# Addendum and Algorithm Explanation

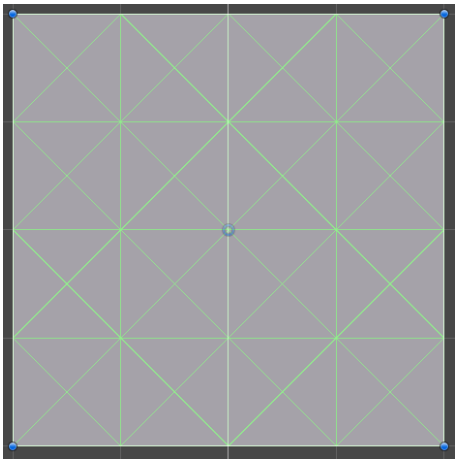
## Difficulty

The three difficulties determine the board size and shape count. Easy, medium and hard have integer values of 0, 1 and 2 respectively. The size is calculated by `difficulty + 4` and the shape count is calculated by `difficulty * 3 + 6`, following table shows the resulting values:

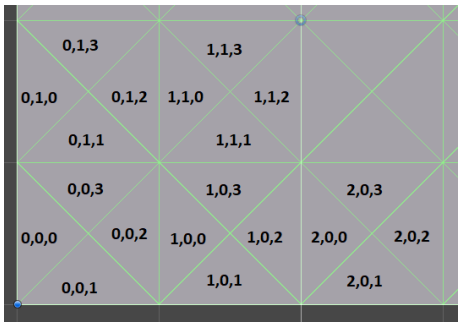
| | Board Size | Shape Count |
|---|---|---|
| Easy | 4 | 6 |
| Medium | 5 | 9 |
| Hard | 6 | 12 |

### Triangle and Shape Generation

4 right Triangles which neighbor each other on the right angle are created for each square in the Board. Which means there are `size * size * 4` triangles in total. Triangles are held in a 3D Triangle array named _triangles[,,]. X and Y values are column and row of the Triangle, and the Z value is which one of the 4 triangles that triangle is in that square. At this point the Board looks like following:



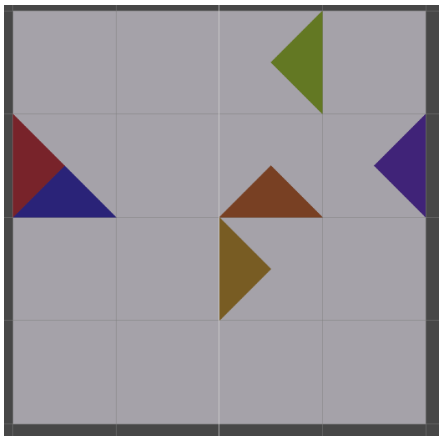
And here is an example of what a portion of triangles would have as coordinates:



In addition to the _triangles[,,] array, all the Triangles are added to the _availableTriangles list. This list holds all the triangles on the Board that don't have an assigned parent yet.

Then **shape count** number of Shapes are created. Shape are just Triangle holders with functions of their own. Each Shape is added to the **_availableShapes** list. This list keeps track of every list that still has a neighboring parentless Triangle that can be added to it. This will help us keep track of which shapes we can pick randomly to add triangles to.

Each Shape is given a random Triangle to start things up. Whenever a Triangle is added to a Shape, it is given the Shape's color. At this stage the Board looks like this:



Each time a Triangle is added to a Shape, The Board finds the neighbors of the added Triangles which don't have any parents and adds these to the Shape's possibleTriangles list. This means each Shape keeps track of which Triangles they can expand into. When a Triangle is added to a Shape, additionally, every Shape in the Board is looped through and we check if any of the Shapes have that Triangle in their possibleTriangles list. If so, we remove the Triangle from the list. What this does is to ensure that no Shape in the Board continues considering a Triangle to expand into which has just been added to a Shape.

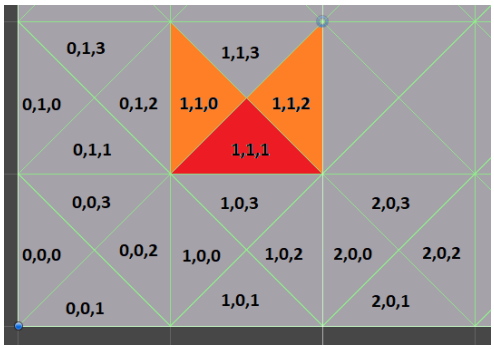
Then, while there are Triangles in the _availableTriangles list, we choose a random Shape from the _availableShapes list, pick a random Triangle from its _possibleTriangles list and add it to the shape. Again, we consider neighboring Triangles and add them to the Shape's _possibleTriangles list and make sure the Triangle that is just added is removed from every Shape's _possibleTriangles lists.

After each time we take out a Triangle from a Shape's _possibleTriangles list, we check if there are any Triangles left in the list. If there are no Triangles left, it means the Shape has grown to its fullest size and is surrounded by walls and other Shapes. If this is the case, we remove the Shape from our _availableShapes list, so that when we randomly choose a next Shape from it, we don't choose a completed Shape.

## Finding a Triangle's Neighbors

I've been talking about adding neighboring Triangles to a Shape's _possibleTriangles list. Now I will talk about how I get to find the neighboring Triangles of a Triangle with my custom coordinate system.

Since a Triangle has 3 edges it has 3 neighboring Triangles (with the exception of Triangles on the edge of the Board which will have 2 Triangles). The first two Triangles to consider are the ones that share the same square as our original Triangle. Here is how it looks like:




Wherever a Triangle may be, we can find these two Triangles by adding 1 and 3 the Z value of the Triangle's Coordinate system and moding it. Here is the code that finds these two:

```
AddPossibleTriangleToShape(shape, _triangleGrid[x, y, (z + 1) % 4]);
AddPossibleTriangleToShape(shape, _triangleGrid[x, y, (z + 3) % 4]);
```

The third neighbor is trickier. First, I've calculated this number for each of the 4 Triangle in a box. Here are the formulas:

| Original x | Original y | Original z | | Neighbor x | Neighbor y | Neighbor z |
|---|---|---|---|---|---|---|
| i | j | 0 | | i - 1 | j | 2 |
| i | j | 1 | | i | j - 1 | 3 |
| i | j | 2 | | i + 1 | j | 0 |
| i | j | 3 | | i | j + 1 | 1 |

To turn this into a single formula we need to do translations on all 3 dimensions. For x and y we will translate z and add it to them, finding Z is easier, it's a single translation.

$x = x + ( ( z - 1 ) \% 2 )$

$y = y + ( ( z - 2 ) \% 2 )$

$z = ( z + 2 ) \% 4$

In addition, as I've explained before a Triangle may not have this third neighbor. So first we check if the x and y coordinates allow for such a Triangle to exist, if they do, we also add this third Triangle. This is how it looks like in code:

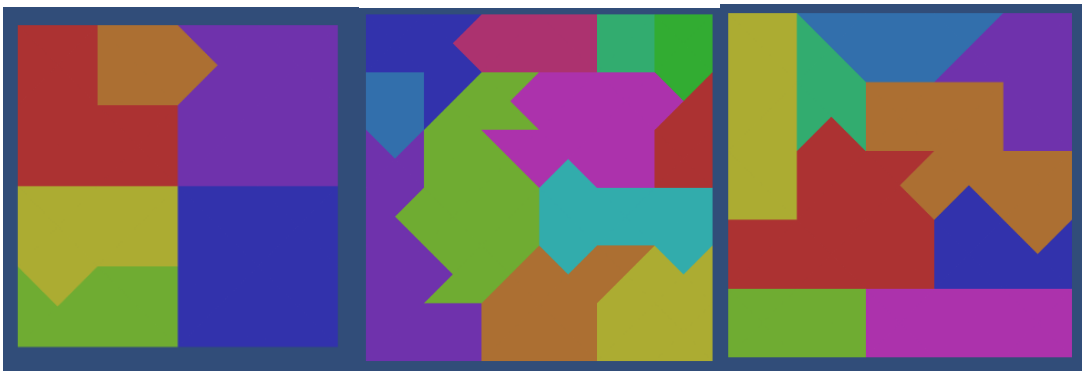
```
x += (z - 1) % 2;
y += (z - 2) % 2;

if (x > -1 && x < lookup.size && y > -1 && y < lookup.size)
{
    AddPossibleTriangleToShape(shape, _triangleGrid[x, y, (z + 2) % 4]);
}
```

## Checking for Puzzle Completion

### Position Check Algorithm:

When we are done looping through the Triangles and adding them to the Shapes, they are positioned where they are supposed to be to end the puzzle. So, before dislocating them elsewhere for the player to find them we save their positions. When the player is done moving a Shape, the Shape checks if it is where it is supposed to be. If so, it calls on the GameMaster to check if the other Shapes are also correctly placed. If this is the case the puzzle is completed and a new one is generated.

The problem with this method is that even though it is efficient, there might be puzzles where there are multiple solutions. If the player solved the puzzle with a solution other than how the Shapes were initially located, the GameMaster will refuse to load the next puzzle, thinking the placement is wrong. Here are some examples of "unfinished" puzzles using this algorithm:



So, even though I think it might be much more expensive, I have decided to try the following method:

### RayCasting Algorithm:

Since there are exactly the number of Triangles as the number of Triangle slots, for the completion, everywhere in the Board should be under a single layer of triangles. After a Shape is dropped, we start raycasting on every Triangle slot center, in total size * size * 4 times. If at any point the raycast returns an array of size other than 1, we stop raycasting since it means the puzzle is not finished yet. Though this method is more expensive, it stops raycasting once a single raycast returns a false value.

With this algorithm we ensure that no matter how the player solves the puzzle the GameMaster will accept it and generate a new puzzle. I was not sure about how much more expensive this one is compared to the other, so I played 10 games in each difficulty with each algorithm and measured the ticks passed using TimeSpan, between user dropping a Shape and GameMaster finishing checking for puzzle completion. Here are the averages:

| Average Tick (n=10) | Position Check Algorithm | Raycasting Algorithm |
|---|---|---|
| Easy | 1526 | 2070 |
| Medium | 1450 | 1571 |
| Hard | 1374 | 1876 |

So, raycasting wasn’t as expensive as I thought it would be. The reason the first algorithm takes less time in average the more difficult a puzzle is, is that it doesn’t run if the manipulated Shape is not placed correctly. Since you tend to move Shapes around more when it is more difficult, this makes the average go down. The reason Raycasting seems to have random results is that the algorithm starts from the bottom left corner to raycast, so if you happen to start placing Shapes from to bottom left the algorithm will take more time to finish, which in the chaos and randomness of the puzzles and how the player solves them, makes the results seem random.

After seeing the results, I couldn’t decide which one to take out and which one to keep, so I added an option in the Scriptable Object to switch between the two.

Additionally, there is this weird bug in the Raycasting algorithm that I couldn’t figure out. Sometimes when you finish a puzzle it doesn’t accept it and you need to just click on a random Shape and it gets that the puzzle is finished. If somebody can help me figure out why this happens I would really appreciate that.

## Save Load System

I made a class called PuzzleReadWrite to deal with read and write. It has the size, shape count and shapes list as members and when the GameMaster tells it to read from a while it saves the data in its members.

### Save

It first saves size and shape count in two lines, then it loops through the shapes list and for each Shape it writes “shape” to a line, followed with the coordinated of each Triangle that belongs to that Shape. Triangle coordinates are converted to comma separated strings and is written as such.

### Load

It reads the size and shape count and loads them into its variables. Afterwards, in a while loop it reads the lines one by one. When it reads the string “shape”, it saves the last read Shape and creates a new empty list of coordinates. When it reads a line other than “shape” it separates it by commas and converts the three strings into a coordinate and adds it to the last generated list. When it encounters “shape” again or when it reached the EOF it saves the last read Shape again. So, at the end it ends up with a list of lists of coordinates which can easily be used by the Board to assign Triangles to Shapes.

## Additional Info

- The Triangles are the ones that have colliders, so when you click and drag them around, they tell their parent Shape to move around
- I added a score system. Each puzzle you start with a score of shape count * 2 and with every move it decrements. The score you end up with when you finish the puzzle is added to your total score, which means if you play it perfectly you get shape count amount of score from a puzzle
- I added a hint button. When you click it, it decreases your score a little bit and snaps a random Shape into its correct position, paints it white and locks it. This can be used once per puzzle
- After a certain y value, the dropped Shapes start snapping to the grid.
- I created a Helper class to divide the methods of the Board class since it was getting too crowded