

Assignment

Stage Two Submission

2805ICT/3815ICT/7805ICT

Student name: Yasin Cakar	Student ID: S2921450	Enrolled Course Code: 3815ICT
Student name: David Todorovic	Student ID: S5167246	Enrolled Course Code: 3815ICT
Student name: Jerry Li	Student ID: S2928643	Enrolled Course Code: 3815ICT
Student name: Mohammad Mari	Student ID: S5185052	Enrolled Course Code: 7805ICT

Table of Contents

Table of Contents.....	2
1.0 Project Planning and Documentation.....	3
1.1 Time Schedule.....	3
1.2 Total working hours.....	4
1.3 Effort and contribution table.....	4
1.4 Automatic Documentation.....	5
2.0 Advanced Design.....	7
2.1 MVC architectural design pattern.....	7
2.2 Design pattern 1.....	8
2.3 Design pattern 2.....	10
2.4 Design tactic.....	12
2.5 Random maze generation.....	13
2.6 Path search algorithms.....	19
3.0 Testing.....	22
3.1 Software test description.....	22
3.2 Software test report.....	29
4.0 Reflection.....	33
5.0 Video link.....	34
6.0 Appendices.....	34

1.0 Project Planning and Documentation

1.1 Time Schedule

This table should reflect who did what, how long you expected sections to take and the actual hours it took to perform the tasks.

Task		Plan				Actual		
#	Task Name	Student	Planed Time	Cumulative Time	Finished Date	Time	Cumulative Time	Finished Date
1	Project plan	Mohammad Mari Yasin Çakar DavidTodorovic Jerry Li	2 hours		6 Sep	5 hours		10 Sep
2	2.1 MVC architectural design pattern	Mohammad Mari	4 hours		13 Sep	6 hours		17 Sep
2	2.2 Design pattern 1	Mohammad Mari	4 hours		13 Sep	6 hours		20 Sep
2	2.3 Design pattern 2	Mohammad Mari Yasin Çakar	1 hours		20 Sep	6 hours		23 Sep
2	2.4 Design tactic	Mohammad Mari	2 hours		20 Sep	4 hours		23 Sep
2	2.5 Random maze generation	Yasin Çakar Jerry Li David Todorovic	6 hours		22 Sep	11 hours		24 Sep
2	Path search algorithms	Jerry Li	5 hours		13 Sep	10 hours		17 Sep
3	Software test description	Mohammad Mari Yasin Çakar DavidTodorovic Jerry Li	1 hours		24 Sep	4 hours		25 Sep
	Software test	Yasin Çakar	2		27 Sep	2		30 Sep

	report	David Todorovic Jerry Li	hours			hours		
	Reflection	Mohammad Mari	0.25 hour		7 Oct	1 hour		8 Oct
	Video link	David Todorovic Jerry Li	3 hours		9 Oct	8 hours		11 Oct

1.2 Total working hours

Student Name (#ID)	Plan (hours)	Actual (hours)
Yasin Çakar	12	32
Jerry Li	14	35
Mohammad Mari	14.25	28
David Todorovic	11	25
Total working hours	102.5	120
Average working hours per person	25.63	30

1.3 Effort and contribution table

Student	Effort Level* (Rating from 0 – 5, the information is filled by the group)	Contribution Level* (Rating from 0 – 5, the information is filled by the group)	Justification If a student received level rating of 3 or less, your group need to give explanation for the low level rating
Yasin Çakar	5	5	N/A
Jerry Li	5	5	N/A
Mohammad Mari	5	5	N/A
David Todorovic	5	5	N/A

- *Level ratings, 5 = excellent, 4 = good, 3 = reasonable, 2 = poor, 1 = unacceptable, 0 = none

1.4 Automatic Documentation

[Your group needs to use an auto documentation generator (such as doxygen or javadoc) to generate documents of the source code. Please use screenshots to demonstrate that a suitable document generator has been applied in developing this project.]

We have chosen the JSDoc as our auto documentation generator, JSDoc is a tool that generates documents for JavaScript applications, libraries, and modules based on the annotation information in the JavaScript file.

Class: Ghost

Ghost(ghost_index, x, y, w, h, speed, is_scared, scared_time, color)

new Ghost(ghost_index, x, y, w, h, speed, is_scared, scared_time, color)

This is the Ghost Class

Parameters:

Name	Type	Description
ghost_index	array	the ghost list
x	int	x coordinate
y	int	y coordinate
w	Int	the width of the ghost
h	Int	the height of the ghost
speed	Int	ghost movement speed
is_scared	Boolean	check if the ghost is scared
scared_time	Float	scared timer
color	String	The color of the ghost

Source: [ghost.js, line 15](#)

Members

corner

used when ghosts run to their corner instead of pacman

Source: [ghost.js, line 27](#)

current_cell

the current position of the ghost

Source: [ghost.js, line 40](#)

img_indx

the horizontal image of the ghost

Source: [ghost.js, line 46](#)

isEaten

check if the ghost is eaten by pacman

Source: [ghost.js, line 19](#)

Methods

chooseNextDirection()

This is the controller for next ghost movement

Source: [ghost.js, line 99](#)

Returns:

possible cells for move action

Home

Classes

Ghost
HUD
Level
Pacman
Pellet
PowerPellet
randomMazeSetup
Wall

Global

addCommas
boom_sound
button
button1
button2
button3
button4
cell
checkNeighbours
congratulations_sound
current
draw
exitGame
font
gameover
ghost_img
ghosts
grid
hud
intro_music
keyPressed
level
maze
model
mode2
newLevel
normalMode
pacman_chomp_sound
pacman_death_sound
pacman_eatghost_sound
pacmans
power_pellet_sound
preload
randomMode
setup

chooseRandom(possible_cells)

This is the controller for the Ghost to Move Randomly

Parameters:

Name	Type	Description
possible_cells	Array	path coordinate

Source: [ghost.js, line 215](#)

Returns:

Next cell position for the Ghost movement

die()

This is the function that is called when the Ghost is eaten by the pacman

Source: [ghost.js, line 267](#)

followTarget(pacman_index, possible_cells)

This is the controller for the Ghost to chase the pacman

Parameters:

Name	Type	Description
pacman_index	Array	Pac-Man position
possible_cells	Array	Path coordinate

Source: [ghost.js, line 192](#)

Returns:

distance between Pac-Man and Ghost

getScared()

Gost reaction when Power Pellet is activated

Source: [ghost.js, line 234](#)

move(cellX, cellY)

This function is the ghost Movement controller

Parameters:

Name	Type	Description
cellX	Int	X coordinate
cellY	Int	Y coordinate

Source: [ghost.js, line 75](#)

restart()

Ghost respawn

Source: [ghost.js, line 281](#)

show()

This function is to show the ghosts

Source: [ghost.js, line 51](#)

switchImage()

Ghost texture

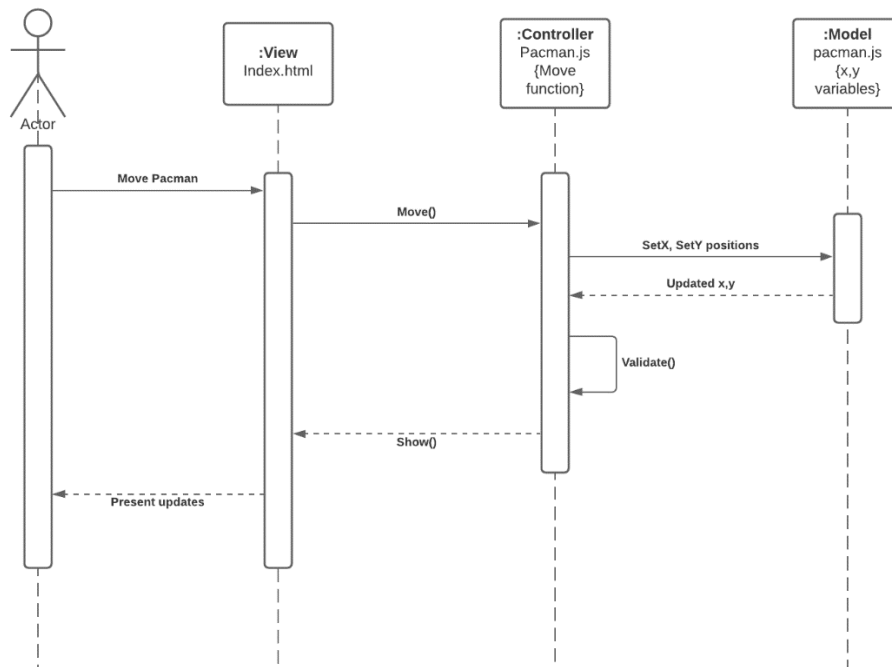
Source: [ghost.js, line 226](#)

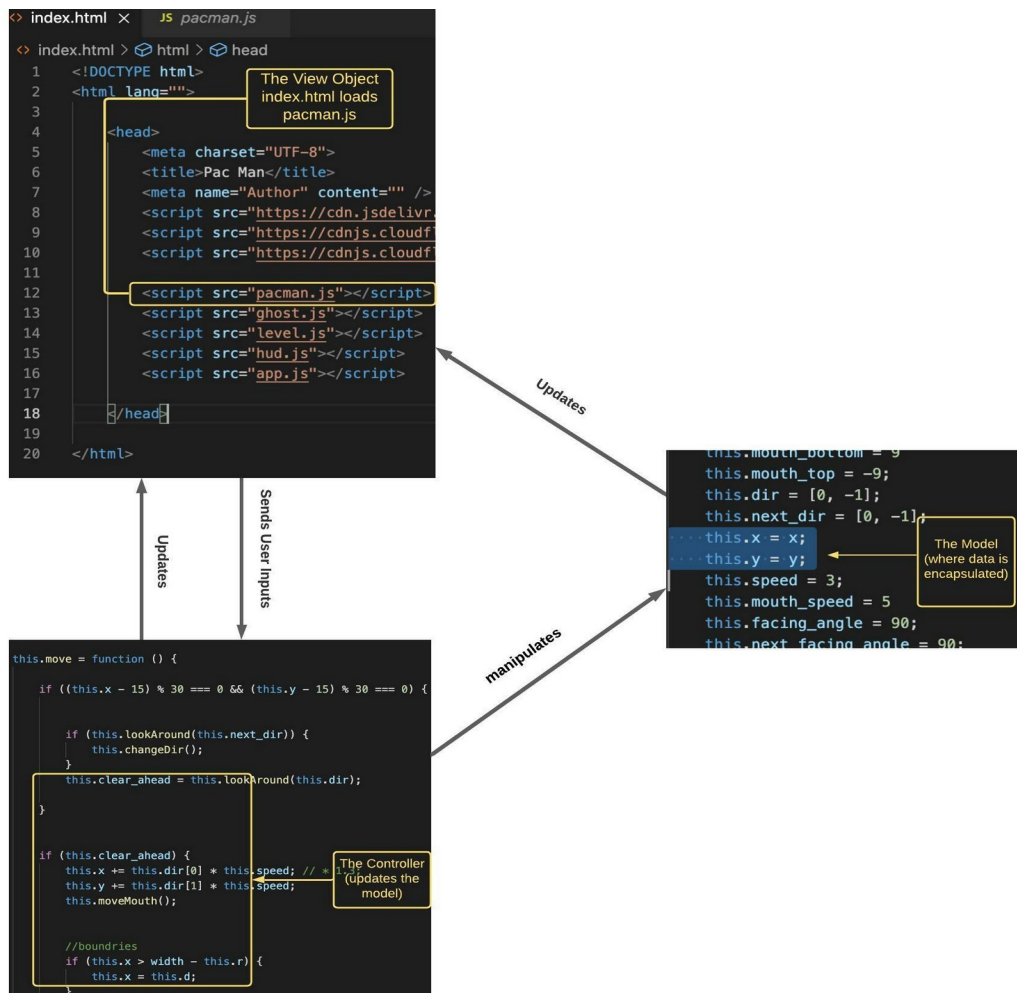
2.0 Advanced Design

2.1 MVC architectural design pattern

Model / view / controller (MVC) is a software architecture design pattern consist of three objects. The “Model” which corresponds to the data, the “View” which corresponds to the screen presentation and the “Controller” which corresponds to the user’s interface reaction to the user inputs. The intent of the design is to decouple object so that changes to one object can affect others without the need to know the implementation details of others.

The following sequence diagram shows how pacman “move” is represented in MVC pattern, basically pacman game is visually presented to the user via web browser (**The View Object**). The user is able to perform basic I/O command through the view such as moving packman. The user interface reacts to it and calls the move() method (**The Controller Object**) which updates x and y variables (**The Model component**)

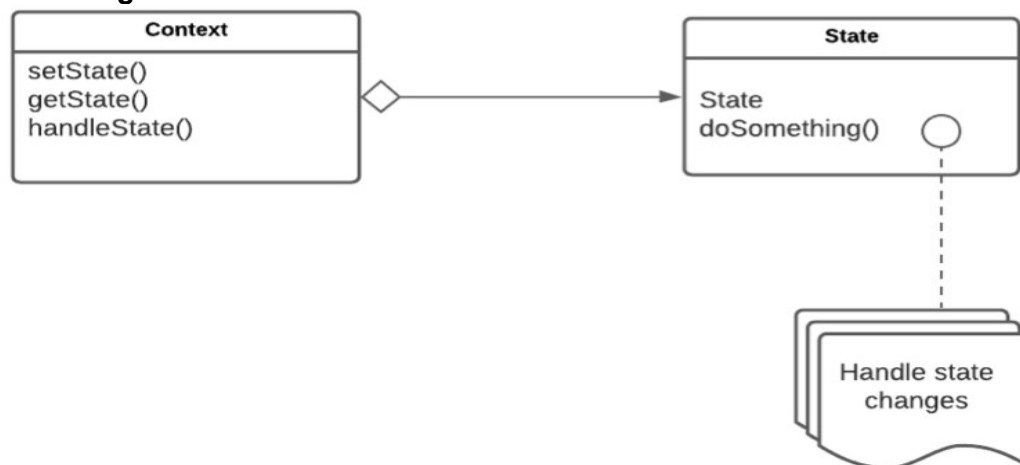




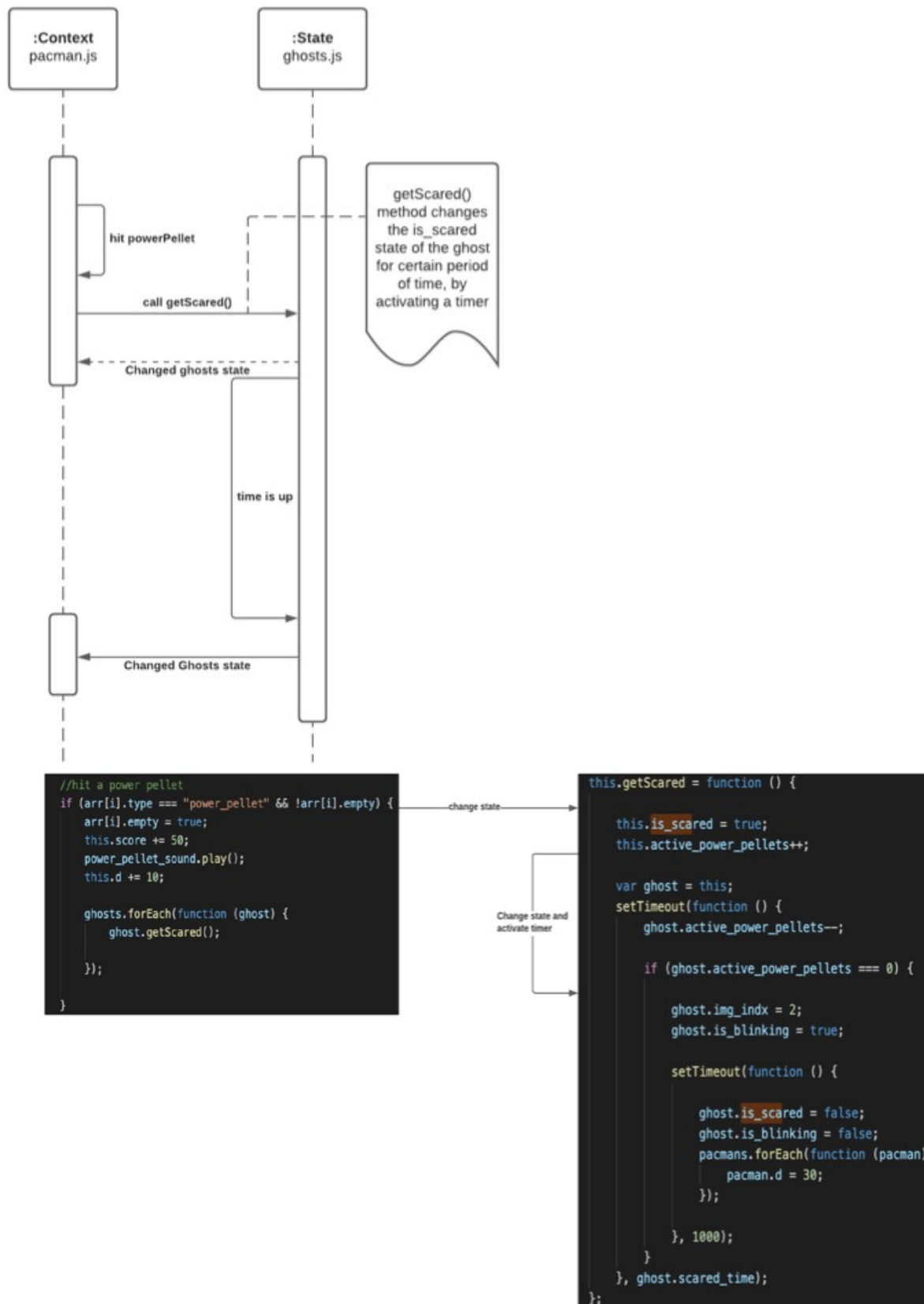
2.2 Design pattern 1

State design pattern has been applied to the ghost object. The ghosts go through two different states (ghosts chasing packman & ghosts scared of pacman). The ghost object can change its behaviour at runtime based on its state.

UML Diagram:



How it works (implementation):



2.3 Design pattern 2

Iterator design pattern has been applied to the random maze generation feature. Iterator is a design pattern that lets you iterate through elements of a collection without exposing its underlying implantation.

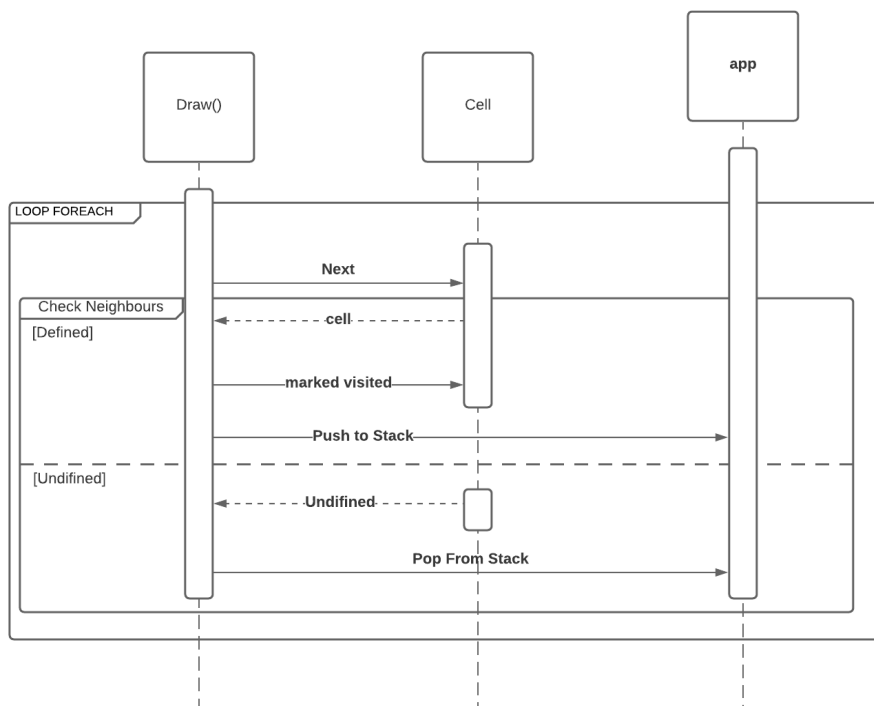
The following diagram illustrates how the iterator design pattern used in the implementation of the recursive backtracker algorithm.

The algorithm does a depth first search where it iterates the maze grid where each grid is a cell object in the code and initially marked as unvisited, the draw function traverses the maze grid iterating through each unvisited cell and making each newly visited cell as current.

Each current cell is subsequently marked as visited, each of these cells are pushed onto a stack to be called upon when there is no neighbouring cell left to be randomly selected as the current cell.

When a current cell finds that it has no unvisited neighbours for the path traversed, it backtracks the path covered that is determined by a sequence, that is the cell objects popped from the stack until it finds an unvisited cell object along the backtracked path traversed.

When an unvisited neighbour along the path backtracked is found the depth-first search backtracking logic is repeated until all cells on the grid are marked as visited.



Code snippet:

Draw object:

```
// STEP 1 - Choose randomly one of the visited
var next = current.checkNeighbours(); //Check
random unvisited one and return it.
if(next){
    next.visited = true;

    // STEP 2
    stack.push(current);

    // STEP 3
    removeWalls(current,next);

    // STEP 4 - Make the chosen cell the current
visited.
    current = next;
} else if (!(stack.length == 0)) {
    var aCell = stack.pop();
```

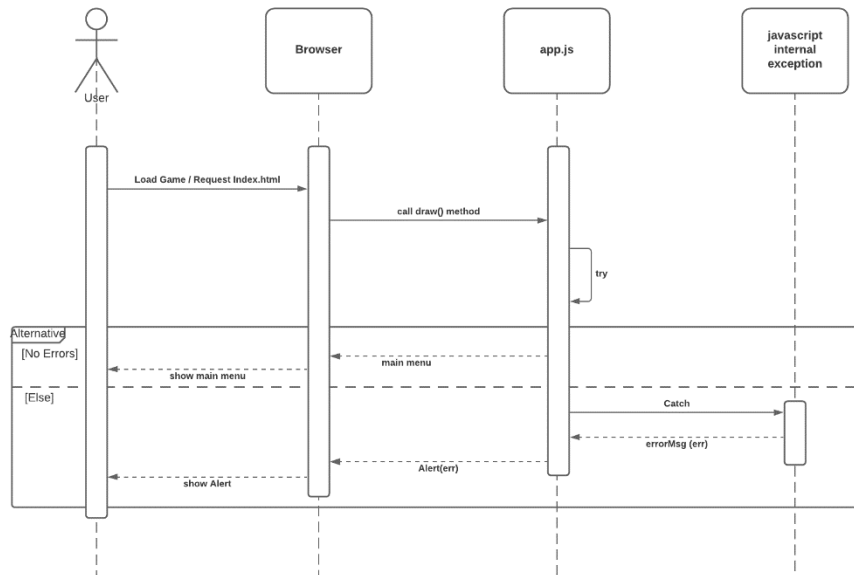
Cell object:

```
// Are there neighbours that have not been visited, if so select one of
those
if(neighbours.length > 0){
    var rand = floor(random(0,neighbours.length))// Pick a random
neighbour randomly
    return neighbours[rand]; // return a randomly selected neighbour
cell
} else {
    return undefined;
}
```

2.4 Design tactic

One of the design tactics that we implemented in our software is “**Fault detection**” using exceptions. “try/catch” statements have been used to enable us to detect faults that we may encounter in our software, consequently, improved software operational efficiency and reduced troubleshooting time

Design diagram:



Implementation:

The following screenshot illustrate the use of “try/catch” in the draw() function that creates the main menu

```

function draw() {
  //Main Menu
  try {
    if (mode == 0) {
      image(logo, 410, 315, 450, 300);
      image(logo1, 400, 100, 300, 120);
      fill("White");
      textSize(35);
      text("2021", 260, 160);
      textSize(25);
      text("3815ICT / 7805ICT", 300, 205);
      textSize(15);
      text("Jerry Li", 5, 700);
      text("Yasin Cakar", 5, 730);
      text("Mohammad Mari", 5, 760);
      text("David Todorovic", 5, 790);
      // Play button
      button = createImg('images/play.png');
      button.position(325, 475);
      button.size(175, 75);
      button.mousePressed(mode1);
      // Configure button
      button2 = createImg('images/configure.png');
      button2.position(323, 580);
      button2.size(185, 85);
      button2.mousePressed(mode2);
      // Exit button
      button1 = createImg('images/exit.png');
      button1.position(325, 700);
      button1.size(175, 75);
      button1.mousePressed(exitGame);
    }
  } catch (err) {
    alert(err)
  }
}

```

Triggering example (fault injection):

Created fault by calling non-existence method

Catch Operation / Error Message

```

text("Jerry Li", 5, 700);
text("Yasin Cakar", 5, 730);
text("Mohammad Mari", 5, 760);
text("David Todorovic", 5, 790);
// Play button
button = createImgggg('images/play.png');
button.position(325, 475);

```

ReferenceError: Can't find variable: createlmggggg

Close

2.5 Random maze generation

The random maze generation technique used for the assignment is the **Recursive backtracker** or technically the *depth-first search recursive backtracker maze generator*. The pseudo code for this implementation is as follows:

1. Make the initial cell the current cell and mark it visited.
2. While there are unvisited cells
 1. If the current cell has any neighbors which have not been visited
 1. Choose randomly one of the unvisited neighbors
 2. Push the current cell to the stack.
 3. Remove the wall between the current cell and the chosen cell.
 4. Make the chosen cell the current cell and mark it as visited.
 2. Else if stack is not empty
 1. Pop a cell from the stack.
 2. Make it the current cell

First of all we concept is to implement a grid on a canvas, each grid cell on the canvas in code is to be a "Cell" object. The program will figure what wall it needs to figure to make a maze pattern.

First of all the cell object needs to know where it is in the grid, that is which column and row, as a coordinate. There needs to be a data structure where each cell knows about each of its walls if it is open or closed.

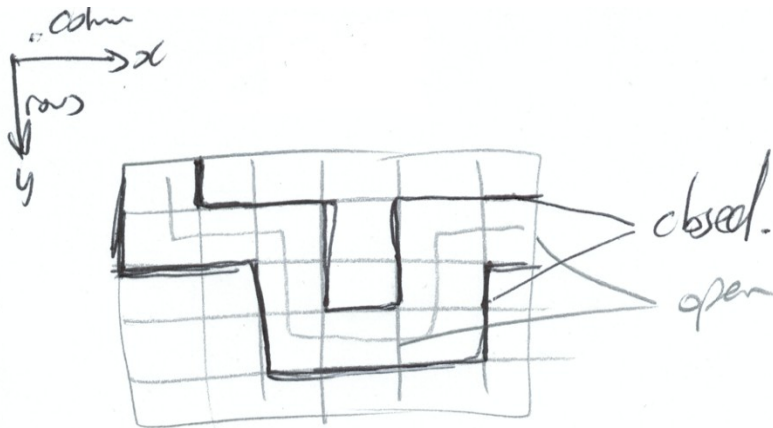


Figure 1 - Concept of Depth first search

In this section the development of this algorithm will be explained step by step.

The first step is to draw the maze grid where the forward and backtracking algorithm will be applied. (See [Appendix A](#))

Now that there is a grid, each cell wall needs to be represented as individual lines. Now each point in the cell grid is represented with respect to its coordinates and the width of the square as shown in the image below:

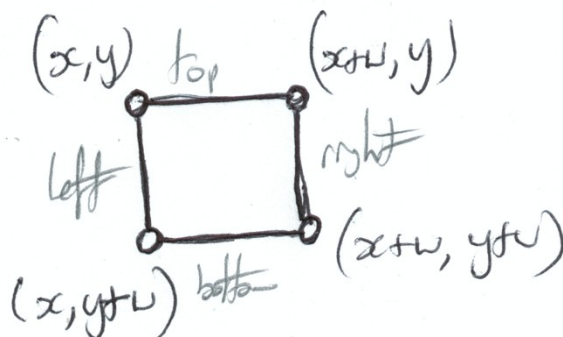


Figure 2 Illustration of how the maze generator will draw walls using coordinates.

So now we replace the line `rect(x,y,w,w)` with `line(x,y,x+w,y),line(x+w,y,x+w,y+w),line(x+w,y+w,x,y+w),line(x,y+w,x,y);` (See [Appendix B](#))

Next, additional functionality was added for the code to determine if it is to have a wall on it top, right, bottom and left. The method decided was the create an array map attribute for each cell object that determined using Boolean values whether it had a wall or not with true meaning to place a wall and false to not have a wall for the side it mapped to. The line added was:

```
this.walls = [true,true,true,true]; //[top,right,bottom,left]
```

Indexes for each Boolean value was used to decide whether to draw a line between the points shown in figure 2.

Next we follow the pseudocode algorithm and make the initial cell (this can be any cell) the current cell and mark it as visited. A visited attribute is added for the cell and a global variable named current, the initial cell is initialized as `current = grid[0]` each visited cell is marked as visited using a Boolean value.

The next step is to determine if the current cell has any neighbors that have not been visited, this is implemented using a function called `checkNeighbours()`, where for each cell located on the grid with the coordinates (c,r) each neighbor is checked if it is visited using the coordinates shown below.

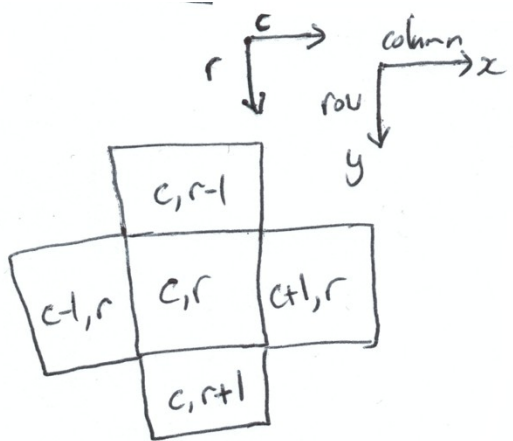


Figure 3 Illustration of neighbours of cell with coordinates (c,r)

However as neighbors should only be cells in the maze grid the `checkNeighbours()` function calls an `index(c,r)` function to check that the column and row coordinate in figure 3 is not an edge case as illustrated below.

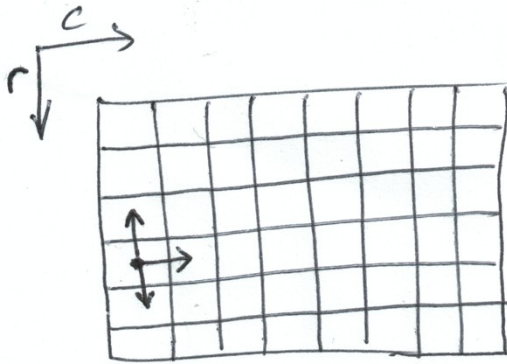


Figure 4 Illustration of an example boundary case on the cell grid

The `checkNeighbour()` function verify that the neighboring cell exists, that is it is within the boundaries of the maze grid and that is has not been visited. The function then returns a randomly selected neighbor from the array, otherwise it returns and undefined value. (See [Appendix C](#))

In a current cell, the code looks at all its available neighbors to find any that is not visited, this selected cell is made the current cell to fulfil the requirement “Make the chosen cell the current cell and mark it as visited”

```
var next = current.checkNeighbours(); //Check the neighbours find a random unvisited one and return it.
if(next){
    next.visited = true;
    current = next;
}
```

Up to this point of the code the following parts of the algorithm have been satisfied

1. Make the initial cell the current cell and mark it visited.
2. While there are unvisited cells
 1. If the current cell has any neighbours which have not been visited
 1. Choose randomly one of the unvisited neighbours
 2. Push the current cell to the stack.
 3. Remove the wall between the current cell and the chosen cell.
 4. Make the chosen cell the current cell and mark it as visited.
 2. Else if stack is not empty
 1. Pop a cell from the stack.
 2. Make it the current cell

The next part to remove a wall between the current cell and a neighboring cell marked as visited. In order to determine whether the a wall is to the right or the left of the cell, on the x-axis, or whether the wall is at the top or the bottom of the current cell, on the y-axis. The logic below is used to determine this case which will return a value of 1 or -1.

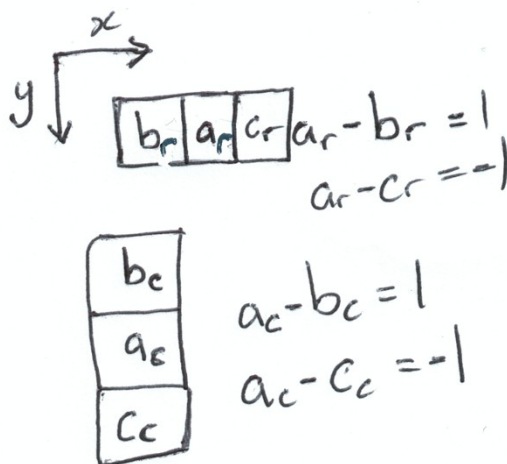


Figure 5 Illustration of the logic to determine which walls are to be removed.

This logic is implemented in a function called `removeWalls(a,b)` as shown below.

```
function removeWalls(a,b){  
  
    var x = a.c - b.c;  
    console.log(a);  
  
    console.log(x);  
    if(x === 1){  
        a.walls[3] = false;  
        b.walls[1] = false;  
    } else if (x === -1){  
        a.walls[1] = false;  
        b.walls[3] = false;  
    }  
  
    var y = a.r - b.r;  
    console.log(a);  
    console.log(y);  
    if(y === 1){  
        a.walls[0] = false;  
        b.walls[2] = false;  
    } else if (y === -1){  
        a.walls[2] = false;  
        b.walls[0] = false;  
    }  
}
```

To this point the added functionality is “3. Remove the wall between the current cell and the chosen cell.”. The remaining part of the algorithm hitherto is:

<div>2. Push the current cell to the stack</div> <div>Else if stack is not empty</div> <div>1. Pop a cell from the stack.</div> <div>2. Make it the current cell</div>
--

The code so far is shown in [Appendix D](#).

To this point the part of the reverse backtracking algorithm completed is highlighted in yellow the remaining sections to be discussed is shown in green.

1. Make the initial cell the current cell and mark it visited.
2. While there are unvisited cells
 1. If the current cell has any neighbours which have not been visited
 1. Choose randomly one of the unvisited neighbours
 2. Push the current cell to the stack.
 3. Remove the wall between the current cell and the chosen cell.
 4. Make the chosen cell the current cell and mark it as visited.
 2. Else if stack is not empty
 1. Pop a cell from the stack.
 2. Make it the current cell

Up to this point a maze grid is created with a cell object that marches around the grid removing walls until it is stuck. This is the point where backtracking needs to be implemented to for the cell to reverse back to a point where along the pattern it marched it has an available space to go to complete maze generation.

To push the current cell to the stack we create a stack array, `var stack = [];`, and push the current index onto the stack, that is to place it in the last index of the array as follows `stack.push(current);`

The stack is called upon when the current cell gets stuck as it traverses when there is no unvisited neighbor left. The stack is used to keep track of the path traversed, to backtrack the current cell when it gets stuck, given the stack is not empty.

```
current = next;
} else if (!(stack.length == 0)) {
    var aCell = stack.pop();
    current = aCell;
}
```

The completed algorithm can be found in [Appendix E](#).

The algorithm demonstrated in the appendix was altered to have 27 rows and columns connected by blocks. In this code the team has used lines on a 8 by 8 grid canvas to develop the maze generation concept.

2.6 Path search algorithms

The ghosts in the game they are taking different algorithms to calculate the path to chase the Pacman.

```
/**
 * ghost 1 is random
 */
if (this.ghost_index === 0) {
    return this.chooseRandom(possible_cells);
}

this.chooseRandom = function (possible_cells) {
    return random(possible_cells).map(function (cell_index, i) {
        return cell_index - this.current_cell[i];
    }, this);
}
```

The first ghost is moving randomly, it calls the chooseRandom() function to choose next move, and it doesn't chase the Pacman.

Code

```
/**
 * ghost 2 follows pacman 1
 */
if (this.ghost_index == 1) {
    return this.followTarget(0, possible_cells);
}

this.followTarget = function (pacman_index, possible_cells) {
    var target = this.is_scared ? this.corner : [pacmans[pacman_index].x, pacmans[pacman_index].y];
    /**
     * measure distances
     */
    var distances_from_target = possible_cells.map(function (possible, i) {
        return dist(possible[0] * 30, possible[1] * 30, target[0], target[1]);
    }, this);

    var shortest_choice_indx = distances_from_target.indexOf(Math.min.apply(null, distances_from_target));
    return possible_cells[shortest_choice_indx].map(function (cell_index, i) {
        return cell_index - this.current_cell[i];
    }, this);
}
```

The second ghost is chasing the Pacman, it calls the followTarget() function to get the distance for Pacman position and available movement.

```
search: function(grid, start, end, heuristic) {

    var heuristic = heuristic || astar.manhattan;
    var openHeap = new BinaryHeap( function(node) {return node.f;} );
    openHeap.push(start);

    while (openHeap.size() > 0) {
        // Grab the lowest f(x) to process next. Heap keeps this sorted for us.
        var currentNode = openHeap.pop();
        // End case -- result has been found, return the traced path
        if (currentNode === end) {
            var curr = currentNode;
            var ret = [];
            while(curr.parent) {
                ret.push(curr);
                curr = curr.parent;
            }
            return ret.reverse();
        }
        // Normal case -- move currentNode from open to closed, process each of its neighbors
        currentNode.closed = true;
        var neighbors = astar.neighbors(grid, currentNode);

        for (var i = 0, il = neighbors.length; i < il; i++) {
            var neighbor = neighbors[i];

            if ( neighbor.closed || neighbor.isWall() ) {
                // not a valid node to process, skip to next neighbor
                continue;
            }

            var gScore = currentNode.g + 1;
            var beenVisited = neighbor.visited;

            if (!beenVisited || gScore < neighbor.g) {
                // Found an optimal (so far) path to this node. Take score for node to see how good it is.
                neighbor.visited = true;
                neighbor.parent = currentNode;
                neighbor.h = neighbor.h || heuristic(neighbor.pos, end.pos);
                neighbor.g = gScore;
                neighbor.f = neighbor.g + neighbor.h;

                if (!beenVisited) {
                    // Pushing to heap will put it in proper place based on the 'f' value.
                    openHeap.push(neighbor);
                } else {
                    // Already seen the node, but since it has been rescored we need to reorder it in the heap
                    openHeap.rescoreElement(neighbor);
                }
            }
        } // for
    } // while
    // No result was found -- empty array signifies failure to find path
    return [];
}, // search
```

The third ghost is chasing the Pacman and this function is used to compute the shortest path for the ghosts, we use the Manhattan distances as an admissible heuristic.

A* search algorithm computes $f(n) = g(n) + h(n)$. The neighbour.g is the shortest distance from start position to current node, we need to check if the path we have arrived at this neighbour is the shortest one we have seen yet. 1 is the distance from a node to its neighbour, and this could be variable for weighted paths. Then neighbor.h is the heuristic function which use Manhattan distance. And last the neighbour is result of the A* search algorithm calculation.

```
manhattan: function(pos0, pos1) {  
    var d1 = pos1.x - pos0.x;  
    if (d1 < 0) d1 = -d1; // eq. Math.abs();  
    var d2 = pos1.y - pos0.y;  
    if (d2 < 0) d2 = -d2;  
    return d1 + d2;  
}, // manhattan
```

Although AStar needs to be run every time a path is to be found, it is very quick and does not expand too many nodes beyond those on the optimal path.

3.0 Testing

3.1 Software test description

Test Environment

The test environment must meet the minimum recommended specifications to execute JavaScript software code.

Operating Systems:

The end product is cross platform

- Windows 10
- Windows 8, 8.1
- Windows 7
- Windows vista
- Windows server 2008 and later
- Linux
- Unix
- Mac

Hardware Environment:

- Processor: x86 or x64
- RAM : 512 MB (minimum), 1 GB (recommended)
- Hard disc: up to 3 GB of free space may be required

Software Environment:

- Code editor
- Local server extension or web application bundler.
 - Live Server extension for visual studio, or
 - Parceljs
- Hard disk: up to 100MB of free space may be required,
- Web Browsers with JavaScript enabled, see table below

Internet Explorer	Microsoft Edge	Mozilla Firefox	Chrome	Opera	Safari	
8 +	Latest	Latest	22 +	17 +	12 +	5 +

Software Test Cases

Test Case ID	Te_001	Test Case Description	Test the move function of pac-man in the Game		
Created By	Yasin Çakar	Reviewed By	David Todorovic	Version	1.0
QA Tester's Log	Test Pac-man move function, analyze code and report to David				
Verified Requirement		REQ1 8	The user can control PacMan direction to go up, down, right or left in the maze		
Tester's Name	Yasin	Date Tested	1-Sep-21	Test Case (Pass/fail/Not Executed)	PASS
Step #	Prerequisites:		Step #	Test Data/Action ([Data] & <Action>)	
1	Access any of the recommended browsers		1	Button: <Key up>	keycode: [38]
2			2	Button: <Key down>	keycode: [40]
3			3	Button: <Key left>	keycode: [37]
4			4	Button: <Key right>	keycode: [39]
Test Scenario					
Step #	Step Details	Expected Results	Actual Results	Pass/Fail/Not executed/Suspended	
1	Access and open the source code	Source code should be configured to run with a server application	As expected	Pass	
2	Point localhost server to source code directory	Server application should open and point to source code	As expected	Pass	
3	Run the local server	Server should launch the code	As expected	Pass	
4	Start Game	The game should be polling for keypress	As expected	Pass	
5	Press the control buttons mentioned in <u>Test Data/Action</u>	Game starts, Ghost characters move, pac-man character waits for user input	Pac-man character respond to key presses and moves in all four directions	Pass	

Test Case ID	Te_002	Test Case Description	Test Maze border in the game		
Created By	Yasin Çakar	Reviewed By	David Todorovic	Version	1.0
QA Tester's Log	Test the level class stops pac-man movement when it hits a wall share findings with David and Jerry.				
Verified Requirement	REQ17	Pacman moves inside the game maze as soon as the user presses any key			
	REQ18	The user can control PacMan direction to go up, down, right or left in the maze			
	REQ19	Pacman will continue to move inside the maze unless it collides with a ghost or with maze wall			
Tester's Name	Yasin	Date Tested	1-Sep-21	Test Case (Pass/fail/Not Executed)	PASS
Step #	Prerequisites:		Step #	Test Data/Action ([Data] & <Action>)	
1	Access any of the recommended browsers		1	Button: <Any key>	
2			2	Button: Control Buttons mentioned in Te_001	
3			3		
4			4		
Test Scenario					
Step #	Step Details	Expected Results	Actual Results	Pass/Fail/Not executed/Suspended	
1	Access and open the source code	Source code should be configured to run with a server application	As expected	Pass	
2	Point localhost server to source code directory	Server application should open and point to source code	As expected	Pass	
3	Run the local server	Server should launch the code	As expected	Pass	
4	Start Game	The game should be polling for keypress	As expected Pac-man character	Pass	

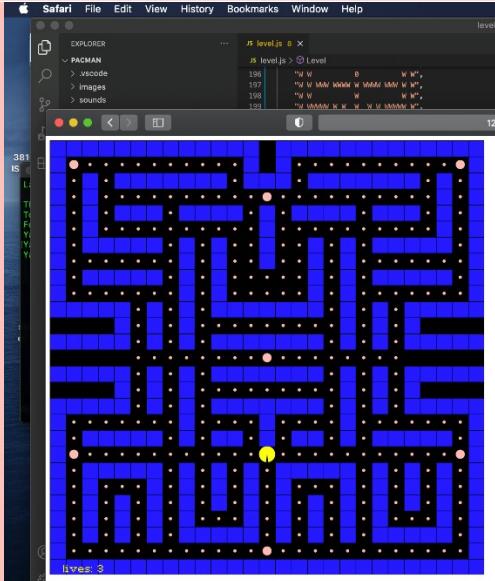
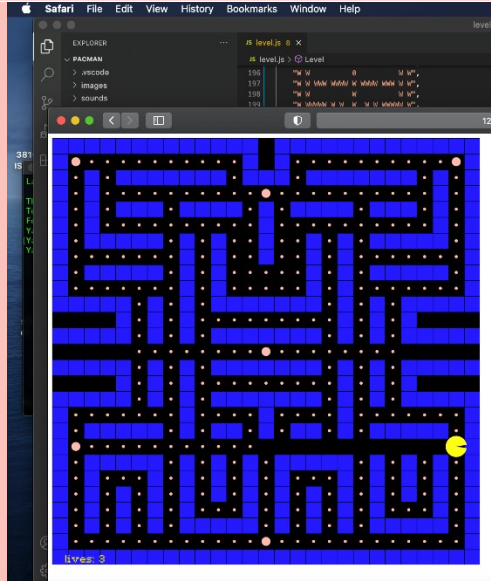
			respond to key presses and moves in all four directions	
5	Move into a wall	Pac-man stops moving until a new direction is given	As expected	Pass

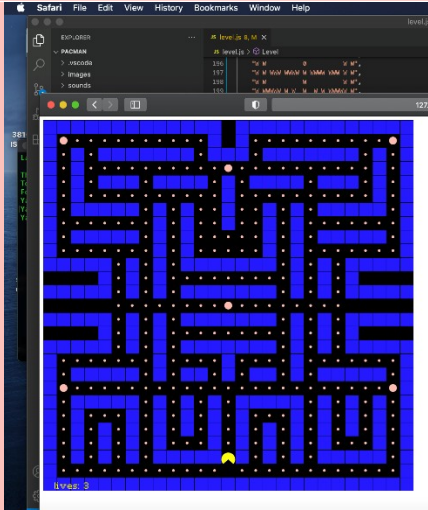
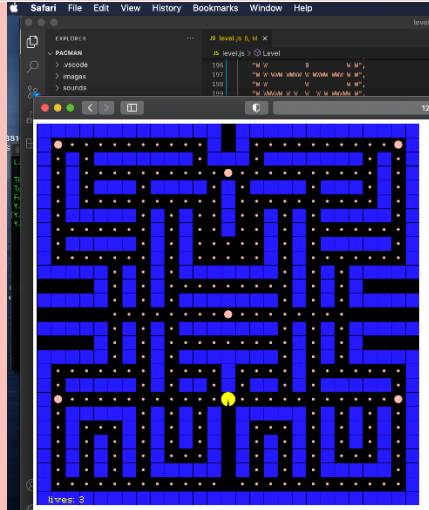
Test Case ID	Te_003	Test Case Description	Test the EatGhost(ghost) method when a power pellet is consumed		
Created By	Yasin Çakar	Reviewed By	David Todorovic	Version	1.0
QA Tester's Log	Test if pac-man can eat a ghost after eating a power pellet. Share findings with team.				
Verified Requirement		REQ22	Eating power pellet dots causes the 4 coloured ghosts to turn blue for a pre-set time.		
		REQ23	Pacman can eat the ghosts if ghost color is blue		
Tester's Name	Yasin	Date Tested	1-Sep-21	Test Case (Pass/fail/Not Executed)	PASS
Step #	Prerequisites:		Step #	Test Data/Action ([Data] & <Action>)	
1	Access any of the recommended browsers		1	Pacman to collide with a power pellet	
2			2	Pacman is then to be collide with a Ghost.	
3			3		
4			4		
Test Scenario					
Step #	Step Details	Expected Results	Actual Results		Pass/Fail/Not executed/Suspended
1	Access and open the source code	Source code should be configured to run with a server application	As expected	Pass	
2	Point localhost server to source code directory	Server application should open and point to source code	As expected	Pass	
3	Run the local server	Server should launch the code	As expected	Pass	
4	Start Game	The game should be polling for keypress	As expected	Pass	
5	Press the control buttons mentioned in Test Data/Action	Game starts, Ghost characters move, pac-man character waits for user input	Pac-man character respond to key presses and moves in all four	Pass	




			directions	
6	Move Pacman into a power pellet	Ghosts Characters turn blue	As expected	Pass
7	Move pacman into a Ghost character	Ghost character is eliminated, pac-man is unaffected in the game canvas	As expected	Pass

3.2 Software test report

This section will discuss the results of the test cases in Section [3.1 Software test Description](#). This section contains the results to confirm the test outcomes of section 3.1.

Test Case ID:	Te_001
Test Case Description:	Test the move function of pac-man in the Game
Expected Results:	Pacman moves in the direction prompted
Actual Results:	As Expected
Test Results:	
Before Test Data/Action	
	
After Test Data/Action	
	
Discussion of Findings	
Ghost elements of the game are removed, on game start up pac-man is successfully moved from the starting point to the right of the canvas.	
The same results were true for UP, DOWN and LEFT functionalities. (Note the trail of pellets consumed)	
Errors	
Error Type:	Description:
N/A	N/A

Test Case ID:	Te_002
Test Case Description:	Test Maze border in the game
Expected Results:	Pacman stops moving after colliding with a wall, the code waits for new direction without crashing.
Actual Results:	As Expected
Test Results:	
Before Test Data/Action	After Test Data/Action
	
Discussion of Findings	
Ghost elements of the game are removed, on game start up pac-man is successfully up starting from the bottom of canvas until it is obstructed. (Note the trail of pellets consumed)	
Errors	
Error Type:	Description:
N/A	N/A

Test Case ID:	Te_003
Test Case Description:	Test the EatGhost(ghost) method when a power pellet is consumed
Expected Results:	Pacman is able to eat the ghost when they are blue.
Actual Results:	As Expected
Test Results:	
Before Test Data/Action	After Test Data/Action
	 
Discussion of Findings	
Pac-man was successfully able to eat (i.e. eliminate from the canvas) any ghost after colliding with a power pellet.	
Errors	
Error Type:	Description:
N/A	N/A

Discussion:

By following good programming practices such as test-driven development, in the case of this team Test first development was employed where each functionality was tested separately as unit tests.

As each functional requirement was satisfied as discrete independent modules the team worked on integrating each of these discrete functionalities.

The testing methodologies employed above are white box testing methods. The test cases performed for section 3.1 are the acceptance tests to satisfy the functional requirements in section 2.1 in stage, in order to satisfy the teams acceptance criteria, that is to satisfy stage 2 of demonstrating understanding of Software engineering principles.

4.0 Reflection

[Master student only, you need to write at least one page]

This Assignment activity covered the implementation of Pacman game using architecture design patterns, to use best software developments methods to design a working game that fulfills the requirements to deliver the expected product developed using industry standard design methods in accordance to our design diagram structures and algorithms.

The aim and challenge of this assessment was to build the right software (requirements of this assignment being the Pacman game), as well as to build it “right” to ensure design efficiency, modularity, reusability and testability.

Since graphical user interface “GUI” is one of the main elements of the game, and a key visual indicator of successful logic implementation. Model/View/Controller (MVC) software architecture design pattern has been used in order to decouple components and enable reuse as well a flexibility in implementation various views on existing model code.

The team have decided to use JavaScript as the main programming language to ensure that the game can be executed as cross platform through the use of a webserver and a browser. JavaScript’s library “P5.js” has been used to create the game’s graphics and interface, this has been seen in section 2.5 for the random maze generation algorithm where the final code in Appendix E can be executed independently on <https://editor.p5js.org/>.

JSDoc tool was used to generate the code document, which conveniently utilized existing comments to generate documentation output. By iteratively adding code comments throughout the development process to the source code, this also enabled the communication of the functionality of each method and class. This practice allowed each team member to make changes to other members’ code in between each commit within the version control-based development.

One of the design tactics that we used in our implementation is “fault detection”, which helped the team during the implementation phase to early catch errors and rectify them, consequently improved efficiency and reduced development time. As well as promoting the practice of test first development and unit testing between each of the development stages.

During our project planning sessions, we divided the project into smaller sections and distributed the work across the team, we reviewed each other work, provided and received feedback. The most important lesson learned with this approach was the priority and sequencing of each section while managing dependencies between these sections, for given task deadlines. We learnt how to work efficiently and effectively as a team.

Overall, as a team, we’re proud of our final creation. The complexity of the project had taught us many techniques, tools and improved our overall skill level.

5.0 Video link

[please put the URL link of your video, please make sure the video can be viewed by the assessor]

<https://web.microsoftstream.com/video/3013947d-993c-4f0e-8e6f-61a5107394c1>

6.0 Appendices

Appendix A

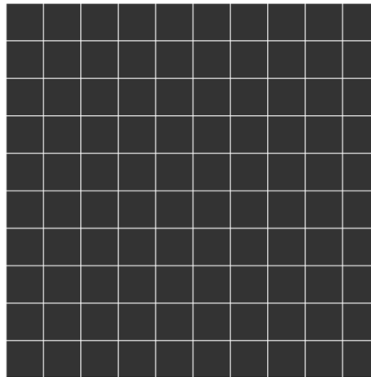
CODE
<pre>var cols,rows; var w = 40; // Width and height of each cell object var grid = []; // Stores all the Cell objects function setup(){ createCanvas(400,400); cols = floor(width/w); // Total number of columns rows = floor(height/w); // Total number of rows // Creating Cell objects in the Maze Grid // for every row go through every column for(var r = 0; r < rows; r++){ // Rows for(var c = 0; c < cols; c++){ // Columns var cell = new Cell(c,r); // Create 100 cell objects grid.push(cell); } } } function draw(){ background(51); for(var i = 0; i < grid.length; i++){ // Loop through all and display grids grid[i].show(); } } /* * Constructor Function for a cell object */ function Cell(c,r){ this.r = r; // Column number this.c = c; // Row number</pre>

```

this.show = function(){
  var x = this.c*w; // column location * cell width
  var y = this.r*w;
  stroke(255);
  noFill();
  rect(x,y,w,w);
}
}

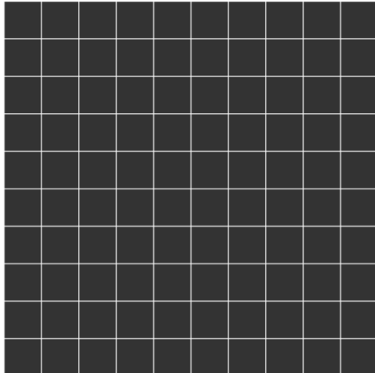
```

OUTPUT



Appendix B

Lines Changed	CODE
	<pre> var cols,rows; var w = 40; // Width and height of each cell object var grid = []; // Stores all the Cell objects function setup(){ createCanvas(400,400); cols = floor(width/w); // Total number of columns rows = floor(height/w); // Total number of rows // Creating Cell objects in the Maze Grid // for every row go through every column for(var r = 0; r < rows; r++){// Rows for(var c = 0; c < cols; c++){// Columns var cell = new Cell(c,r); // Create 100 cell objects grid.push(cell); } } } function draw(){ </pre>

	<pre> background(51); for(var i = 0; i < grid.length; i++){ // Loop through all and display grids grid[i].show(); } } /* * Constructor Function for a cell object * */ function Cell(c,r){ this.r = r; // Column number this.c = c; // Row number this.show = function(){ var x = this.c*w; // column location * cell width var y = this.r*w; stroke(255); line(x,y,x+w,y); line(x+w,y,x+w,y+w); line(x+w,y+w,x,y+w); line(x,y+w,x,y); // noFill(); // rect(x,y,w,w); } } </pre>
	<div>OUTPUT</div>
	

Appendix C

CODE
<pre> var cols,rows; var w = 40; var grid = []; var current; // The current cell, used when the program starts traversing cells on the grid. </pre>

```

function setup(){
  createCanvas(400,400);
  cols = floor(width/w);
  rows = floor(height/w);

  // for every row go through every column
  for(var r = 0; r < rows; r++){ // Rows
    for(var c = 0; c < cols; c++){ // Columns
      var cell = new Cell(c,r); // Create 100 cell objects
      grid.push(cell);
    }
  }

  current = grid[0];
}

function draw(){
  background(51);
  for(var i = 0; i < grid.length; i++){ // Loop through all and display grids
    grid[i].show();
  }

  current.visited = true;
  var next = current.checkNeighbours(); //Check the neighbours find a random unvisited one
  and return it.
  if(next){
    next.visited = true;
    current = next;
  }
}

function index(c,r){

  // Columns must be between 0 and cols-1,else invalid
  // Rows has to be between 0 and rows-1,else invalid
  if(c < 0 || r < 0 || c > cols-1 || r > rows-1){
    return -1;//Invalid
  }

  return c + r * cols; // Getting an index into a one-dimensional array
}

function Cell(c,r){
  this.r = r; // Column number
  this.c = c; // Row number
  this.walls = [true,true,true,true]; //[top,right,bottom,left]
}

```



```

this.visited = false;

this.checkNeighbours = function(){
    var neighbours = [];

    //var index = c + (r-1) * cols;// Getting an index into a one-dimensional array
    var top    = grid[index(c,r - 1)];
    var right  = grid[index(c + 1,r)];
    var bottom = grid[index(c,r + 1)];
    var left   = grid[index(c - 1,r)];

    if(top && !top.visited){ // If top exists and has not been visited
        neighbours.push(top);
    }
    if(right && !right.visited){
        neighbours.push(right);
    }
    if(bottom && !bottom.visited){
        neighbours.push(bottom);
    }
    if(left && !left.visited){
        neighbours.push(left);
    }

    // Are there neighbours that have not been visited, if so select one of those
    if(neighbours.length > 0){
        var rand = floor(random(0,neighbours.length))// Pick a random neighbour randomly
        return neighbours[rand]; // return a randomly selected neighbour
    } else {
        return undefined;
    }
}

this.show = function(){
    var x = this.c*w; // column location * cell width
    var y = this.r*w;
    stroke(255);

    if (this.walls[0]){
        line(x,y,x+w,y);
    }
    if (this.walls[1]){
        line(x+w,y,x+w,y+w);
    }
    if (this.walls[2]){
        line(x+w,y+w,x,y+w);
    }
}

```

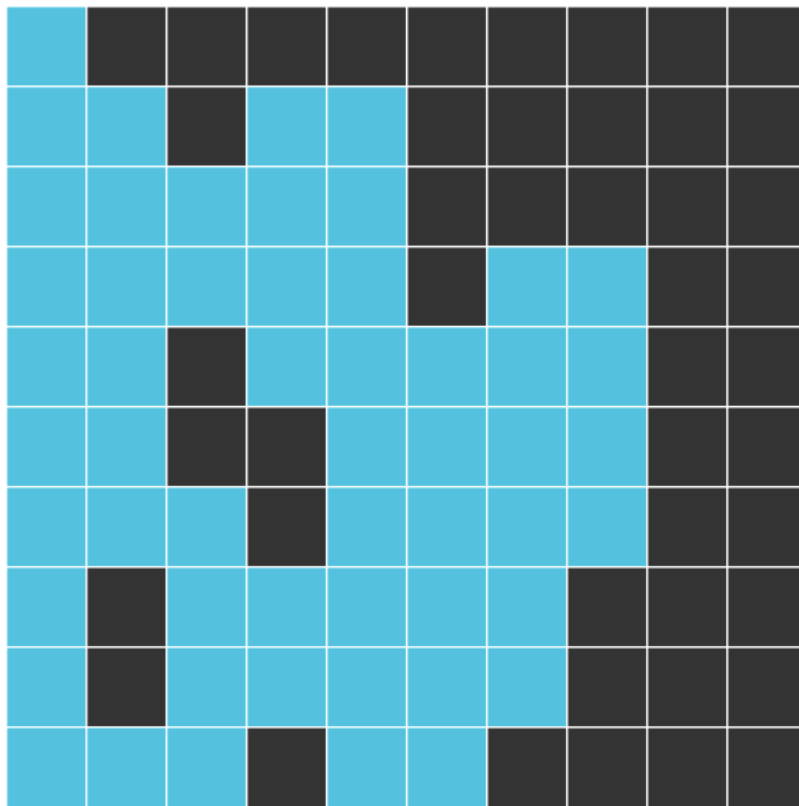
```

    if (this.walls[3]){
        line(x,y+w,x,y);
    }

    if(this.visited){ // if the cell has been visited show it as coloured
        fill('#00c3e3');
        rect(x,y,w,w);
    }
}
}
}

```

OUTPUT



Appendix D

CODE

```

var cols,rows;
var w = 50;
var grid = [];

var current; // The current cell, used when the program starts traversing cells on the grid.

function setup(){
    createCanvas(400,400);
}

```

```

cols = floor(width/w);
rows = floor(height/w);
frameRate(10);

// for every row go through every column
for(var r = 0; r < rows; r++){ // Rows
  for(var c = 0; c < cols; c++){ // Columns
    var cell = new Cell(c,r); // Create 100 cell objects
    grid.push(cell);
  }
}

current = grid[0];
}

function draw(){
  background(51);
  for(var i = 0; i < grid.length; i++){ // Loop through all and display grids
    grid[i].show();
  }

  current.visited = true;
  current.highlight();
  // STEP 1 - Choose randomly one of the visited neighbours
  var next = current.checkNeighbours(); //Check the neighbours find a random unvisited one
and return it.
  if(next){
    next.visited = true;
    // STEP 3
    removeWalls(current,next);

    // STEP 4 - Make the chosen cell the current cell and mark it as visited.
    current = next;
  }
}

function index(c,r){

  // Columns must be between 0 and cols-1,else invalid
  // Rows has to be between 0 and rows-1,else invalid
  if(c < 0 || r < 0 || c > cols-1 || r > rows-1){
    return -1;//Invalid
  }

  return c + r * cols; // Getting an index into a one-dimensional array
}

```

```

function Cell(c,r){
  this.r = r; // Column number
  this.c = c; // Row number
  this.walls = [true,true,true,true]; //[top,right,bottom,left]
  this.visited = false;

  this.checkNeighbours = function(){
    var neighbours = [];

    //var index = c + (r-1) * cols;// Getting an index into a one-dimensional array
    var top = grid[index(c,r - 1)];
    var right = grid[index(c + 1,r)];
    var bottom = grid[index(c,r + 1)];
    var left = grid[index(c - 1,r)];

    if(top && !top.visited){ // If top exists and has not been visited
      neighbours.push(top);
    }
    if(right && !right.visited){
      neighbours.push(right);
    }
    if(bottom && !bottom.visited){
      neighbours.push(bottom);
    }
    if(left && !left.visited){
      neighbours.push(left);
    }

    // Are there neighbours that have not been visited, if so select one of those
    if(neighbours.length > 0){
      var rand = floor(random(0,neighbours.length))// Pick a random neighbour randomly
      return neighbours[rand]; // return a randomly selected neighbour
    } else {
      return undefined;
    }
  }

  this.highlight = function(){
    var x = this.c*w;
    var y = this.r*w;
    noStroke();
    fill(0,0,255,100);
    rect(x,y,w,w);
  }

  this.show = function(){
    var x = this.c*w; // column location * cell width

```

```

var y = this.r*w;
stroke(255);

if (this.walls[0]){
    line(x,y,x+w,y);
}
if (this.walls[1]){
    line(x+w,y,x+w,y+w);
}
if (this.walls[2]){
    line(x+w,y+w,x,y+w);
}
if (this.walls[3]){
    line(x,y+w,x,y);
}

if(this.visited){ // if the cell has been visited show it as coloured
    noStroke();
    //strokeWeight(0);
    //fill('#00c3e3');
    fill(0, 195, 227);
    rect(x,y,w,w);
    console.log("Remove wall");
    console.log(x);
    console.log(y);
}
}
}

function removeWalls(a,b){

var x = a.c - b.c;
console.log(a);

console.log(x);
if(x === 1){
    a.walls[3] = false;
    b.walls[1] = false;
} else if (x === -1){
    a.walls[1] = false;
    b.walls[3] = false;
}

var y = a.r - b.r;
console.log(a);
console.log(y);
if(y === 1){

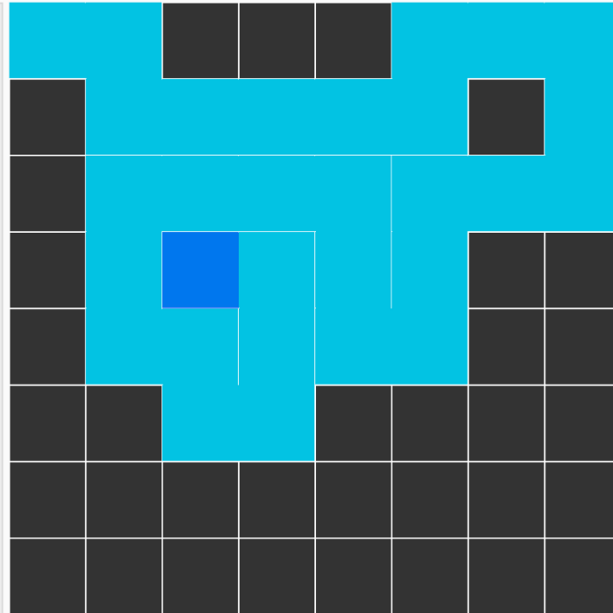
```

```

    a.walls[0] = false;
    b.walls[2] = false;
  } else if (y === -1){
    a.walls[2] = false;
    b.walls[0] = false;
  }
}

```

OUTPUT



Appendix E

CODE

```

// Remove maze walls
// Next step of the algorithm: Backtracking

// Deminatrations for Group 1 3805ICT/7805ICT
// Demonsatrtor: Yasin and Group 1

var cols,rows;
var w = 50;
var grid = [];

var current; // The current cell, used when the program starts traversing cells on the grid.

var stack = [];

function setup(){

```

```

createCanvas(400,400);
cols = floor(width/w);
rows = floor(height/w);
frameRate(10);

// for every row go through every column
for(var r = 0; r < rows; r++){ // Rows
  for(var c = 0; c < cols; c++){ // Columns
    var cell = new Cell(c,r); // Create 100 cell objects
    grid.push(cell);
  }
}

current = grid[0];
}

function draw(){
  background(51);
  for(var i = 0; i < grid.length; i++){ // Loop through all and display grids
    grid[i].show();
  }

  current.visited = true;
  current.highlight();
  // STEP 1 - Choose randomly one of the visited neighbours
  var next = current.checkNeighbours(); //Check the neighbours find a random unvisited one
  and return it.
  if(next){
    next.visited = true;

    // STEP 2
    stack.push(current);

    // STEP 3
    removeWalls(current,next);

    // STEP 4 - Make the chosen cell the current cell and mark it as visited.
    current = next;
  } else if (!(stack.length == 0)) {
    var aCell = stack.pop();
    current = aCell;
  }
}

function index(c,r){

  // Columns must be between 0 and cols-1,else invalid

```

```

    // Rows has to be between 0 and rows-1,else invalid
    if(c < 0 || r < 0 || c > cols-1 || r > rows-1){
        return -1;//Invalid
    }

    return c + r * cols; // Getting an index into a one-dimensional array
}

function Cell(c,r){
    this.r = r; // Column number
    this.c = c; // Row number
    this.walls = [true,true,true,true]; //[top,right,bottom,left]
    this.visited = false;

    this.checkNeighbours = function(){
        var neighbours = [];

        //var index = c + (r-1) * cols;// Getting an index into a one-dimensional array
        var top = grid[index(c,r - 1)];
        var right = grid[index(c + 1,r)];
        var bottom = grid[index(c,r + 1)];
        var left = grid[index(c - 1,r)];

        if(top && !top.visited){ // If top exists and has not been visited
            neighbours.push(top);
        }
        if(right && !right.visited){
            neighbours.push(right);
        }
        if(bottom && !bottom.visited){
            neighbours.push(bottom);
        }
        if(left && !left.visited){
            neighbours.push(left);
        }

        // Are there neighbours that have not been visited, if so select one of those
        if(neighbours.length > 0){
            var rand = floor(random(0,neighbours.length))// Pick a random neighbour randomly
            return neighbours[rand]; // return a randomly selected neighbour
        } else {
            return undefined;
        }
    }

    this.highlight = function(){
        var x = this.c*w;

```



```

    var y = this.r*w;
    noStroke();
    fill(0,0,255,100);
    rect(x,y,w,w);
  }

  this.show = function(){
    var x = this.c*w; // column location * cell width
    var y = this.r*w;
    stroke(255);

    if (this.walls[0]){
      line(x,y,x+w,y);
    }
    if (this.walls[1]){
      line(x+w,y,x+w,y+w);
    }
    if (this.walls[2]){
      line(x+w,y+w,x,y+w);
    }
    if (this.walls[3]){
      line(x,y+w,x,y);
    }

    if(this.visited){ // if the cell has been visited show it as coloured
      noStroke();
      //strokeWeight(0);
      //fill('#00c3e3');
      fill(0, 195, 227);
      rect(x,y,w,w);
      console.log("Remove wall");
      console.log(x);
      console.log(y);
    }
  }
}

function removeWalls(a,b){

  var x = a.c - b.c;
  console.log(a);

  console.log(x);
  if(x === 1){
    a.walls[3] = false;
    b.walls[1] = false;
  } else if (x === -1){

```

```
    a.walls[1] = false;
    b.walls[3] = false;
  }

  var y = a.r - b.r;
  console.log(a);
  console.log(y);
  if(y === 1){
    a.walls[0] = false;
    b.walls[2] = false;
  } else if (y === -1){
    a.walls[2] = false;
    b.walls[0] = false;
  }
}
```

OUTPUT

