

2D Shape Area Generative Estimation Algorithm

Student ID: 2003128

1 PROBLEM

The difficulty of calculating the area of a two-dimensional (2D) shape fluctuates highly. Regular polygons have straightforward formulas, accurately finding the bounded area of the shape that are easy for even children to grasp, whilst other shapes need creativity and lengthy methods to be able to find their area. Difficulty increases with non-polygon shapes such as ovals, quatrefoils, tori, as well as irregular polygons that can have many different forms. Take, for example, Figure 1 and Figure 2. These are highly unusual shapes whose areas have no intuitive way of being calculated.

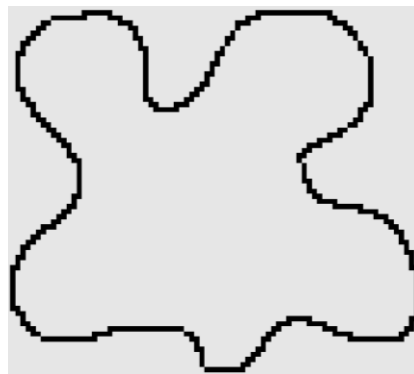


Figure 1



Figure 2

The initial approach to solving such a problem, one that is taught from a GCSE mathematics level, is to break the shape down into smaller components, and summate the area of each individual component to get a total. This works very well in some cases, especially when the shape can be broken down into a set of regular polygons or ovals, but less well in other, more complex shapes. An interesting solution to a particularly irregularly shaped room, shown in Figure 3, makes use of this technique, splitting the room into three components made up of rectangles with semi-circular ends. (Walls, 2010)

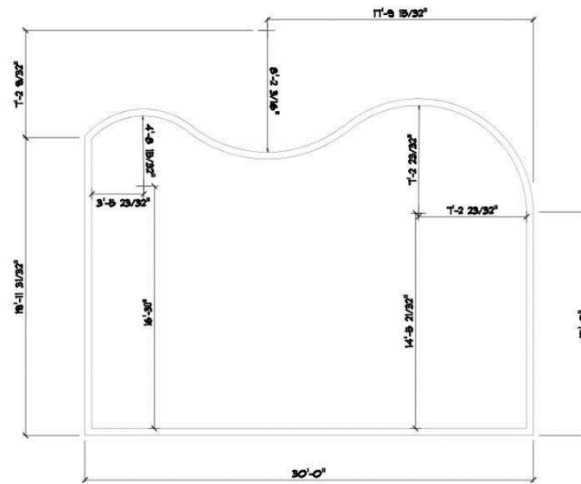


Figure 3

After taking the integral of each of these components, the area of the room can be found. By inspecting a shape, formulas can be created to work out its area quickly and efficiently, however this is not an easy task to do programmatically. Computers may not be able to identify these sub-shapes to apply the appropriate formulas without the aid of concepts such as computer vision and artificial intelligence. (Dickson, 2020)

Due to this, an alternate solution may be needed that can work on any shape, give a good estimation of the shape's area, and can be easily understood – even by the layman. The solution proposed in this document aims to achieve all three of these criteria.

2 EXISTING SOLUTIONS

2.1 ARTIFICIAL INTELLIGENCE

The use of AI could arguably be the most powerful solution to calculating the area of any given 2D shape. The approach of finding sub-shapes within a larger shape could be done for much more complex shapes with the use of AI; being able to recognise these patterns extremely quickly and accurately after training. Not only this, but it could be parallelised to further increase efficiency very easily.

Shape detection is a relatively easy task for AI and can be done with as little as one hidden layer for simple shapes and a small training dataset. (Rieke, 2017)

Generally, employing the use of an artificial intelligence is a much more complex task than alternative methods and does have the disadvantage of needing training to be able to perform the task to a desired accuracy. It can make mistakes and often the inner workings of an AI are unknown, meaning extra review is needed to assure correct performance. This all requires far higher overhead, understanding and specialisation than other approaches.

2.2 ORTHOGONAL TRAPEZOID METHOD (OTM)

This method of calculating a shape's area is a programmatic implementation of the most common approach to solving the problem, as described in the problem identification section. By splitting a

polygon into orthogonal trapezoids, the shape area can be found by applying the trapezoid area formula:

$$Area = \frac{1}{2}h(b_1 + b_2)$$

Where b_1 and b_2 are parallel sides of the trapezoid and h is the perpendicular distance between the parallel sides.

This is achieved by obtaining the coordinate locations of all vertices the polygon has and drawing horizontal lines through each vertex. This creates a set of trapezoids that can be found and stored based on the line and vertex intersections of these horizontal rules, then used to find the total area of the polygon. (Wijeweera & Kodituwakku, 2017)

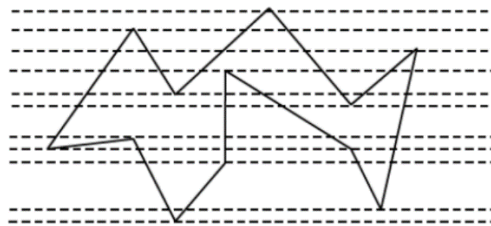


Figure 4

This could also theoretically be used to calculate shapes with curved lines, provided the shape had vertices, using the trapezoid rule of approximating area by using trapezoids (Sterling, 2005). This would yield less accurate results than with standard, straight-line polygons, but would not be out of the scope of the implementation.

OTM does have limitations, however. Shape areas able to be found by this method must not only have vertices, but also the coordinate location of each vertex must be known. This prevents a lot of use cases for shapes such as ovals or tori, as well as when the location of a shape is unknown.

2.3 THE SHOELACE FORMULA

This is a mathematical algorithm used to find the area of a polygon with known vertices coordinates using cross-multiplication. This is done by following the following steps:

1. Store all the vertices in anticlockwise order in an array.
2. Calculate the sum of multiplying each x coordinate with the y coordinate from the vertex next in the array (wrapping back to the start of the array if needed).
3. Calculate the sum of multiplying each y coordinate with the x coordinate in the same way.
4. Subtract the two sums, taking the absolute value.
5. Divide the result by two to get the area of the polygon.

(101 Computing, 2019)

This is very rudimentary to implement programmatically and can be done very efficiently. From the research done by Ruhuna Journal of Science, compared to OTM, they found that the performance ranged from 22 times faster in the best-case scenario, up to 77 times faster in the worst-case scenario, as shown in Figure 5. The algorithm has a time complexity of $O(n)$ with n being the number of vertices and a space complexity of $O(1)$. (Wijeweera & Kodituwakku, 2017)

POLYGON	OTM	SLF	OTM ÷ SLF
1	4554	204	22.3
2	5345	241	22.2
3	9914	291	34.1
4	11,641	327	35.6
5	14,408	361	39.9
6	20,791	406	51.2
7	28,416	453	62.7
8	21,939	466	47.1
9	33,098	490	67.6
10	41,047	532	77.2

Figure 5 – Set of random polygons, the number of clock cycles needed to compute the area of the same polygon 10^8 times.

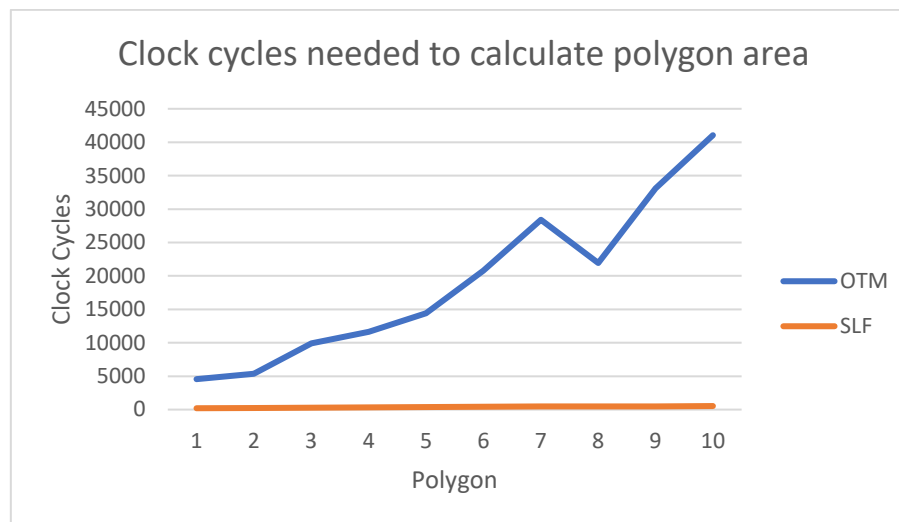


Figure 6

Due to the nature of the calculations being performed, a concern for this algorithm is generating an overflow error if the shape complexity is high enough that the sums are high-order numbers. Furthermore, this approach suffers from the same drawbacks as OTM, needing clearly defined vertices and only applying to simple polygons. Whilst a clear improvement from OTM, it is not without flaws as a general-purpose solution.

3 PRACTICAL IMPLEMENTATIONS

The real-world use cases of an algorithm that can find the area of any 2D shape are plentiful. One practical example would be carpeting the floor of an unusual room footprint without having a surplus or shortage of carpet. Room floor plans come in all shapes and dimensions, from standard rectangular shape to other exotic shapes depending on the architecture. A similar situation would be for sowing grass seeds into a lawn, again without any surplus or shortage of seeds.

Another example could be for an area of effect ability or item within a computer game. Being able to generatively calculate the area of effect would mean completely random shapes could be generated with no issue, allowing for a better game experience in certain situations. The fluid nature of the ink in the computer game Splatoon¹ is a very good example of this as the objective of the game is to have

¹ <https://splatoon.nintendo.com/what-is-splatoon-2/>

the majority of ink colour painted across the game arena, which requires the surface area of all the ink of the arena to be calculated. As seen in Figure 7 the paint is irregularly and randomly shaped – the ideal use case for the algorithm.



Figure 7

4 ALGORITHM SPECIFICATION

4.1 OVERVIEW

The proposed algorithm will take a cellular image of a shape and calculate the area generatively from a given starting position within the shape boundary. It will wrap an array data structure into a Grid class to be able to achieve this, reading and manipulating the elements of the array recursively to determine how many atomic points lay within the boundary of the shape.

Each atomic point on the grid will be considered a cell, and from a given starting cell the algorithm will make use of a simple cellular automata algorithm to exponentially calculate the new area based on surrounding cells for each discrete step, or generation, of the algorithm.

4.2 CELLS

To calculate the area of a shape, first some form of measuring device must be established. Real world examples would be a unit of measurement such as a metre or foot, or a digital medium such a pixel. To represent one of these measurements, cells will be used to show some atomic finite point on the plane where the shape image resides. The benefit of not describing any one measurement is that the approach can then be applied to any form of measurement with a simple logical substitution.

To approach the details of what information a cell will hold and how it behaves, first the definition of cellular automaton must be established. A cellular automaton A is a collection of cells on a grid that evolve over time based on a set of rules. This is specified by a quadruple $\langle L, S, N, f \rangle$ where:

- L is a finite square grid of cells (i, j) .
- $S = \{1, 2, \dots, k\}$ is a set of states. Each cell in L has a state $s \in S$. For the implementation, these integer states will be wrapped within an enumerator class `CellStates` for code clarity.
- N is the neighbourhood, describing the cells around any given cell (i, j) . A complete neighbourhood exists for every cell on the grid. The two common neighbourhoods are a five-cell von Neumann neighbourhood (Figure 8): $\{(0, 0), (\pm 1, 0), (0, \pm 1)\}$, and a nine-cell Moore neighbourhood (Figure 9): $\{(0, 0), (\pm 1, 0), (0, \pm 1), (\pm 1, \pm 1)\}$. For the implementation, the von Neumann neighbourhood will be used to prevent cells outside the boundary of the shape from being checked.

- f denotes the update rule space. This computes the state of a given cell for the next step based on its neighbours' states and its own state.

(Javaheri, et al., 2015)

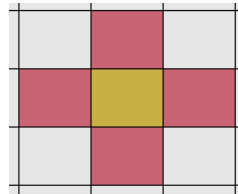


Figure 8 – von Neumann neighbourhood

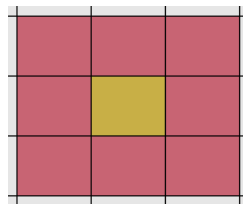


Figure 9 – Moore neighbourhood

The possible states of any one cell are as follows:

- Default: the representation of the plane, or an “empty” cell.
- Painted: denotes the boundary of a shape on the plane.
- Start Point: the location where the algorithm will begin.
- Area: represents a cell that has been parsed by the algorithm as the area within a shape.

With S and N defined, the update rule space can now also be defined. Unlike other, more complex cellular automata systems, all rules can be described as the rule space is not too large, even with it being unstructured.

f
If s is start point, look at N of the cell. Add 1 to total area counter.
If s is default, change s to area and look at N of the cell. Add 1 to total area counter.
If s is painted, add 1 to total area counter.
If s is area, ignore.
If s is start point and has already been checked, ignore.

These rules are easy to understand and not plentiful, so can be implemented easily programmatically. The decision to include the painted cells, or the boundary of the shape, in the area stems from the idea that the boundary of a shape is infinitely small. Based on the knowledge of objects like Gabriel's Horn (Tran, et al., 2022), which has an infinite surface area but finite volume, it can be extrapolated that any object's boundary can be infinitely large or small, whilst the area remains finite. A demonstration of this can be shown using the formula for a square; $Area = side^2$. As shown in Figure 10, a simple 5 unit by 5 unit square should have an area of 25 units², however if the painted cells are not counted the result is just 9 units², which is incorrect.

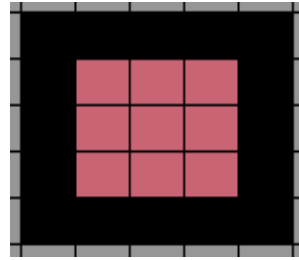


Figure 10

A special rule must also be included to not count the starting position more than once, as otherwise an endless loop may be entered, meaning the algorithm would never end. The algorithm should end when no more cells are able to be checked. Figure 11 shows a demonstration of what the algorithm behaviour is expected to look like for some arbitrary shape over 13 generations.

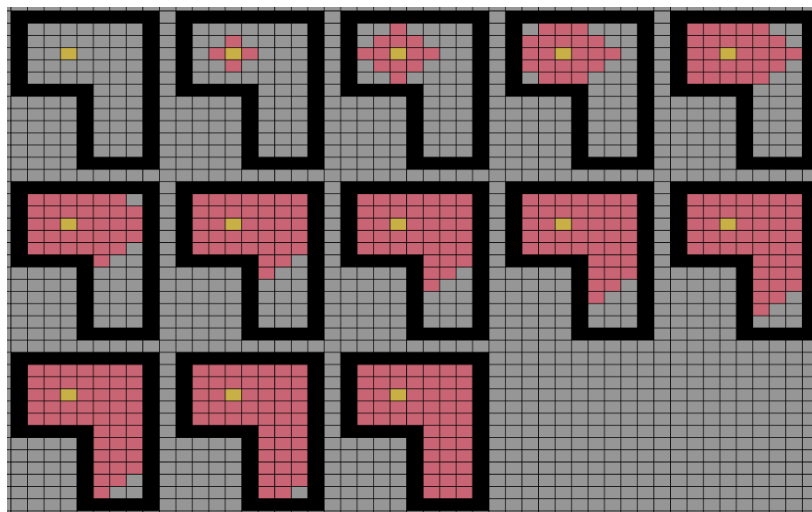


Figure 11

4.3 THE PLANAR GRID

Any 2D shape must exist on some 2D plane of arbitrary dimensions. This can be thought of as a coordinate-agnostic storage device containing the information about each atomic point of the shape.

A Grid class will be used to wrap the underlying data structure – an array of Cell objects. With a one-dimensional (1D) array all necessary operations can be performed to complete the algorithm. Upon constructing the grid, it will validate the arguments as follows:

- **numberOfCells:** The number of cells that make up each row and column must not exceed 256 to prevent lag.
- **cellSize:** The cell size must not be less than 2 as to not become unresponsive.
- $cellSize * numberOfCells$ must not exceed the screen width or height as this will make the window go off screen.

Assuming this validation is successful, the constructor will then set the appropriate visualisation layout and build the grid. Grid will have the following properties, all which have appropriate setter methods and getter methods:

- array **cells:** an array of Cell objects which forms the grid.
- int **numberOfCells:** the number of cells in each row and column.

- Cell **startPoint**: the starting point cell for where the algorithm will begin.
- GridBagConstraints **gridBagConstraints**: arranges cells into a grid formation on-screen, for demonstration purposes only.
- int **mouseButton**: the mouse button that is pressed down. Will be 0 if no button is pressed down.
- CellStates **dragPaintState**: the state every Cell should be set to whilst drag painting with the mouse.

Grid will have several methods to provide the necessary functionality for the algorithm. Heading 4.3.1 to 4.3.3 describe these methods.

4.3.1 buildGrid (int numberOfCells, int cellSize)

This method creates a 1D array of Cell objects using a nested for loop to iterate over each row and column. The 2D grid position of a cell can be flattened into a 1D array index with the following formula:

$$\text{cell index} = ((\text{row index}) * (\text{number of cells in the row})) + (\text{column index})$$

Figure 12 shows the pseudocode for this functionality.

```
function buildGrid(int numberOfCells, int cellSize) {
    // List of Cell objects
    Array[]<Cell> cells = new Array[numberOfCells * numberOfCells]<Cell>;

    // For each row in the grid
    for (int i = 0; i < numberOfCells; i++) {
        // For each column in the grid
        for (int j = 0; j < numberOfCells; j++) {
            // Create a Cell object in each position on the grid
            int cellGridPosition = (i*numberOfCells) + j;
            Cell cell = new Cell(cellGridPosition, cellSize);
            // Add it to cells array
            cells[cellGridPosition] = cell;
        }
    }
}
```

Figure 12

4.3.2 setAllChangedSinceClick (boolean bool)

This method sets the changedSinceClick property for every Cell in the grid to the same value, necessary for drag-painting to work. Figure 13 shows the pseudocode for this functionality.

```
function setAllChangedSinceClick(boolean bool) {
    for (Cell cell) in cells {
        cell.setChangedSinceClick(bool);
    }
}
```

Figure 13

4.3.3 getNeighbouringCells (int id)

This method implements the von Neumann neighbourhood by finding the four cells around a given cell and returning them in an array. The argument id here is synonymous with the grid position or index of the cell. Figure 14 shows the pseudocode for this functionality.

```
function getNeighbouringCells(int id) {
    Array[]<Cell> neighbourhood = new Array[4]<Cell>;

    // Cell to the left (-1,0)
    // If it is the first cell in a row, return null
    if (id % numberOfCells == 0) {
        neighbourhood[0] = null;
    } // Otherwise, return the Cell before
    else {
        neighbourhood[0] = cells[id-1];
    }

    // Cell above (0,+1)
    // If it is the first row, return null
    if (floor(id/numberOfCells) == 0) {
        neighbourhood[1] = null;
    } // Otherwise, return the cell numberOfCells before,
    // mimicking the same position in the previous row
    else {
        neighbourhood[1] = cells[id-numberOfCells];
    }

    // Cell to the right (+1,0)
    // If it is the last cell in a row, return null
    if (id % numberOfCells == numberOfCells-1) {
        neighbourhood[2] = null;
    } // Otherwise, return the next Cell
    else {
        neighbourhood[2] = cells[id+1];
    }

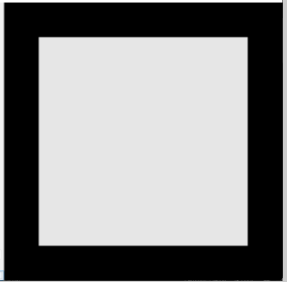
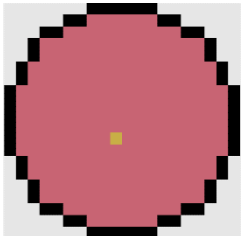
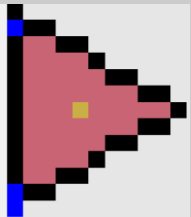
    // Cell below (0,-1)
    // If it is the last row, return null
    if (floor(id/numberOfCells) == floor(cells[-1].id/numberOfCells)) {
        neighbourhood[3] = null;
    } // Otherwise, return the cell numberOfCells after,
    // mimicking the same position in the next row
    else {
        neighbourhood[3] = cells[id+numberOfCells];
    }

    return neighbourhood;
}
```

Figure 14

5 TESTING AND EVALUATION

Test	Expected Outcome	Actual Outcome	Evaluation
An 8x8 square	Area: 64 units ²	Area: 59 units ²	See 5.1

			
A circle with a radius of 10 	Area: 314.2 units ²	Area: 316 units ²	See 5.2
A triangle with base 13 and height 11 	Area: 71.5 units ²	Area: 76 units ²	See 5.3

5.1 AN 8X8 SQUARE

Interestingly, the result of this test turned up 5 cells short of the expected area. Upon testing squares of other sizes, they also resulted in an area five units less than expected (for example the 4x4 square shown in Figure 15 which should have an area of 16 units²).

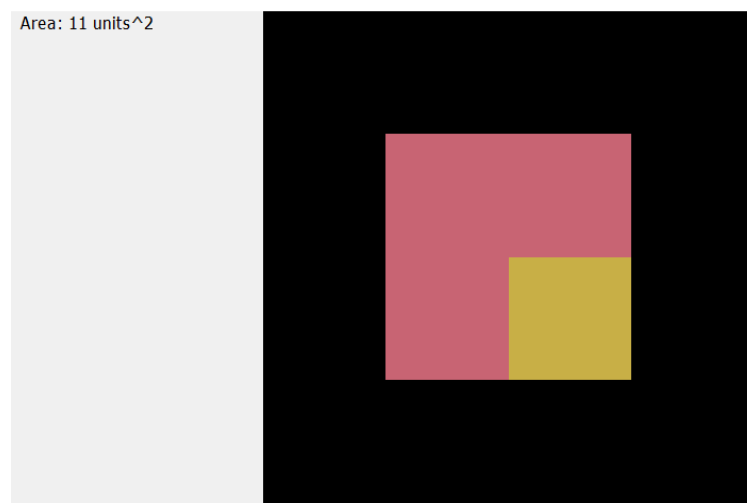


Figure 15

This is caused by the von Neumann neighbourhood approach, as it means shape boundary corners are never checked. Furthermore, the start point cell is not counted either, which in the case of a square totals five cells that are not checked. This can be verified by removing the corners of the square, which as shown in Figure 16, yields the same 11 unit² result. By removing all boundary cells, the algorithm

will simply count all the cells on the grid. This can be used to demonstrate that the starting cell is not counted as instead of a 64 units² result with an 8x8 grid, the area is 63 units² (Figure 17).



Figure 16

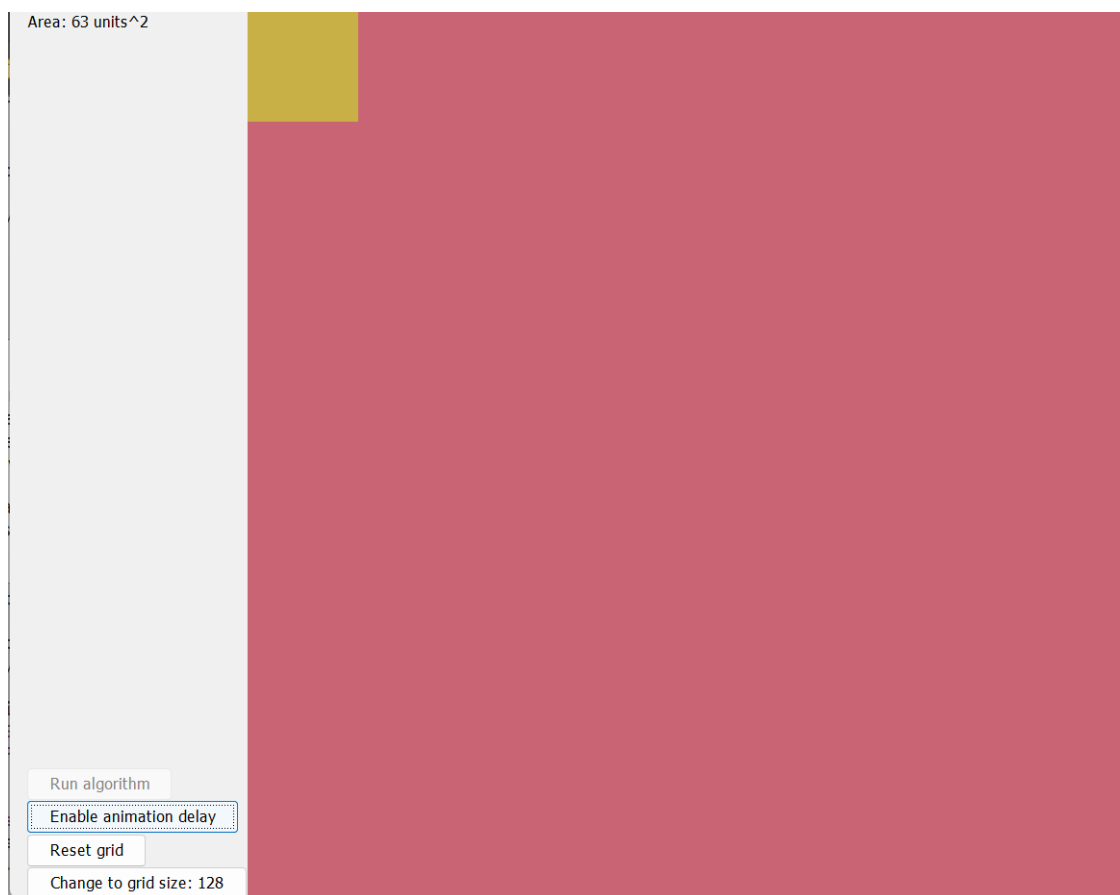


Figure 17

To fix the start position cell not being counted, the algorithm can simply start with an area of 1 rather than 0 to account for the start position. To fix corners not being counted, at the end of the algorithm it can loop over every cell to check for painted cells who have at least one painted neighbour cell that has already been counted, and then count the cell if it meets that criterion (Figure 18).

```
for (Cell cell) in cells {  
    if (cell.state != PAINTED || cell.beenCounted == true) {  
        continue;  
    }  
    Array[]<Cell> neighbours = grid.getNeighbouringCells(cell.id);  
    for (Cell neighbour) in neighbours {  
        if (neighbour.state == PAINTED && neighbour.beenCounted == false) {  
            area = area + 1;  
            cell.beenCounted = true;  
            break;  
        }  
    }  
}
```

Figure 18

5.2 A CIRCLE WITH RADIUS 10

The discrepancy between the expected and actual area is unavoidable due to the atomic nature of the cell being rectangular. As π is irrational, there is no way to entirely represent the area of a circle visually as there will always be some decimal remainder that is unaccounted for. That said, the error of measurement is acceptable so is not an issue.

5.3 A TRIANGLE WITH BASE 13 AND HEIGHT 11

In the same way as the circle, the area will not be exactly accurate due to the cellular nature of the algorithm. One issue that came up though was that in some cases, not every boundary cell would be counted. This occurs when the boundary cell is a corner that is next to another corner boundary cell that is checked later so has not yet been counted (meaning the current cell being checked is not counted). This is a rare edge-case that would require significant modification of the algorithm, so would be better suited for a future iteration of the program.

6 CONCLUSION

The issue of calculating the area of unusual and irregular shapes is one that is very important. Whilst there are well-known methods for finding the area of irregular polygons, a general-purpose method without compromise is very difficult to come by. The 2D shape area generative estimation algorithm proposed allows for a good approximation of any given shape within the limitations of a cellular plane. Making use of a simple cellular automaton system, the algorithm can reliably find an area for any shape provided.

Improvements can be made to the algorithm, though. The use of recursion significantly increases the time and space complexity of the algorithm, whereas other algorithms such as QuickFill (Fleming, 2020) and Span Fill have a significant efficiency advantage. Other optimisations to the current algorithm could be validating neighbours' states before adding them to an ArrayList to decrease the space needed in memory or parallelising the code by using threads to handle the four neighbour directions independently.

Overall, however, the algorithm achieves the three goals it set out to – it is a cohesive and easy to understand algorithm that can work on any given shape and give a good estimation of the shape's area.

7 REFERENCES

- 101 Computing, 2019. *The Shoelace Algorithm*. [Online]
Available at: <https://www.101computing.net/the-shoelace-algorithm/>
[Accessed 16 May 2022].
- Dickson, B., 2020. *Computer vision: Why it's hard to compare AI and human perception*. [Online]
Available at: <https://bdtechtalks.com/2020/08/10/computer-vision-deep-learning-vs-human-perception/>
[Accessed 16 May 2022].
- Fleming, E., 2020. *What is flood fill technique?*. [Online]
Available at: https://www.sidmartinbio.org/what-is-flood-fill-technique/#What_is_flood_fill_technique
[Accessed 18 May 2022].
- Javaheri, M., Blackwell, T., Zimmer, R. & Majid, M., 2015. *Information Gain Measure for Structural Discrimination of Cellular Automata Configurations*, London: University of London.
- Rieke, J., 2017. *Object detection with neural networks — a simple tutorial using keras*. [Online]
Available at: <https://towardsdatascience.com/object-detection-with-neural-networks-a4e2c46b4491>
[Accessed 16 May 2022].
- Sterling, M. J., 2005. *Trigonometry Workbook For Dummies*. 1st ed. s.l.:For Dummies.
- Tran, K., Kau, A. & Khim, J., 2022. *Gabriel's Horn*. [Online]
Available at: <https://brilliant.org/wiki/gabriels-horn/>
[Accessed 16 May 2022].
- Walls, J., 2010. Finding the Area of an Irregularly Shaped Room. *Undergraduate Journal of Mathematical Modeling: One + Two*, 2(2), pp. 3-8.
- Wijeweera, K. R. & Kodituwakku, S. R., 2017. A simple algorithm for calculating the area of an arbitrary polygon. *Ruhuna Journal of Science*, Volume 8, pp. 67-75.

8 APPENDIX

Demonstrations of the program and algorithm:

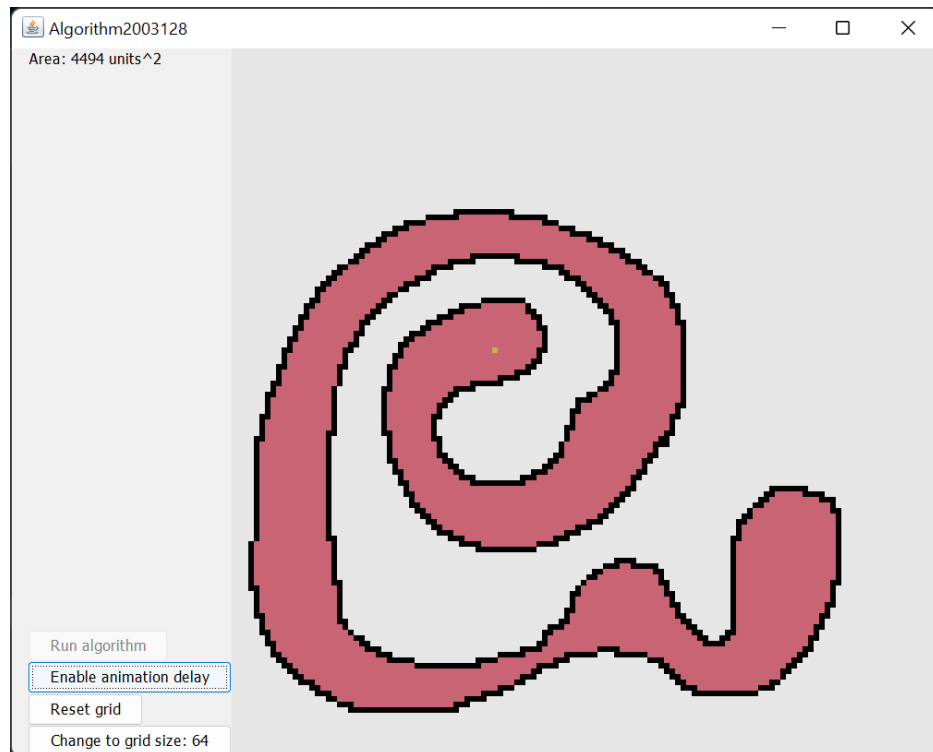


Figure 19

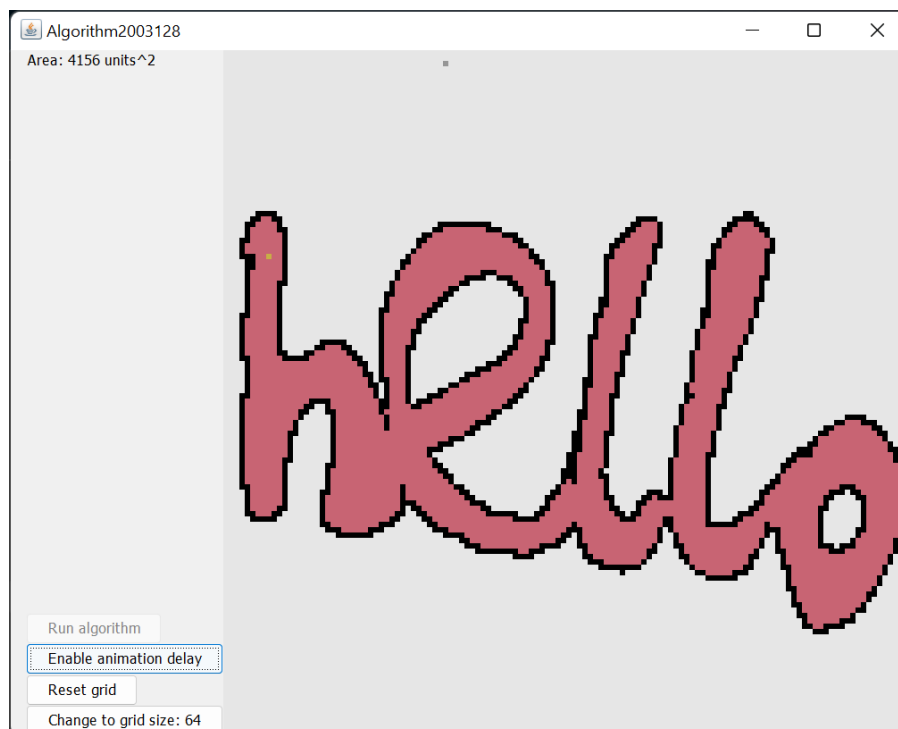


Figure 20

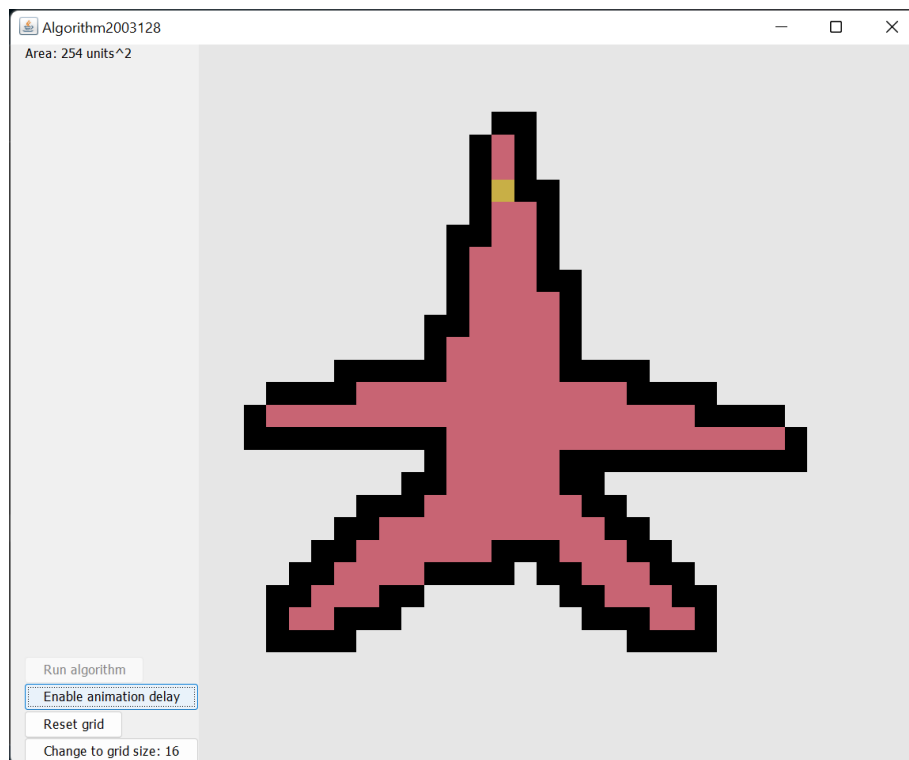


Figure 21

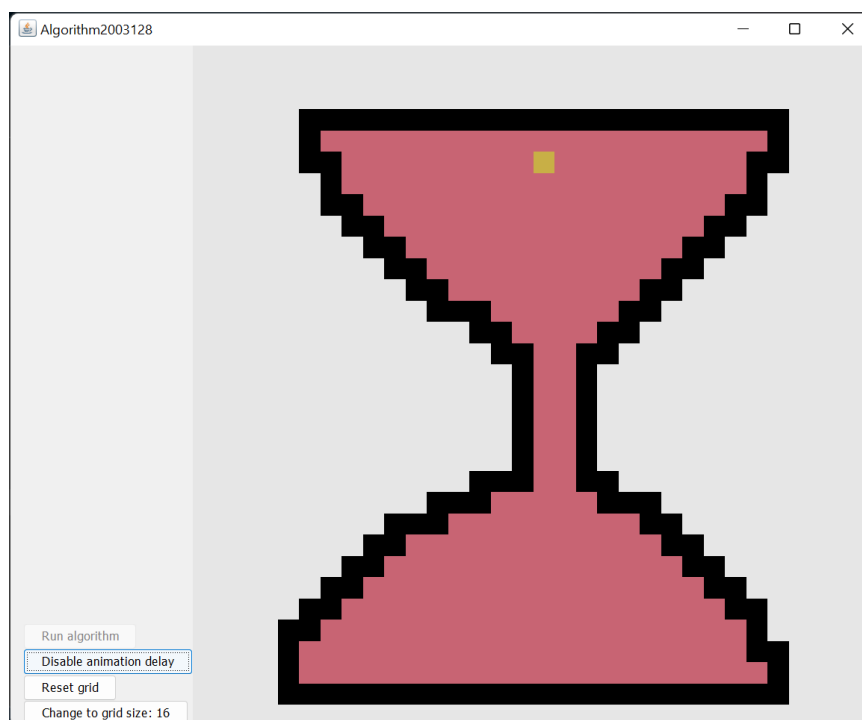
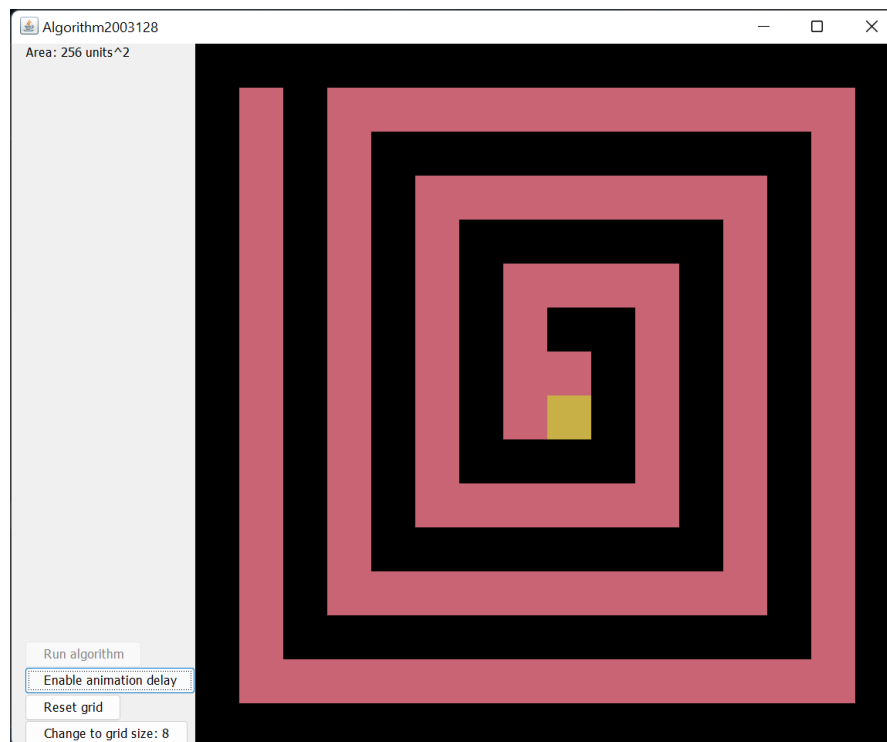


Figure 22

*Figure 23*