02122 Software Technology Project

Alexander Collignon s194605
Carl A. Jakcson s194585
Joel A. V. Madsen s194580

March 2021

Danmarks Tekniske Universitet

Contents

1 Abstract

Throughout this paper, we will describe the development process of a 2D puzzle application, as well as a solving algorithm for such puzzles. The paper will describe the design choices and implementation process of the program, and goes in depth with the components of the final application. This paper includes analysis of the problems we met during the implementation process. It also elaborates on the design and implementation of the final program. Furthermore it will touch on limitations of the current program, as well as potential fixes for these known limitations. The paper will also describe how we can create, distinguish, and fit pieces together correctly when solving the puzzle. The project is created using the programming language Java, and we use a library for Java called Processing to utilize both it's visual competences and other functionalities. The paper will furthermore go in depth with the graphical interface and interaction with the user. Lastly it will look back on the development process, and go in depth with some solutions that ultimately did not work out. Finally we conclude, that during the process we managed to accomplish most of the goals we set out to do at the beginning, but that there is still room for improvement and minor details.

2 Introduction

Written together

The purpose of this paper is to account for and describe our work on the puzzle problem. We will discuss our design choices and explain how the puzzle problem has been tackled. We will also touch on design flaws, complications and problems that we underwent during the process. The paper will explain all the aspects of our working program, and give the reader great insight in the final project.

2.1 The problem

Written together

For this paper, the task was to create a puzzle game. Our puzzle game must include certain elements, and needs to be playable by a user. The user should be able to choose between generating a random puzzle or reading in a puzzle from a file. The puzzle program also needs to be able to identify identical pieces, and it should also be able to solve a given puzzle by itself.

2.2 The goals

Written together

We have several goals which we would like to accomplish for this project, but in order not to get ahead of ourselves, we created a MoSCoW prioritization model, to get a better grasp of our ideas and ambitions. The MoSCoW model is listed below.

- MUST
  - Interactive puzzle pieces
  - Solving algorithm
  - Puzzle generator
  - Individuality identification algorithm
  - Read JSON files given by instructor
- SHOULD
  - Puzzle solver algorithm for big puzzles
  - Puzzle snapping
- COULD
  - Image mapping onto puzzle
  - Piece merging

- WONT
  - Animations
  - Sound effects
  - Three dimensional support

3 Analysis of the problem

Written by Joel

To analyze the puzzle problem, we used a top-down approach where we looked at the puzzle problem as a whole, and tried to divide the problem into smaller sub-problems. We repeated this process until we had well defined problems that were easy to understand.

After dividing our problem into sub-problems we were left with some main problems that had to be solved, these main problems include:

- Creating an interactive piece
- Generating a puzzle
- Checking if two pieces are identical
- Solving a puzzle only based on the pieces
- Checking if the puzzle is solved

In the next part of this section, we will further analyze the problems stated above, repeat the top-down approach on each main problem, and come up with possible solutions.

3.1 Analysis of interactive pieces

Written by Joel

A piece is the main component in a puzzle, without pieces a puzzle cannot exist. In the broad understanding pieces can be defined in various ways, therefore we start by defining our understanding of a piece datatype:

Piece definition:
A piece is a polygon with a defined origin[1], containing an arbitrary amount of vertices, where the vertices are defined from the origin of the piece. A piece with position on the outer circumference of the puzzle, must not have more than two vertices on the edges facing out of the puzzle.

Observing the piece problem, we can now further divide it into smaller sub-problems, that are necessary for creating an interactive piece. The sub-problems are listed below, from high to low priority.

[1] An origin can be an arbitrary point, it acts as a perspective point for the vertices.

- Creating the polygon shape
- Checking if the shape contains a point
- Moving and rotating the shape
- Snapping two pieces together
- Mapping an image onto the shape

We will now analyze each sub-problem further and either come with a possible solution, or divide it into further sub-problems, and repeating the process.

3.1.1 Creating the polygon shape

Written by Joel

By the above definition, we want to be able to create a piece by supplying an origin point and a list of vertices with an arbitrary length, containing (x, y) coordinate pairs, like shown below:

```
Center:    (x, y)
Vertices:  {(-sideLength, -sideLength),
            ( sideLength, -sideLength),
            ( sideLength,  sideLength),
            (-sideLength,  sideLength)}
```

The above would represent a square at position (x, y) and side length sideLength. To accomplish this we can make use of functionalities contained within in the processing library.

3.1.2 Interacting with the piece & checking for a point

Written by Joel

We want to be able to interact with the polygon piece, meaning moving and rotating the piece. To move the piece we can update the origin of the piece, since the vertices are defined from the origin point. To rotate the piece around the origin, we can use simple sine and cosine math, to calculate the position at an angle. To check if a point is contained within the piece, we can use functionalities contained within the processing library.

3.1.3 Mapping an image onto the piece

Written by Joel

We would also like to be able to map an image onto a piece, which we can divide further into three sub-problems.

1. Resizing and cropping the image.
2. Splitting the image into sprites.

3. Mapping each sprite onto the corresponding piece.

To accomplish these sub problems we will use methods and functionalities contained within the processing library, namely image manipulation functionalities.

3.1.4 Snapping two pieces

Written by Joel

To understand the meaning of the sub-problem piece snapping, we start by defining the meaning of it:

Piece snapping is an auto-completion, of the positional property, of a piece, when the piece is within a certain distance of one of it's corresponding neighbour pieces[2].

We have two main ideas of how to solve this sub-problem.

1) Extend four points from the center of a piece in four directions (top, bottom, right and left), and check if one of the four points is contained within one of the corresponding neighbor piece. If so, the piece must be close to the neighbor piece.

2) Calculate the distance between the centers of a piece and it's neighbor pieces, check whether the distance to one of the neighbor pieces is less than some threshold, if so the piece must close to its neighbor.

Both implementations are described in section 5.1.4, this is because the implementation of the first method gave rise to some issues, which then resulted in the development of the second method, which took these problems into account.

3.2 Analysis of generating pieces

Written by Carl

When generating the puzzle, we generally follow two ideas.

- Generating a puzzle, i.e. generating a specific number of pieces which fit together.
- Reading a JSON file with given pieces.

When generating our own puzzle pieces, we would like to follow a general column and row format, meaning that the pieces are placed in a grid pattern, and every piece has no more than four neighbours.

[2] The definition of neighbour is: P1 is neighbour to P2 if and only if they share an entire side.

3.3 Analysis of identical pieces

Written by Carl

Our program has to determine whether or not two pieces are identical. We define identical pieces as two pieces that are able to completely cover one another, this means if you lay either of the pieces on top of each other they are able to cover the piece underneath.

The formal definition of identical pieces is described below.
Definition of two identical Pieces: Two Pieces, P1 and P2, are identical, when P1 can make an exact overlap of P2, and P2 can make an exact overlap of P1.

To check if pieces are identical, we have divided the identification check into three steps. We chose this method to save computational power, by not checking pieces that are obviously not identical.

The three steps are:

- Identical pieces have equal circumference.
- Identical pieces have the same amount of points.
- Identical pieces have the same angles and side lengths.

While the two first steps are generally easy to implement, we knew that calculating the angles of all the corners would prove to be easier said than done. This is because we are only interested in the angle pointing towards the center of every piece. Further exploration of the problem is described in 5.4.

3.4 Analysis of computing angles

Written by Alexander and Carl

It is difficult to describe the analysis computing the angles for our pieces, without describing the implementation process. This is because the angle computation method was invented through trial and error process during the implementation phase. Therefore this section is somewhat intertwined with the implementation section of the angles in 5.

3.5 Analysis of the solving algorithm

Written by Alexander

To solve the puzzle, we need to break the solving problem into more manageable sub-problems.

- Grouping the pieces into three distinct groups, corner groups, side groups and middle groups.
- Matching the groups with the possible neighbours of their own and other groups.

- Producing an array of adjacent pieces, so we know which pieces are in fact neighbours.
- Producing an array containing an instance for every match, so we know on which side the pieces are matching.
- Rotating the matching pieces to match the rotation of their corresponding neighbour.
- Moving the matching pieces accordingly to align with their corresponding neighbour.

When rotating and moving the pieces, we want to make sure that all the rotations and placements are correct. Since we rarely have a puzzle where a piece is neighbour to all other pieces, we have to come up with a standard rotation, and final position according to one piece.
To make all other pieces have the same rotation and placement, we need to know their relation to the first piece. The relation should decide how the piece should be rotated according to the initial piece. To find this relation we came up with two ideas.

- The first idea was to find a path that goes through each piece exactly once, starting from the initial piece.
- The second idea is making a tree structure, where the initial piece is the root node and the tree expands to its children, which are the matching pieces, this will produce several short paths from the initial node.

To chose which one to use, we made a rough time complexity analysis. For the first algorithm, we make the assumption that each piece has 4 neighbours, since we want to have a path that covers all pieces, the variable will be the amount of pieces, we call this variable n. In the worst case, we would have to check all possible paths. The big O notation would therefore be $O(4^{n-1})$ ($n-1$ because the initial piece isn't included). This is exponential time, which isn't optimal. In the second algorithm we can find as many paths as we want, but all of them originate in the same piece. For each each piece we have to consider up to 4 different paths, since there is a maximum of 4 neighbours. The big O notation of this is $O(n^4)$. This is potential time, which is faster than exponential time. We will attempt to implement the solving algorithm using the latter method. Limitations of the solving algorithm will be described fully in section 6.

3.6 Analysis of the puzzle completion check

Written by Joel

An undeniable important feature for any "game"-like program, is to check if the user has completed the task handed to them. For this program it is to check whether the user has completed puzzle. To get a better understanding and avoid possible confusion, about what is meant by completing the puzzle, we start by

describing the definition of a complete puzzle in our program.

A complete puzzle is a puzzle where every piece is positioned such that they are share sides with all their possible neighbors.

To accomplish this we can reuse the solution of our method of snapping two pieces together. We know that two pieces are directly besides each other if they have snapped together, therefor we can check if all pieces maintain the logic of the snapping method, if so, the puzzle must be completed.

3.7 Analysis of UI-components

Written by Joel

An important feature of every program with a visual component, is the interaction between the user and the program. To create a good user interaction experience, we have to develop some custom UI components.

After analysing the other sub-problems of the puzzle problem, we listed a couple of UI components that would be necessary to enhance the user experience. The list includes:

- Menubar - which can contain the other elements
- Text input - lets the user input text to the program
- Switch - lets the user switch between generator and reader
- Slider - lets the user chose any value between a minimum and a maximum
- Button - lets the user click to execute an event

The functionality will be further explained in the corresponding implementation section.

4 Design

Written together

4.1 Model view controller

The program itself is designed as a model-view-controller pattern 1.

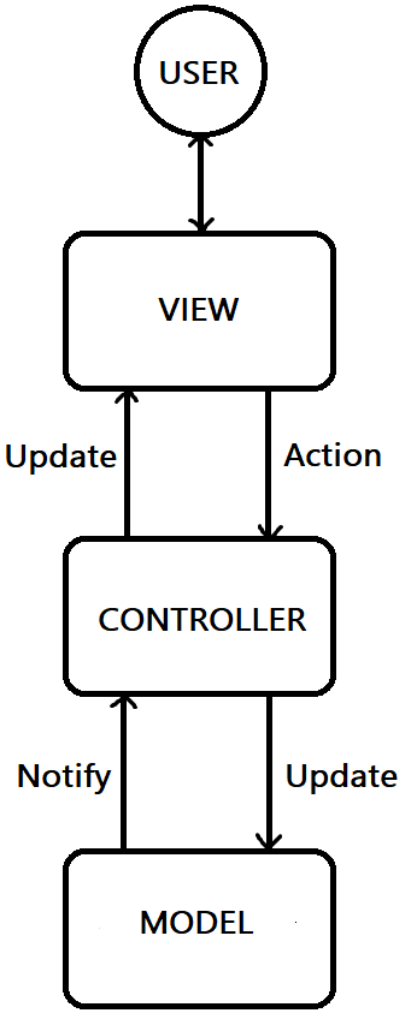


Figure 1: Model-view-controller architecture

A more detailed illustration of this design, based on the implementation phase, can be found in the appendix, where it is clear how the user interacts with the program 26.

Since we are designing a puzzle program, we use a view to control the display, which then interacts with the controller and model which contains all the logic.
We have three major components, a generator, a solver and a graphical user interface. We include the generator and solver in the model component of the design. The graphical user interface is included in the view.

4.2 Technologies and platforms used

We used "Java" as our programming language for this project.
We used the integrated development environment "Eclipse" for this project.
We used "GitHub" for version control.
We used "Draw.io" to construct our class diagram.
We used "Processing" for the graphical user interface, and for the displaying of puzzles.
In several cases we used "Geogebra" to analyze the given mathematical problems concerning angles of the pieces, as well as their positions in the plane.
We used "Trello" to structure the development of components for the program.
We used "Paint" for drawing illustrations used in the report.

5 Implementation

5.1 Piece

Written by Joel

In this section, we will describe how a piece is created, made interactable and made visually appealing.

The piece is created as a separate Java class, containing the below functionalities.

5.1.1 Piece shape

Written by Joel

To generate the polygon shape we use an object called PShape, which is a part of the processing library. The PShape object allows us to create a polygon shape by defining its vertices, as well as allowing other functionalities, such as drawing the shape onto our view.

As per our definition in section 3.7, we can pass a list of an arbitrary amount of vertices and an origin of the polygon, to the PShape object, which will then contain this exact polygon. By adding the origin to each vertex in the list of vertices, the generated polygon will be created with the origin as perspective point.

Updating the piece creation seen in section 3.7, we now represent the (x,y) pairs as a Point object from Java AWT, and supply an origin and vertices list, to create the piece, as seen below.

```
int sideLength = 200/2;
Point2D.Float   center   =  new Point2D.Float(200, 200);
Point2D.Float[] vertices = {new Point2D.Float(-sideLength, -sideLength),
                            new Point2D.Float( sideLength, -sideLength),
                            new Point2D.Float( sideLength,  sideLength),
```

```
                                    new Point2D.Float(-sideLength,  sideLength)};

    Piece piece = new Piece(center, vertices);
```

The above will create piece with a square shape at position (200, 200) with side lengths 200.

5.1.2 Contains mouse

Written by Joel

One very important functionality, is to be able to check whether a point is within the shape. This functionality is widely used throughout the puzzle, and the main usage is to check if the mouse is on a piece, to control whether it can be selected and moved.

Our first intuition was to use a contains method, which followed the PShape object, but we quickly discovered that the contains method was only implemented for the so called Path shapes, which we were not using, as the Path shape would cause other problems in our program, see section 5.1.6.1 where we further explain the problem.

After doing some research we found a paper by W. Randolph Franklin on point inclusion in polygons, which explained how to create a contains method, the paper can be found here. From the paper we derived our own contains method, which can be seen below.

```
public boolean contains(float x, float y) {
    Point2D coords = new Point2D.Float(x,y);
    int i;
    int j;
    boolean result = false;
    for (i = 0, j = vertexCount - 1; i < vertexCount; j = i++) {
        if ((shape.getVertex(i).y > coords.getY()) !=
            (shape.getVertex(j).y > coords.getY()) &&
            (coords.getX() < (shape.getVertex(j).x - shape.getVertex(i).x) *
            (coords.getY() - shape.getVertex(i).y) /
            (shape.getVertex(j).y-shape.getVertex(i).y) +
            shape.getVertex(i).x)) {

         result = !result;
        }
    }
    return result;
}
```

5.1.3 Moving and rotating a piece

Written by Joel

Since every vertex within a piece is defined from the origin of the piece, if we update the origin of the piece, the vertices will update accordingly. To move the piece we first need to check if the mouse is within any piece, to accomplish this, we use the contains method from section 5.1.2. We now need to check three extra conditions, namely whether the mouse is clicked, released or dragged. To do this we utilize methods from the Processing library, mousePressed(), mouseReleased() and mouseDragged(), which check for those exact user inputs.

In the below descriptions it is assumed that the mouse is within a piece.

Inside the mousePressed() method, we store the piece object, which will be addressed as *the selected piece* in the rest of this paper. Within the mouseReleased() function, we wipe the stored piece object, if there is any. Within the mouseDragged() function, we update the stored center position of the piece, to be relative to the position of the mouse on the screen.

To rotate the selected piece, we use yet another method from the processing library, called mouseWheel(MouseEvent e). The mouseWheel(MouseEvent e) function, will be able to return the integer, being 1 or -1, relative to the rotational direction of the mouse wheel. Using this integer we can call a custom function from the piece called rotatePiece(angle), which takes an angle in degrees. The function rotatePiece(angle) utilizes cosine and sine relations to calculate new points, from the given angle.

$$cos(\theta) * ((x_0 - x_1) - x_0) - sin(\theta) * ((y_0 + y_1) - y_0) + x_0$$

$$sin(\theta) * ((x_0 - x_1) - x_0) - cos(\theta) * ((y_0 + y_1) - y_0) + x_0$$

Where θ is the angle in radians, (x_0, y_0) is the origin of the piece, and (x_1, y_1) is the position of one vertex point.

Repeating these calculations for every vertex in the selected piece, the result will be the same shape, but rotated by θ around the origin.

5.1.4 Piece snapping

Written by Joel

In this section we will explain two methods of piece snapping, as well as possible problems with the methods, and end of by concluding which method is more reliable.

5.1.4.1 Piece snapping general

Written by Joel

Snapping happens between the selected piece and one of its neighbor pieces.

To achieve piece snapping we first have to find every neighbor piece for a given piece. To do this we construct a list containing all piece objects in the puzzle, which we can then use as an adjacency list. Utilizing the adjacency list, we can find all neighbors for a given piece, by calculating their corresponding indexes in the list, as shown below.

```
topNeighbor     = pieceIndex - columnCount
bottomNeighbor  = pieceIndex + columnCount
rightNeighbor   = pieceIndex + 1
leftNeighbor    = pieceIndex - 1
```

After finding the corresponding neighbors for the given piece, we check if the pieces have the same rotation, by checking if $sin(\theta)$ and $cos(\theta)$ are equivalent for the two pieces, if this is true, then they must have the same rotation.

$$cos(\theta_{p1}) = cos(\theta_{p2})$$

$$sin(\theta_{p1}) = sin(\theta_{p2})$$

Given the two pieces have the same rotation, we can then check whether the selected piece is within snapping range[3] of a neighbor, and if so move the piece to the edge of the neighbor piece.

To move the selected piece to the correct position, we use cosine and sine relations.

$$snapX = x_{neighbor} - c * cos(\theta + a)$$

$$snapY = y_{neighbor} - c * sin(\theta + a)$$

Where x and y neighbor, is the position of the neighbors origin, c is a constant, which is either the base height[4] or width of the piece, depending on what side the selected piece should snap to. θ is the angle of the piece and a is another constant, which describes what offset angle, from θ, the selected piece should snap to, since θ is equivalent to 0π.

To check whether a neighbor piece is within snapping range of the selected piece, we have constructed two methods which will be described below.

5.1.4.2 Snapping range method 1

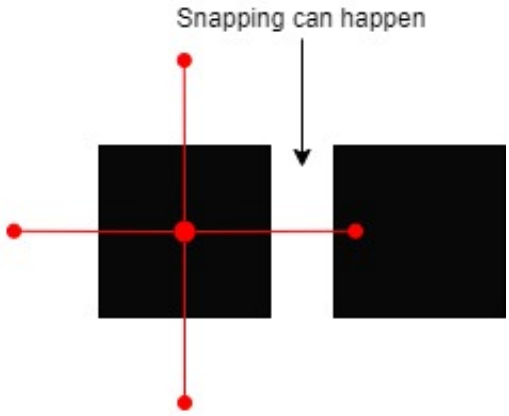
Written by Joel

For method 1 the snapping range is calculated by extending four points, from the center of the selected piece, and checking if either of these points are contained within a the corresponding neighbor piece, meaning left point should be contained within the left neighbor, and so on. If the point is contained within

[3] A set range, from where snapping will be able to happen.

[4] The base hight/width is the height/width of the pieces as squares, without any extra vertices.

the correct neighbor piece, we can then move the selected piece to the neighbor piece using the method described in section 5.1.4.1.



To check if a point is contained within a neighbor piece, we use the contains method described in section 5.1.2.

5.1.4.3 Problems with method 1

Written by Joel

Method 1 comes with one big issue, we are only checking one point for every direction, which needs to be contained within the correct neighbor piece. Essentially this means that if we want piece snapping to happen at a specific point, which is further away than half the width of the neighbor piece, we could run into an issue, where the point overshoots the neighbor piece, and piece snapping won't happen, even though it should.

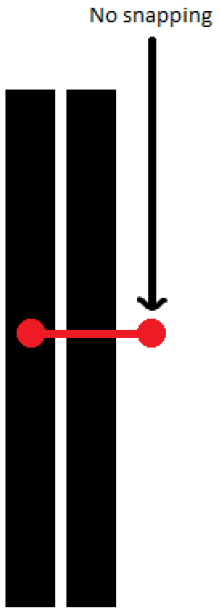


Figure 2: Showing an issue, where the checking point is outside the neighbor piece

Given the situation above, we would assume that piece snapping should happen, but by using the described snapping method, it is clearly not always the case.

5.1.4.4 Snapping range method 2

Written by Joel

This method arose as a result of the problem of method 1.

To eliminate the problem we faced in the first version of the piece snapping algorithm, we had to revisit our method of checking if two pieces were close enough to snap together. We realized that calculating the distance between the centers of the pieces, would result in a much more reliable solution to our problem. By using the distance between centers we could in theory define a range, where snapping could occur, which would eliminate the issue we had with method 1.

We use the distance formula,

$$distance = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

to calculate the distance between the center of the selected piece, and it's neighbor piece. It is now fairly simple to check if the distance is within a range of a minimum and maximum value, that we define as our snapping range.

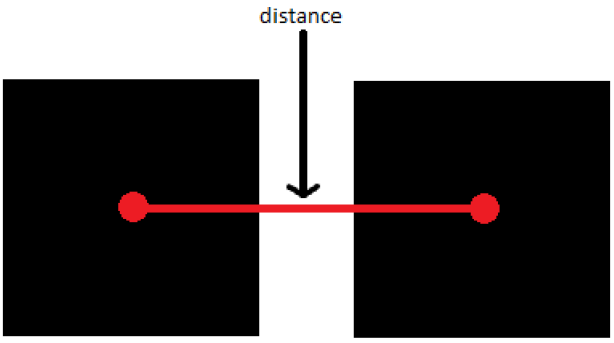


Figure 3: Showing the distance calculation

Since the distance between the center of the pieces is not affected by direction, pieces that should only be able to snap from one side, could now snap from all sides. To account for this problem we had to implement a check, that took the direction between the two pieces into account. To accomplish this, we calculated the difference between the coordinates for the two pieces. By checking if the x and y difference was positive or negative, we could determine how the pieces were positioned in relation to each other.

$$diff_x = (cos(\theta) * x_0) - (cos(\theta) * x_1)$$

$$diff_y = (sin(\theta) * y_0) - (sin(\theta) * y_1)$$

We use cos and sin to account for possible rotated pieces, where we still want to check the coordinate difference.

By taking the sum of the difference in the x and y coordinates, we can now determine if the selected piece is positioned in the correct direction in relation to the neighbor piece.

5.1.4.5 Conclusion on snapping range

Written by Joel

By using method 2, we account for the problems that arose by using method 1, since we now are checking for a range, rather than a single point, meaning overshooting is not possibility.

We conclude that method 2 is much more reliable, and it is therefore also the method we use in the program.

5.1.5 Mapping an image onto the pieces

Written by Joel

In this section we will describe the process of mapping an image onto the puzzle

pieces.

To represent an image, we use the PImage class, contained in the Processing library.

5.1.5.1 Resizing and cropping the image

Written by Joel

The first step in the process of mapping an image onto our pieces, is to crop the image. We do this to assure that the image is formed as a square, since our puzzle will have this form.

We find that for most images it is true that the center part, is the part containing the most meaningful content, and therefor we have chosen to crop away the sides, creating a square around the center of the image. To crop the image we use a function contained within the PImage class, namely the function get(firstX, fistY, secondX, secondY). If we have to crop the image, we know that the width != height of the image, this means that we either have the case where height $<$ width or width $<$ height.
For the first case the center of the image can be cropped as such:

$$get(height/2, 0, height, height)$$

For the second case it can be cropped as such:

$$get(0, width/2, width, width)$$

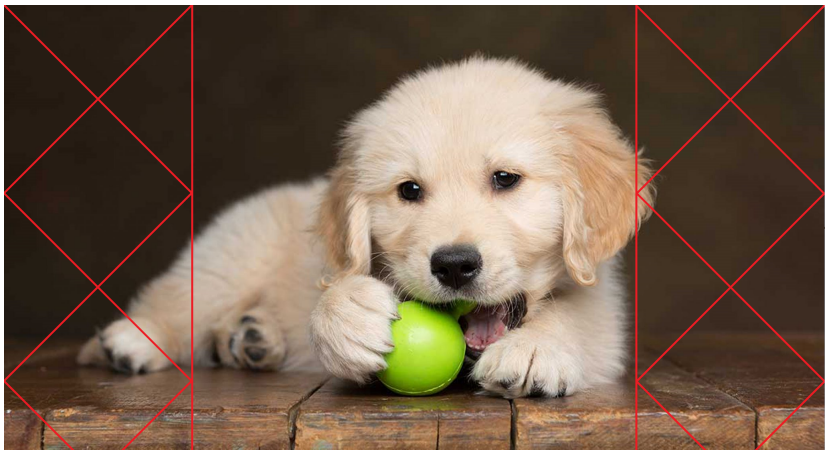


Figure 4: Image, where the red fields are cropped out

After having cropped the image into a square, we proceed to resize the image, scaling it to fit the puzzle size, to do this we use the predefined function resize(sizeX, sizeY), contained in the PImage class.

5.1.5.2 Splitting the image into sprites

Written by Joel

From the last step, we are left with a square image, with the exact size of our puzzle. Using some known information, about the size of the puzzle and how many pieces there are per row and column, we can now proceed to cut the image into smaller squares, one for each of the pieces in the puzzle.

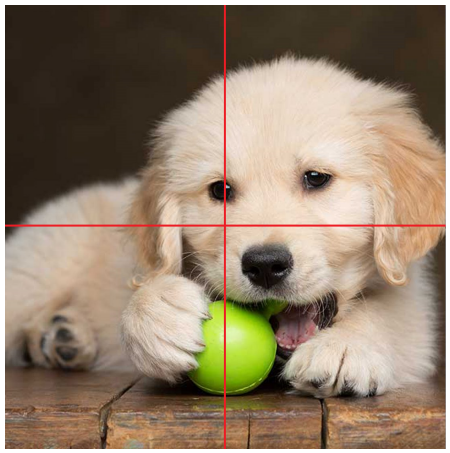


Figure 5: The image split into four, for a puzzle consisting of four pieces

To cut the image we once again make use of the function get(firstX, fistY, secondX, secondY). We are now left with n evenly sized images, where n is the number of pieces, and these images can now be passed to their corresponding piece.

5.1.5.3 Mapping the sprite onto the piece

Written by Joel

To map the images onto the pieces, we can use a functionality that is contained in the PShape class, which lets us set the texture of a PShape object when it is created. The only modification from our original PShape creation, described in section 5.1.1, is that we can now specify the relation of each of the shapes vertices, in correlation to the texture image. This results in a mapping of the image onto the piece, but since the images are squares, and the pieces have vertices greater than four, the image will get distorted by the extra vertices, which results in an abstract representation of the original image.

Figure 6: The image on the left is not distorted, the right image is distorted by mapping it onto a piece

5.1.6 Encountered Problems and fixes

Written by Joel

In this section we will be describing issues and fixes that occurred while developing the pieces.

5.1.6.1 The contains method

Written by Joel

While prototyping our pieces, we we're using the so called Path shape to create our pieces. This allowed us to use a built in contains method, which was included in the special shape. When we began mapping images onto the pieces, we faced a severe problem, the Path shape didn't allow for images to be mapped onto them. We then had two choices, we could either make our own contains method, or our own mapping method. We researched both options, and decided that creating a custom contains method would be the easier and more reliable option, since we found in depth papers, describing how one could check if a point is contained within a complex polygon. The paper we used to construct our custom method can be found 3

5.1.6.2 Image mapping

Written by Joel

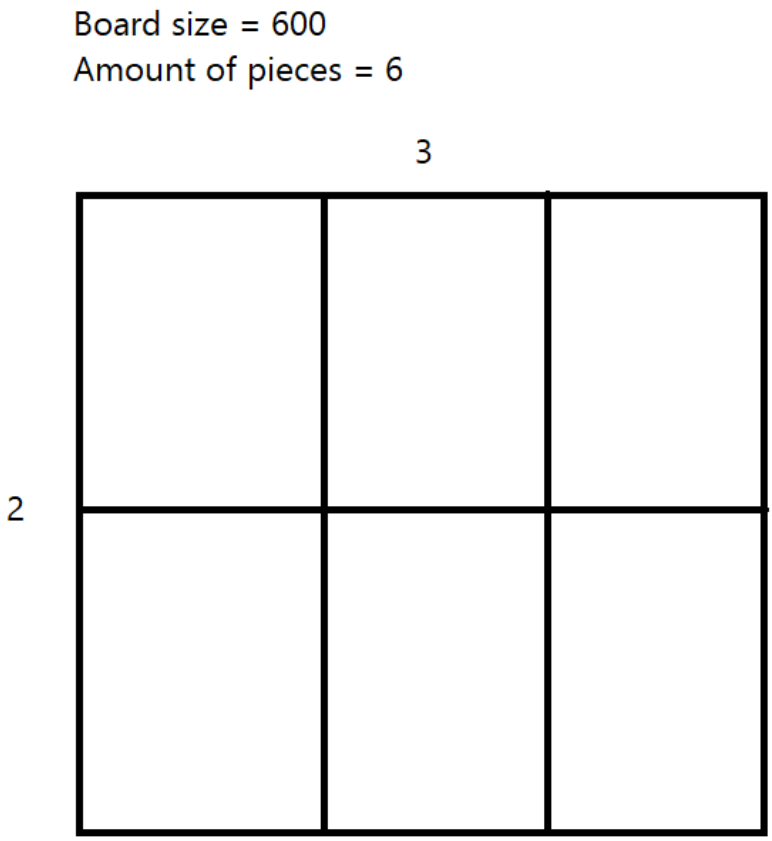
As seen in section 5.1.5.3, when mapping images onto the pieces, the distortion of the pieces result in the image being distorted as well. One probable solution to this, was to access the native array of the images and construct an algorithm that could find the correct image cutout, giving the vertices. Since the mapping of images onto our pieces wasn't one of our highest priority goals, we chose to accept the artistic feel, that the distortion of the images resulted in.

5.2 Generating and reading a puzzle

5.2.1 Generator

Written by Carl

The generator takes three parameters, a board size, piece amount and the number of distortion points. The generator then takes the given amount of pieces and finds the greatest factors of that amount. This enables it to divide the board into evenly sized squares, while dividing the board, the generator will distort the right- and bottom side of each piece and pass the distorted points on to the next piece, to create a matching left- and top side. In the following example, the size is set to 600, the amount of pieces is set to 6 and the distortion points are set to 0. Since the two greatest factors of 6 are 2 and 3, the board is divided accordingly.



When distorting our pieces, we set the following rules for our distortion points

to prevent complications:

- The height, h, of the distortion point measured from the distorted side has to be smaller than the length, l_1, from the distorted point to the corresponding side.
- The height, h, of the distortion point measured from the distorted side has to be smaller than the length, l_0, of the corresponding side divided by two.

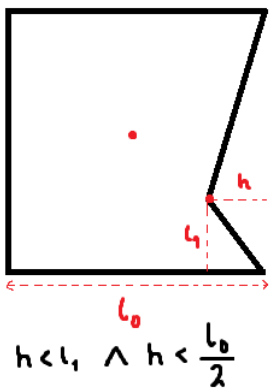
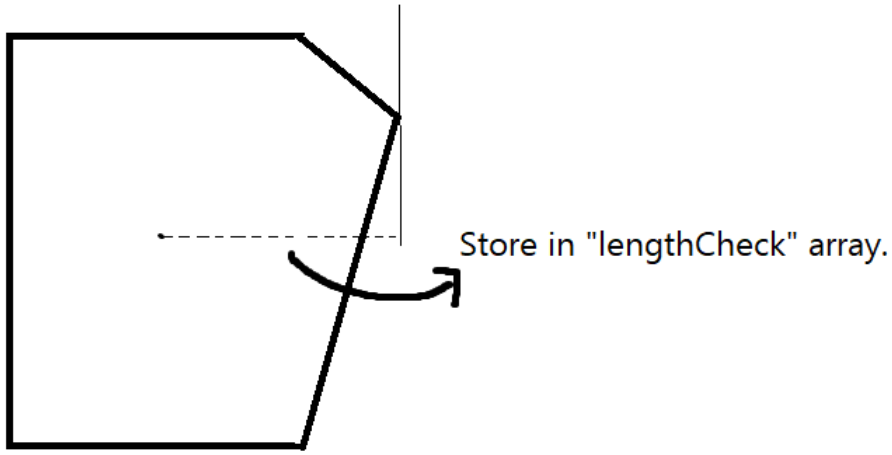


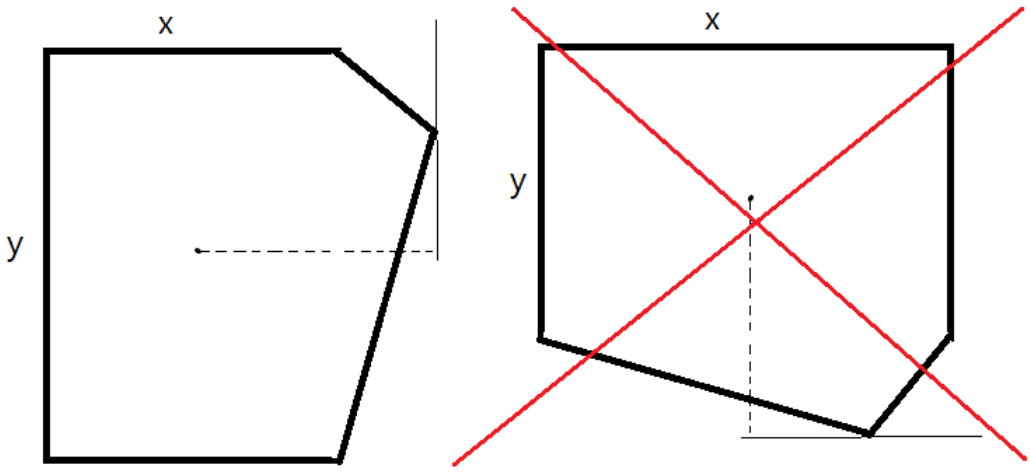
Figure 7: Legal distortion point

This prevents distorted sides overlapping the center, as well as preventing the distorted lines to intersect a different side of the piece, since the angle from the corner of the piece to the nearest point in never greater than $\frac{\pi}{4}$.

To ensure that we don't create duplicate pieces, we store the distance from the center of each piece to the first distorted point and that given piece, and every time we distort a new piece, we make sure to use a different distance between the new center and the first distortion point.



If the pieces have the same length and height, the same precautions will be taken when distorting the bottom side.



If x = y, then the piece on the right can not be generated.

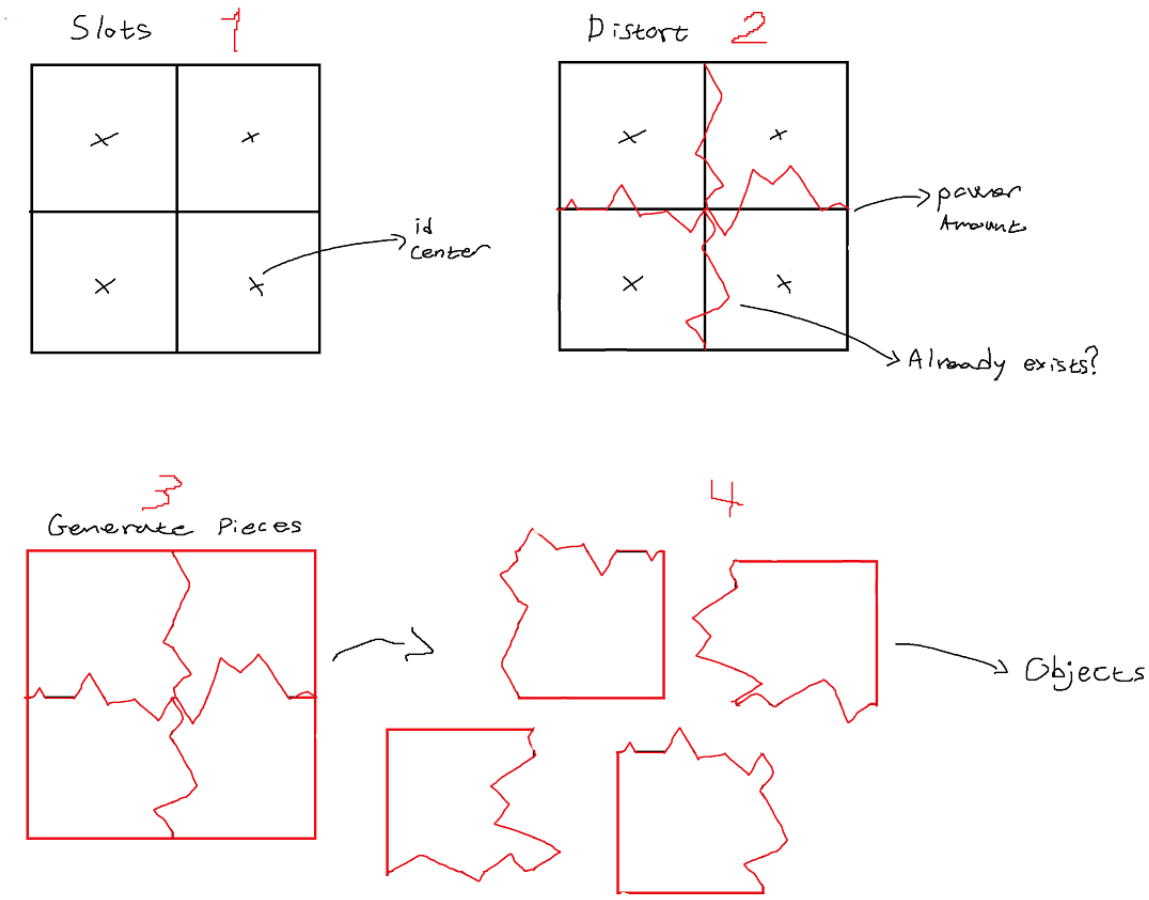


Figure 8: Example of generating four new pieces

5.2.2 Reader

Written by Alexander

The piece reader is a simple reader that takes a JSON file with some puzzle pieces. When the piece has been read in, the reader find a average point of all point in a piece. This is average is subtracted from all points in the piece, this i so the piece matches our datatype, where all points in a piece is defined by its own center.

5.3 Implementation of identical pieces

Written by Alexander and Carl

To check if pieces are identical, we follow the three steps described in the anal-

ysis section 3.3. The implementation itself is described in depth in this section.

- In step number one we start of by looking at the circumference of all the given pieces, if several pieces have the same circumference[5] we add them as pairs to an array, such that a pair contains two pieces with the same circumference.
- In step number two we check each piece in our array, if the piece contains three points on a straight line, we remove the middle point. This is necessary for both step two and step three. After removing the excess points, we make sure that each of the pairs have the same amount of points, if they don't, we remove them from the array.
- In step three we look at each pair in the array, for every piece we store an array containing the angle of a point and the corresponding length of the following side, this will be referred to as an "angle-length array" . Then we compare the two new arrays, if they aren't identical we shuffle the second array, by moving the last element to the front and then we compare again, we do this until we have tried all positions for the second array. If they are identical at any point, we know that the pair of pieces is identical as well. The method can be seen in the illustration below.

[5] The definition of circumference is: the sum of the lengths of all sides of a given piece.

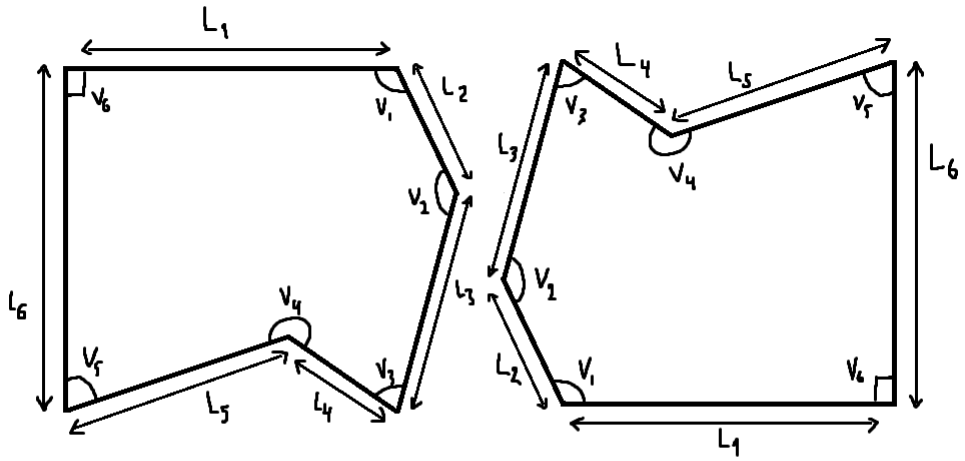


Figure 9: Two identical pieces, and their angle-length arrays

We have had different considerations for step three. In the beginning we thought the easiest way of comparing two pieces, would be to put them on top of each other and rotate them, until one could fully cover the other. We realised that our new method would be way easier to implement, therefore we changed it.
To show the user that two pieces are identical, we coloured them in the same colour, using a new colour for every distinct identical pair.

5.4 Implementation of angle computation

Written by Alexander and Carl

For both the "Identical Pieces" checker and the "Puzzle Solver" we make great use of an algorithm to determine the angles directed towards the center, for each point on the edge of a puzzle piece. While making this algorithm we ran into a few problems and we had to reevaluate our algorithm several times.

5.4.1 The basis of calculating angles

Written by Alexander and Carl

When calculating the angle we know the following:

- The coordinates of three points on the puzzle piece, A, B and C. We want to find the angle in point B
- The center of the piece. The center is for all pieces (0,0) as all pieces are defined from their own center.

When calculating the angle in point B we start by defining two vectors BA and BC and then we use them in the following formula:

$$\theta = arccos(\frac{\vec{BA} \cdot \vec{BC}}{|\vec{BA}| \cdot |\vec{BC}|})$$

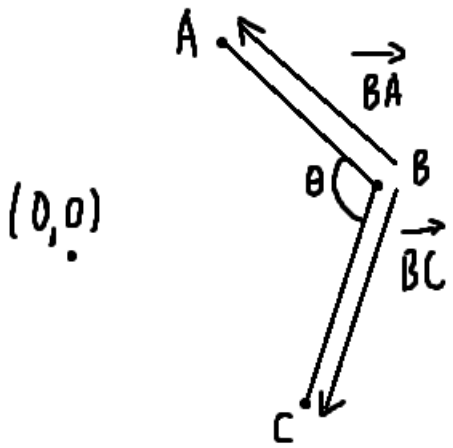


Figure 10: Angle given three points

This formula works perfectly as long as the actual angle is not larger than π. The problem arises when the angle is larger than π, see figure 10. The formula will always find the smallest angle due to the cosine function. Because of this we need to find out if the angle is greater than π, and if it is, the angle becomes:

$$\theta = 2 \cdot \pi - arccos(\frac{\vec{BA} \cdot \vec{BC}}{|\vec{BA}| \cdot |\vec{BC}|})$$

We need to find a way to determine if the angle is greater than π, so we can determine which formula to use. Finding the actual angle proved to be a problem and we had several methods which ended up being flawed, so we had to come up with a new one. In the next section we will describe the steps we went through to end up with the working method.

5.4.2 Point and line method

Written by Alexander

The first method we came up with to determine if the angle is greater or smaller than π, uses the following principle. We make a line between A and C and determine the shortest distance from the center(0,0) to that line. We can find the distance using the formula:

$$distance = \frac{|(x_B - x_A)(y_A) - (x_a)(y_B - y_A)|}{\sqrt{(x_B - x_A)^2 + ((y_B - y_A)^2}}$$

We also find the distance from point B (the point were we want the angle) to the center. Once the two distances have been found we compare them and if the distance from the line to the center is greater than the distance from the point to the center, the angle is greater than π, otherwise it is smaller or equal to π. See figure 11. This method does not work every time, an example could be the one on the most right of figure 11. In this case, the angle is bigger than π, but using the method described it will determine that the angle is smaller π, because the distance d_1 is larger than the distance d_2.

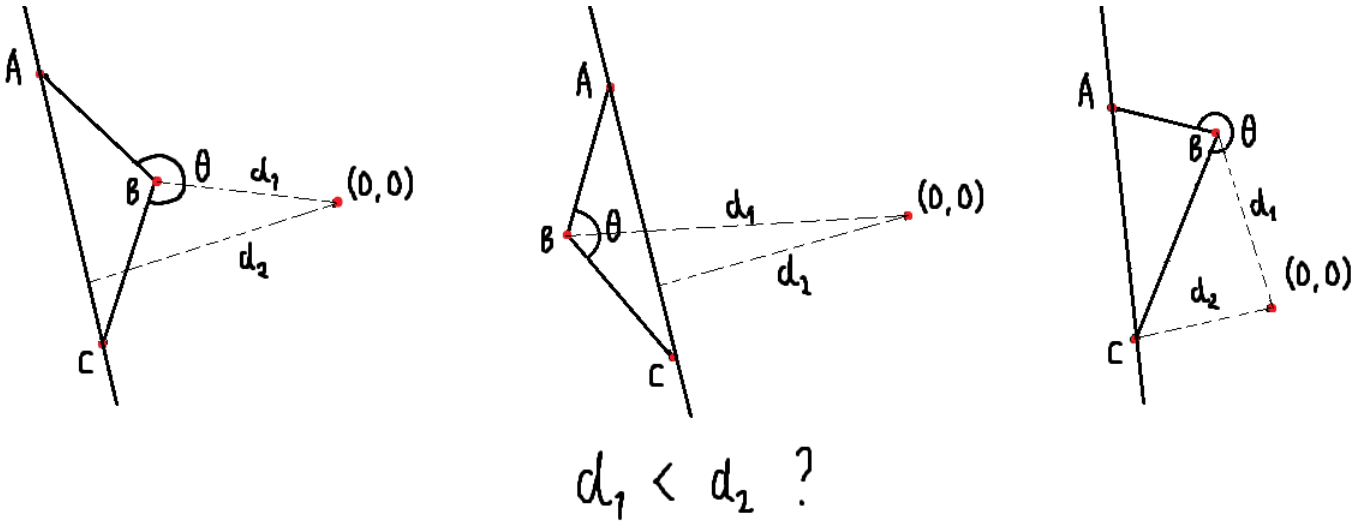


Figure 11: Angle given three points

Method one did not work in all cases, so we developed another method.

5.4.3 Triangle method

Written by Carl

The second method we came up with to determine if the angle is greater or

smaller than PI, makes use of a triangle generated from the points A, C and (0,0). We would check if the triangle contained B inside of it. If the triangle contained B, the angle was over π.
To determine if the triangle contained B, we constructed three smaller triangles made from respectively (B, A, (0,0)), (B, C, (0,0)) and (B, A, C). See figure 12
If the sum of area the three smaller triangles equals the area of the large triangle, the large triangle would contain the point B.
We ended up having the same issue as in the first method, so we had to reevaluate once again. See figure 13'

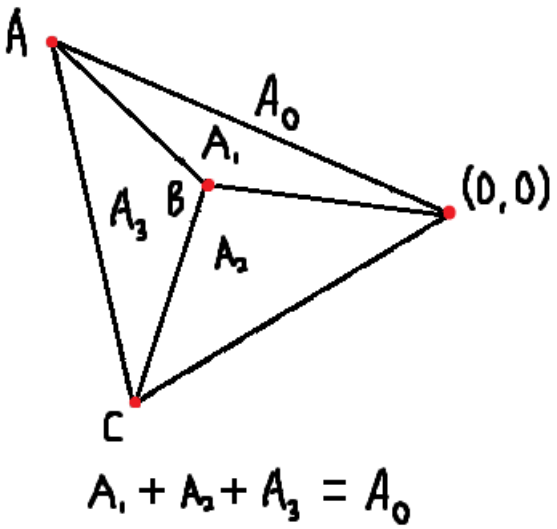


Figure 12: Angle given three points

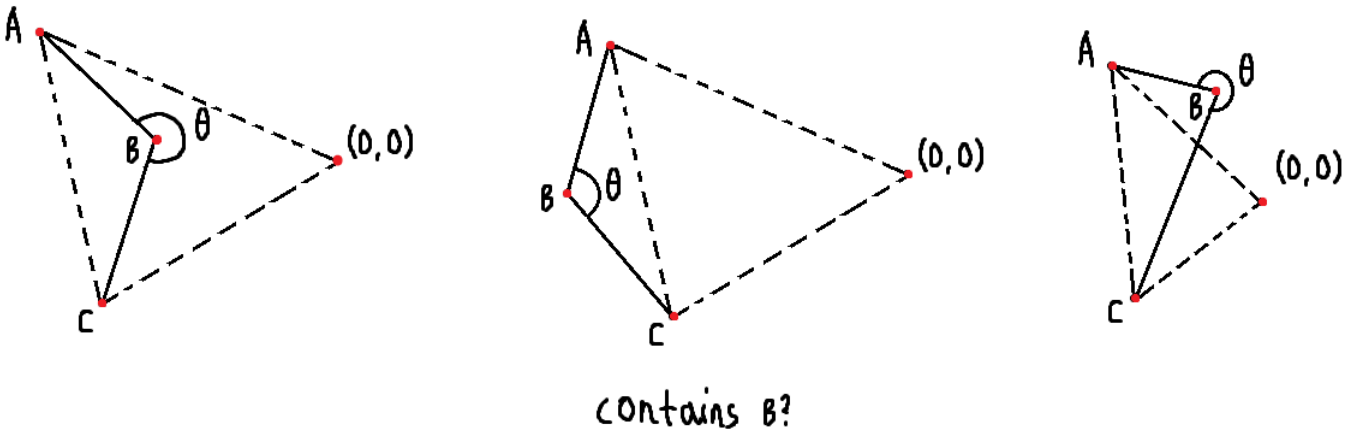


Figure 13: Angle given three points

5.4.4 Two lines method

Written by Carl

The third method we came up with to determine if the angle is greater or smaller than PI, makes use of two lines generated from (A, C) and the parallel line which intersects with B.

We then compare the shortest distance from (0,0) to each of the lines. If the distance d_1 is smaller than the distance d_2, 14 we know that the angle is larger than π. This works in all the cases, assuming the piece does not "fold" on itself, see figure 15.

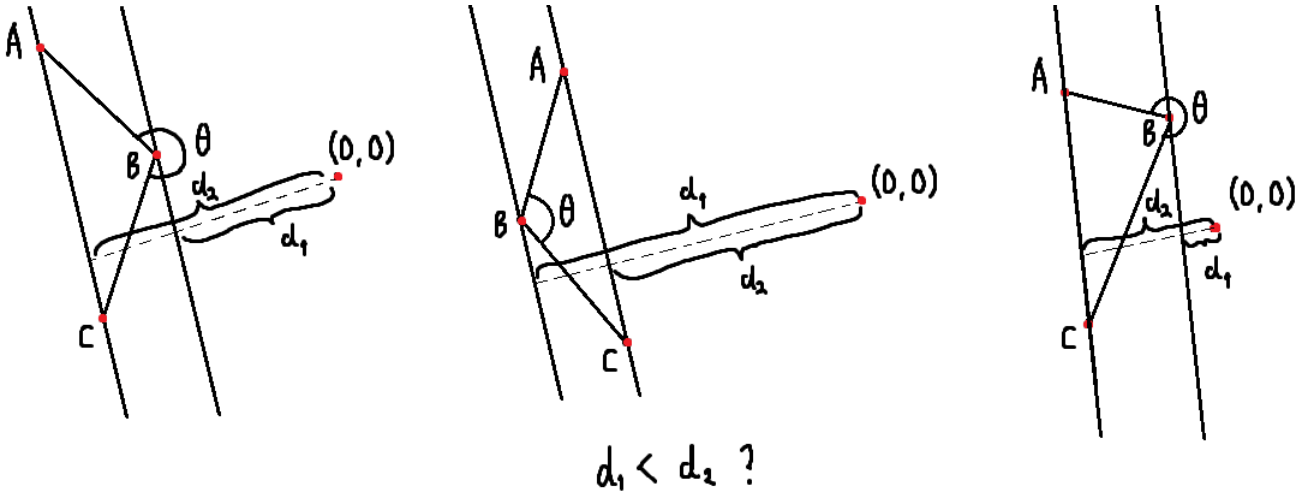


Figure 14: Angle given three points

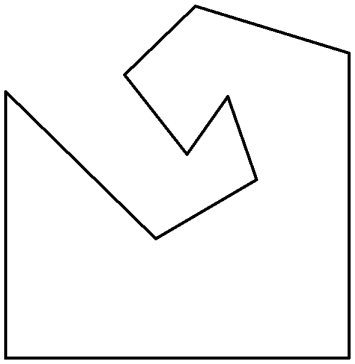


Figure 15: Broken piece

5.5 Implementation of solving algorithm

Written by Alexander

Our algorithm to solve the puzzle uses several steps.
We start off by removing any excess points [6], using our "three points on line" method.
After removing the excess points, we traverse every piece, and for each piece we produce the array containing the angle of each point and the corresponding length as mentioned in section 5.3.

[6] The definition of excess points is: any point P2, located on a straight line from P1 to P3.

We then group our pieces into three categories, corner pieces, side pieces and middle pieces. This causes some limitations for our algorithm, which are mentioned later on. The reason we group the pieces is to find the neighbours of every piece. When we group the pieces, we can reduce the comparisons needed, and therefore the computation time.
The grouping is done by looking at the amount of points for each piece, then we put pieces with the same amount of points in the same group. We know that the group with the fewest amount of points is the "corner piece" group, since they have less neighbours, and therefore fewer points.
We are now creating an "adjacency array" in the following steps. We go through each group, starting with the "corner group", and traverse the pieces in the given group. For every piece we match the "angle-length array" for that given piece with the "angle-length array" of every other possible neighbour[7].
We have a match if we find two angles with a sum of $2 * \pi$ and the corresponding lengths of the angles match up numerically, see 16.

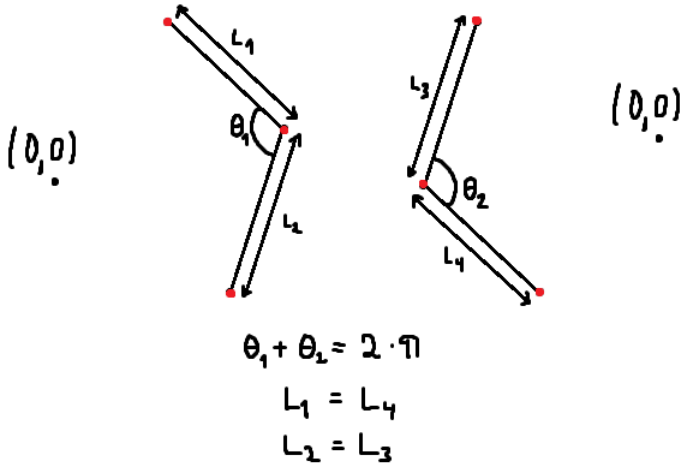


Figure 16: Two Matching Angles

If we find a match, we produce an instance in the adjacency array, containing the piece index of the given piece and the index of the matching piece. Hence we end up with an instance in the adjacency array containing the piece and all of its neighbours, underneath is an example of an adjacency array for a 2x2 puzzle.17

[7] The possible neighbours are decided by which group they are in, and how many piece our puzzle is made up of

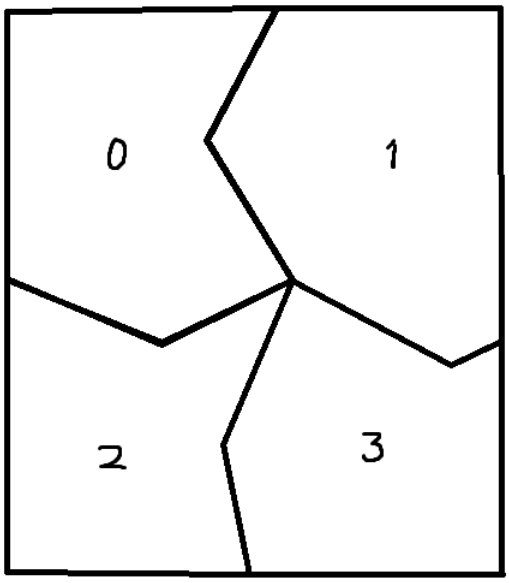


Figure 17: Adjacency array for 4 pieces

Whilst creating our adjacency array, we also create a "matching array", this is simply an array containing the index of the two matching pieces, as well as the index of their respective "angle-length array", in which we found a matching angle and length pair. This is used later on to find the matching points of two pieces. The matching array for the previous 2x2 puzzle would look like this. 18

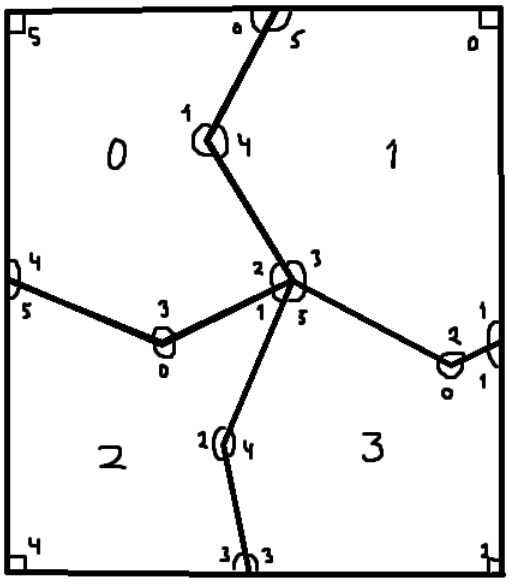


Figure 18: Matching array for 4 pieces

If we look at the instance at index 0, we see that piece 0 matches with piece 1 on the 1st angle in piece 0 and the 4th angle in piece 1.
Once we created the adjacency array and the matching array. We needed to find a method of placing the pieces correctly with the right rotation, as mentioned in 3.5.

5.5.1 Piece placement algorithm

Written by Alexander

The initial solution was to find a path through the pieces. Using the adjacency array and the piece groups we started by choosing a corner piece. As long as the puzzle follows the row and column form mentioned in 5.2, we can be sure that from a corner there exists a path traversing each piece exactly once. The path is generated recursively, and terminates once a path containing all pieces exactly once has been found.
This way we could calculate the angle[8] of the next piece, just by summing the angle of the previous piece in the path and the angle of the next piece.
In our generated puzzles, our algorithm would find the first path rather quick, because it searched depth first. Although when we tested our algorithm on cases

[8] The angle mentioned in this section, is the value that one piece needs to be rotated to match the rotation of its neighbour (the previous piece).

provided as material, the algorithm could take up to several minutes.
Therefore we developed a new method to place the pieces. The algorithm uses the adjacency array. It starts at piece 0, then goes to all the neighbours of that piece, using the adjacency array. This makes the algorithm run through the pieces in a sort of indirect tree structure, where all pieces are rotated in relation to the previous piece, all originating in piece 0. When we rotate the piece, we also give it a new position according to its neighbour other pieces, depending on the path we took.

5.5.2 Finding the angle and placement of a piece

Written by Carl

The rotation of the initial piece is used as the standard rotation of the puzzle. When rotating a piece based on the rotation of another piece we start finding the matching points of the two pieces match. To find these points, we look in the matching array and backtrack through the "angle-length array". In the "angle-length array" we find the points that were used to find the matching angle between the two pieces. Since we know that these points must be the matching points of the two pieces. Once we have the points we can calculate the angles between the sides of the two pieces. We calculate the angle by taking two point on each of the sides, making a line and calculating the angle between the lines. Once the angle is found we add it to the previous found angle, this way all the pieces end up having the same rotation as the initial piece.
When an angle is found, we calculate the offset which the piece has to be moved to match with its neighbour. We find the position of the middle point taken from the three matching points, and then we create a vector from this point, to the corresponding point in its neighbour piece. This is the offset, which we need to move the piece. This works because all points are not defined as a point in the plane, but their relative position to the center of a piece. Therefore making a vector between the points describes the offset that one piece has to have for the pieces to fit together.

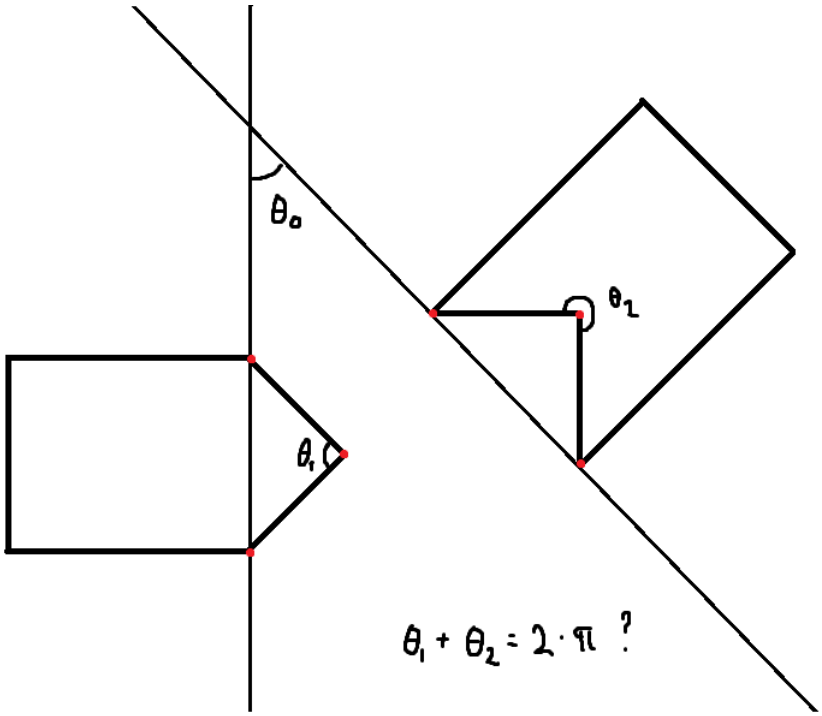


Figure 19: Rotate two matching pieces

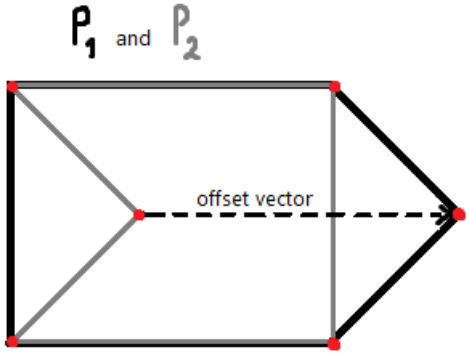


Figure 20: Offset vector for two matching pieces

5.6 Completion check

Written by Joel

In this section we will be describing the implementation of the completion check.

For the definition of a complete puzzle see section 3.6.

To accomplish the task of checking if the puzzle is complete, we utilize and reuse the ideas and implementation of the snapping method 2, described in section 5.1.4.1 and 5.1.4.4. The main difference between the snapping method and the completion check, is that we do not only check if one piece maintain the logic of a piece snap, instead we check every piece, ass well a decreasing the snapping range, to be less than the one used in the original snapping method 2. By doing this we can assure that all pieces are positioned directly besides their corresponding neighbor pieces, and that they are positioned at the correct direction in relation to the other pieces. If all pieces live up to the modified snapping method 2, we can conclude that the puzzle must be complete.

When a puzzle is completed a visual clue, reading "Congratulations, you have completed the puzzle!" is displayed on the screen, and thereafter the user can choose to generate a new puzzle.

5.7 UI components

Written by Joel

In this section we will be describing the implementation and functionalities of our custom UI-components.

Visual components act as a direct information gateway between the user and the program. To construct our visual components we use the processing library, where we can utilize processing's visual capabilities. Processing has functions such as drawing a rectangle, line, ellipse and so on, by combining these visual competences, with processing's ability to listen for specific user input, we can create complex UI-components, such as the ones listed below.

All the UI-components are created as separate Java classes, containing both the logical and visual parts needed to construct each individual component.

5.7.1 Menubar

Written by Joel

The menubar acts as a list, that can store the different UI components, I.e. the text-input, toggler, button and slider. The menubar is the main visual component, and keeps the other components organized within itself.

5.7.2 Text-input field

Written by Joel

The text-input field lets the user type customized text into the field, where after the text-input field which then can be used elsewhere in the program. In our

program we use one text-input field to get the path of the puzzle, that should be read into the program.

5.7.3 Toggle switch

Written by Joel

The toggle switch has one main functionality, namely that it can switch between two values. It can then return the state of the switch, describing which value is currently selected, which can then be used elsewhere in the program. In our program we use one switch object to switch between the generation of a random puzzle, and reading in a predefined puzzle.

5.7.4 Slider

Written by Joel

The slider is a draggable object, that has a minimum and maximum value, using the draggable feature, the user can choose any number in between the minimum and maximum. In our program we use two sliders when generating random puzzles, these sliders are used to set the amount of pieces and distortion points in the generated puzzle.

5.7.5 Button

Written by Joel

The button is a clickable object, that executes an event when clicked by the user. In our program we use three buttons, one to generate a new puzzle, one to solve the puzzle and one to solve the rotational property of every piece.

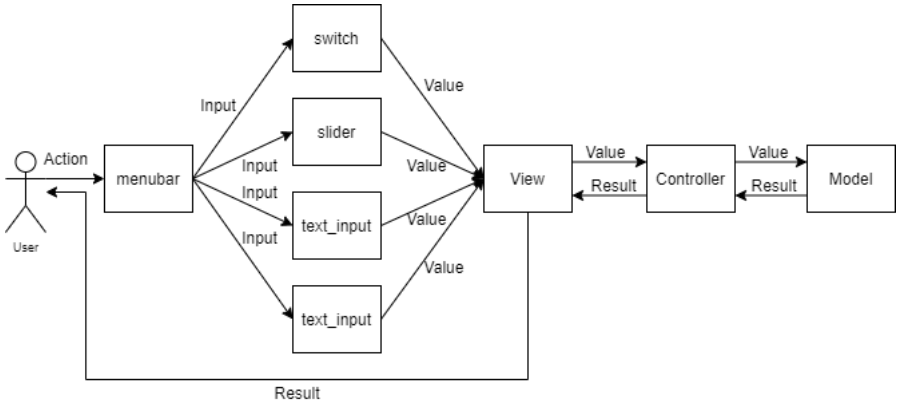


Figure 21: Shows the relation each UI component has with the rest of the program.

5.8 Class diagram

Written by Alexander and Carl

The class diagram for the program can be found in the appendix 25. To see the full picture with options to zoom and pan, follow this link https://drive.google.com/drive/folders/1DMrDVWZepL4hufDyhXGhEXIowXQ757hQ?usp=sharing

6 Limitations of the solving algorithm

Written by Alexander and Carl

The algorithm was made to solve puzzles following the column and row format mentioned in 3.2. Another limitation to the solver is that it only matches on one angle and the lengths of two corresponding sides. The issue with this was when two pieces matched on one angle and two side lengths, we immediately labelled them as matches, meaning we could find several wrong matches with this method. This is never the case for the puzzles generated by our generator, since every piece is unique because of the first angle, therefore we will never have any "fake" matches.

6.1 Limitations of matching on a single angle

Written by Alexander

The limitation of matching on one angle and the corresponding two sides, is visible when trying to solve classic puzzles. When looking at the notches, which the classic pieces are supposed to match on, we immediately find an angle of $\frac{\pi}{2}$, and a corresponding length. This will be the same for the other classic pieces, since they all have an angle of $\frac{\pi}{2}$ to start of their notch. Since the angles are the same, the matching is decided by the sides lengths. This causes the points to match incorrectly 22. Therefore the angles found in order to rotate the piece as well as the found center, will be incorrect.

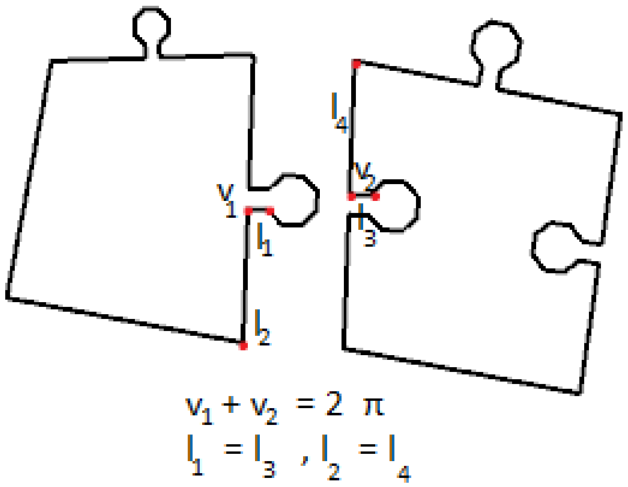


Figure 22: Error matching classic pieces

6.2 Limitations of column and row method

Written by Carl

We also encounter a problem with puzzles where one piece is bigger than the other two, since we are attempting to group the pieces into categories based on how any points each piece has. Here the issue is that we make the assumption that pieces can be placed into groups 5.5. When one piece has an irregular size, this irregular amount of points will cause the grouping to be done incorrectly, and therefore pieces would be matched incorrectly. The following is an example, where the bigger piece would be placed in different group than the corner pieces, because it has more points 23.

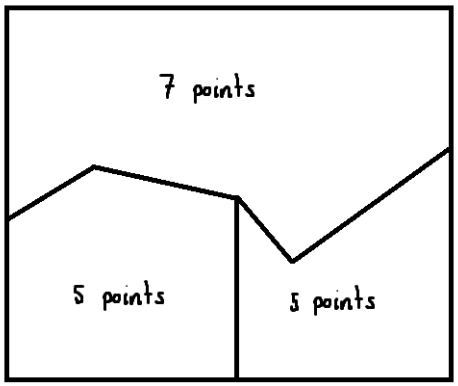


Figure 23: Error matching one big and two smaller pieces

We might be able to solve puzzles like this if we removed the grouping, but we would have to sacrifice computational time, which we don't want to.

6.3 Limitations of piece snapping

Written by Joel

A big problem with piece snapping, is that it only works for puzzle pieces generated by our own generation algorithm. This is because we need the base width and height of the puzzle pieces. For the generated puzzle we know the amount of pieces per row and column, and the size of the board, therefore we can calculate the width and height of the puzzle pieces. When reading a puzzle from a file, we are not given this information, and therefore the piece snapping will not work as intended.

A simple solution to this problem is to calculate how many pieces there are per row and column in the puzzle being read. This could possibly be achieved by using our puzzle solving algorithm to first solve the puzzle, and afterwards go through the solved puzzle, finding the amount of pieces in the row and column.

7 Testing and Debugging

Written together

In this section we will describe our methods of testing and debugging the program.

7.1 Visual debugging

Written by Joel

To get a better grasp of the pieces we wanted to create a visual help, that could show us properties, such as rotation, position, distortion, origin, and so on. To accomplish this task, we created UI elements for each piece, that could be drawn over the actual piece, as a debugging layer.

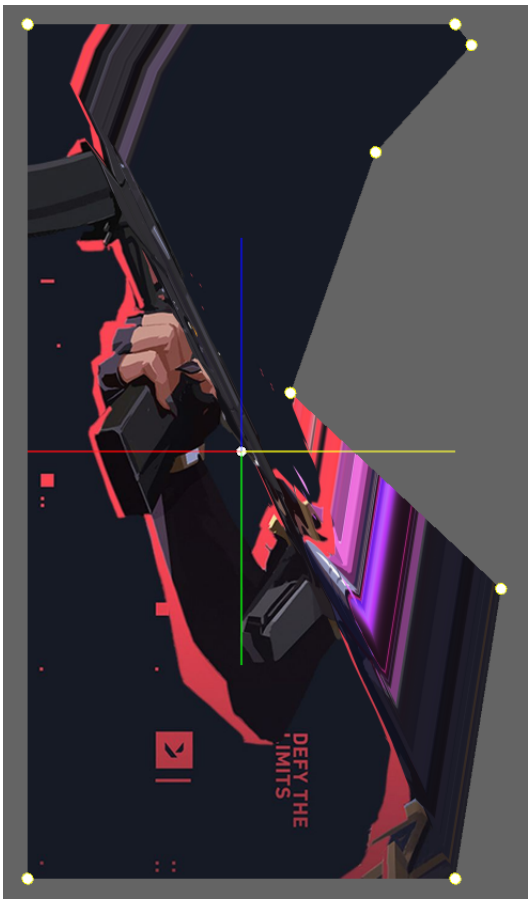


Figure 24: Shows a single piece and the overlaying debugging layer.

The debugging layer has allowed us to get a better understanding of how our pieces are generated, and where there could be some potential bugs, as well as allowing us to get a more straight forward visual representation.

7.2 Testing

Written together

In this section we will be describing our main methods of testing our program.

7.2.1 Manual testing

Written together

One of our most prominent ways of testing the program, was through manual testing. By manual testing we are referring to running the program and testing new components, when they are implemented. By doing this we got a grasp of

the feel and interaction of the new components. This helped us realize whether the new components worked as intended. Making sure that a component was working as intended was important since, the testing of more advanced features often relied on previously made features. A simple example of this could be the testing of identical pieces functionality. Here we expect that pieces are displayed correctly, so we could check whether the algorithm computes a correct result.

7.2.2 Theoretical testing

Written together

Another method of testing our program was what we like to call theoretical testing. By theoretical testing we refer to theorising whether a component would work as intended, this was normally realised when having brain storms about our components, where a sudden possible bug/problem became clear. After conducting a theoretical test, we used manual testing to reconstruct the problem, and see how the program handled the problem in practice.

8 Project management

Written by Joel

8.1 Version control

Written by Joel

As version control we used Git through GitHub, allowing us to easily implement share and merge components that had been developed by different group members. We also used it to keep a backup of the program and old versions, at all times.

8.2 Assigning tasks

Written by Joel

At the beginning of the project we used an online service called Trello to structure which components that had to be developed. Using this service we could assign ourselves to components that we were currently developing, allowing our group members to easily grasp which components were in development, which were done and which still needed to be developed.

8.3 Time management

Written by Joel and Carl

To the best of our ability, we tried to switch between developing and writing the report, this way we could develop a component and write the section of the report, tied to the component. Otherwise we kept a close eye on the development process of each other, helping out if some components were taking longer than

expected. This way we could keep up with our time schedule, and not fall behind.

9 Conclusion

Written together

An obvious way to conclude on this project is by revisiting the MoSCoW model, which we introduced in section 2.2. Using the MoSCoW model we created a priority list, ranging from implementations, which were a "must", "should", "could" or "wont". By following this priority list, we managed to implement almost every point in every category, except piece merging, since it was not a priority, and the category "wont" which, given by the name, were implements we didn't plan on developing. The MoSCoW model allowed us to have a clear path to follow, and what to prioritize. Without the structure of the MoSCoW model, it can be assumed that we would not have implemented as many meaningful components.
We also analyzed the puzzle problem, splitting the problem into minor sub-problems:

- Creating an interactive piece
- Generating a puzzle
- Checking if two pieces are identical
- Solving a puzzle only based on the pieces
- Checking if the puzzle is solved

Through our development, we managed to create interactive pieces, which could be moved and rotated by the user. We also managed to add image mapping to the pieces, allowing for easy visual clues when solving the puzzle. This implementation was streamlined using the processing library, which contributed to many functionalities in our final product. Furthermore we managed to construct a method of generating random puzzles using our own piece datatype. We also created an algorithm to read sample JSON files, containing information about a puzzle, and converting them to the datatype used in our application. We managed to check if two pieces were identical by using a combination of circumference, vertices, angle and length checking, as well as creating visual indications by coloring two identical pieces the same distinct color. We created an algorithm that could solve a puzzle, from nothing but a list of vertices for every piece in the puzzle, and we also created an algorithm, which could go through the puzzle pieces, and check if the puzzle pieces were positioned correctly, meaning the puzzle had been completed.
All of these implementations contributed to a functional puzzle application.

To accomplish all these feats, we used different tools, ranging from math tools,

such as GeoGebra to GitHub as version control, but the most important addition we used was the processing library, which most importantly allowed us to create a graphical user interface, as well as providing us with functionalities for graphical components.

As a final concluding note, we finished the project and completed almost every goal we set out to do. Every major component works as intended, and we had time to work on additional features, which were not mandatory, but we felt that they would contribute positively to the application. As of social experience, we have worked together on multiple occasions, and we know each other and our expectations, which is of big importance in such a project.

10 Appendix

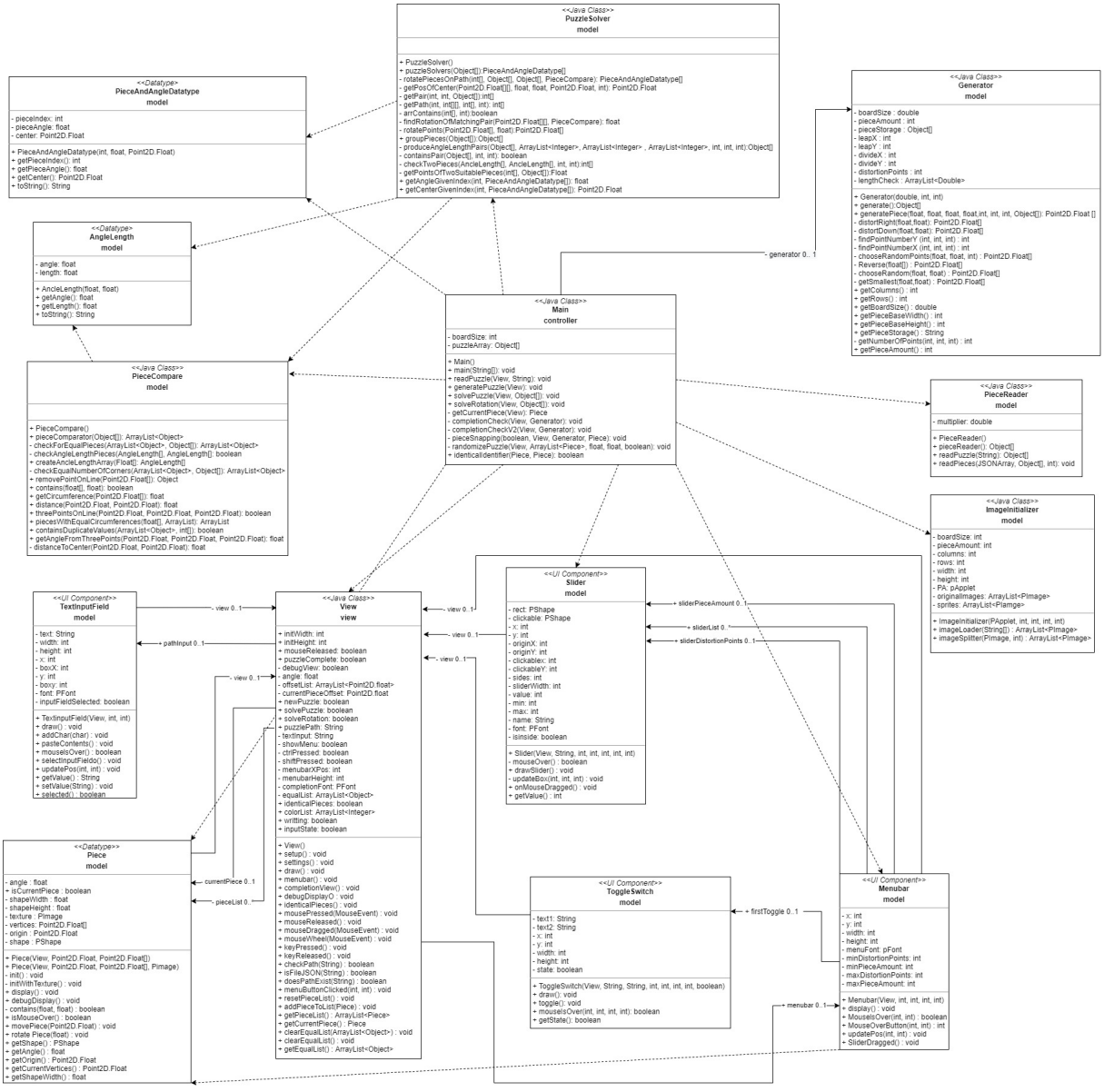


Figure 25: Class Diagram

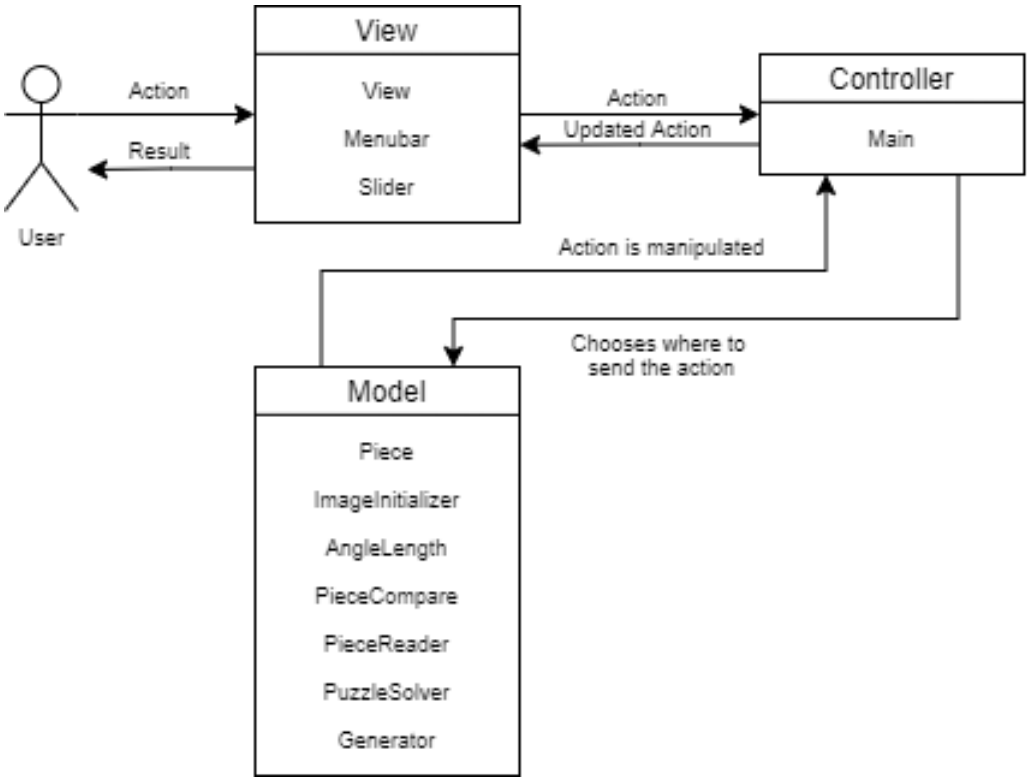


Figure 26: Design of the program in relation to the implementation

References

[1] StackOverflow Q & A https://stackoverflow.com/

[2] Information on Processing https://processing.org/reference/

[3] Point Inclusion in Polygons https://web.archive.org/web/20161108113341/https://www.ecse.rpi.edu/Homepages/wrf/Research/Short_Notes/pnpoly.html

[4] Reverse an array https://www.tutorialkart.com/java/java-array-reverse/

[5] The Github repository for this project https://github.com/Frinkel/SoftwareProject_Puslespil.git