu, 'On the hardness of solving edge matching puzzles as SAT or CSP problems', *Constraints*, vol. 18, no. 1, pp. 7–37, Jan. 2013, doi: 10.1007/s10601-012-9128-9.

[11] 'Definition of disability under the Equality Act 2010', GOV.UK. Accessed: Apr. 17, 2026. [Online]. Available: https://www.gov.uk/definition-of-disability-under-equality-act-2010

[12] 'Disability'. Accessed: Apr. 17, 2026. [Online]. Available: https://www.who.int/news-room/fact-sheets/detail/disability-and-health

[13] J. Molina-López and N. Medina-Medina, 'Design proto-patterns to improve the interaction in video games of people with color blindness', presented at the XX International Conference on Human Computer Interaction (Interacción 2019), ACM. doi: 10.1145/3335595.3335612.

[14] D. Nichols, 'Coloring for Colorblindness'. Accessed: Apr. 19, 2026. [Online]. Available: http://www.davidmathlogic.com/colorblind/

[15] PaulYudin, 'Chill - Chill Music | Royalty-free Music'. Accessed: Apr. 30, 2026. [Online]. Available: https://pixabay.com/music/beats-chill-chill-music-513017/

[16] M. R. Garey, *Computers and intractability: a guide to the theory of NP-completeness*. San Francisco: Freeman, 1979.