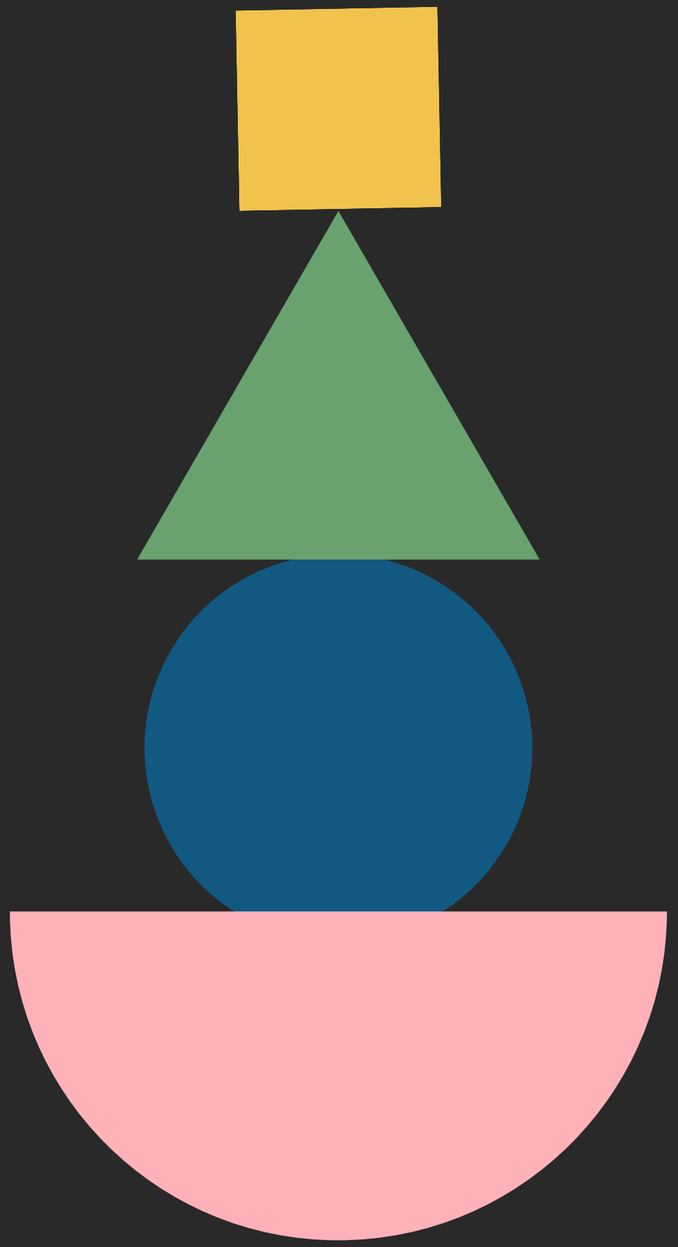


USING PATHFINDER FOR NAVAGATION

MADE BY:
MARKO ILIOSKI

THE PROBLEM

The premise:

Make a program for drone navigation through a environment.

The input:

- Grid matrix of size n×m;
  - Start position;
  - End position;
  - Obstacles ;
- Delivery price;
- Power cost;
- Repair cost;
- Discount (gamma value);

The output:

Optimal path that the drone should follow from the start to the end while considering the obstacles and optimizing utility based on the provided parameters.

THE TEHNOLOGY

Originally in python:

- **Pros;**
  - Fast to code;
  - Fast to execute;
  - Easy to understand;
  - The right tool for the job;
- **Cons:**
  - Ugly;
  - Not as portable as it could be;

Remade in javascript:

- **Pros;**
  - Looks **good**;
  - Easy enough;
  - Portable (it's a web-site)
  - Tools for converting from python
- **Cons:**
  - Slower then python;
  - Not the right tool for the job;

THE SOLUTION

1. The Input
    a. Constants
    b. Grid
2. The Calculation
    a. Step
    b. Converge
3. The Path

THE INPUT

CONSTANTS

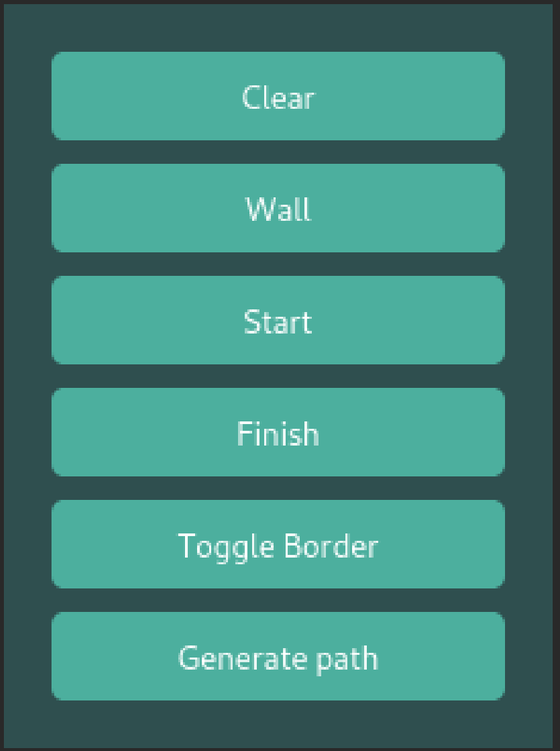
Global variables for convenience.
Dynamic input.

```
// Function to get values from input fields
function updateValues() {
  ROWS = parseInt(document.getElementById("rows").value);
  COLUMNS = parseInt(document.getElementById("cols").value);
  deliveryPrice = parseFloat(document.getElementById("deliveryPrice").value);
  powerCost = parseFloat(document.getElementById("powerCost").value);
  repairCost = parseFloat(document.getElementById("repairCost").value);
  discount = parseFloat(document.getElementById("discount").value);
  generateGrid();
}
```

THE INPUT

1.b

GRID



Select draw mode.
Draw on a interactive grid.
Change the resolution of the grid.

THE CALCULATION

2.C STEP

```
// Calculate the utility at all neighboring positons while using no speed boost
// direction_probability * (-1 * powerCost + (discount * values[next[y]][next[x]]))
let s =
  0.7 * (-1 * powerCost + discount * values[s_posn[0]][s_posn[1]]) +
  0.15 * (-1 * powerCost + discount * values[w_posn[0]][w_posn[1]]) +
  0.15 * (-1 * powerCost + discount * values[e_posn[0]][e_posn[1]]);

let w =
  0.7 * (-1 * powerCost + discount * values[w_posn[0]][w_posn[1]]) +
  0.15 * (-1 * powerCost + discount * values[n_posn[0]][n_posn[1]]) +
  0.15 * (-1 * powerCost + discount * values[s_posn[0]][s_posn[1]]);

let n =
  0.7 * (-1 * powerCost + discount * values[n_posn[0]][n_posn[1]]) +
  0.15 * (-1 * powerCost + discount * values[w_posn[0]][w_posn[1]]) +
  0.15 * (-1 * powerCost + discount * values[e_posn[0]][e_posn[1]]);

let e =
  0.7 * (-1 * powerCost + discount * values[e_posn[0]][e_posn[1]]) +
  0.15 * (-1 * powerCost + discount * values[n_posn[0]][n_posn[1]]) +
  0.15 * (-1 * powerCost + discount * values[s_posn[0]][s_posn[1]]);
```

Only 4 directions.
Value for error.

The value with the biggest value is the **BEST** move.

THE CALCULATION

CONVERGE

```
// Loop Until Board Converges
while (true) {
  for (let i = 0; i < ROWS; i++) {
    for (let j = 0; j < COLUMNS; j++) {
      let currPosn = [i, j];
      // Posible error here
      if (
        (currPosn[0] !== goal[0] || currPosn[1] !== goal[1]) &&
        !rivals.some(
          (pos) => currPosn[0] === pos[0] && currPosn[1] === pos[1]
        )
      ) {
        calcNextMoves(currPosn, values, policies);
      }
    }
  }
  if (converges(prev, values)) {
    // Stop looping and return if board converges
    break;
  } else {
    // Else, continue looping and keep track of previous values nxn matrix
    prev = copyValues(values);
  }
}
return values[start[0]][start[1]]; // Returns utility values at the start position
```

Run the calculations for each cell.

This is looped forever untill it converges.

THE PATH

```
let currentRow = start[0];
let currentCol = start[1];

while (policies[currentRow][currentCol] !== 0) {
  // Move to the next cell based on the policy
  switch (policies[currentRow][currentCol]) {
    case 1:
      currentCol++; // Move right
      grid[currentRow][currentCol] = CELL_VALUES.PATH;
      break;
    case 2:
      currentRow--; // Move up
      grid[currentRow][currentCol] = CELL_VALUES.PATH;
      break;
    case 3:
      currentCol--; // Move left
      grid[currentRow][currentCol] = CELL_VALUES.PATH;
      break;
    case 4:
      currentRow++; // Move down
      grid[currentRow][currentCol] = CELL_VALUES.PATH;
      break;
    default:
      // Invalid policy
      return null;
  }
```

Transform the policies from directions to path.

Direction from each block but we only need to calculate it from the start.