

*2018/19*

**Student Name: Zhengli.Yu**

**Student ID:** 201155869

**Project Title:** COMP390.

**Pac-Man**

**Supervisor: Alexei Lisitsa**

**DEPARTMENT OF**

**COMPUTER SCIENCE**

The University of Liverpool, Liverpool L69 3BX

**Pac-Man Project Report**

Zhengli.Yu\_201155869

1. **Abstract**

The main objective of this report is to provide a complete record of the *COMP-390 Pac-Man* project. This report will be divided into 8 sections and covers all aspects during the implement of the project. Bibliography and necessary appendices will also be attached with this report.

The main objective of the project is to develop a PacMan game. This game is based on the original PacMan game and several new game systems are added. One of the key components of this game is the maze generation system. Therefore, background research about the original Pac-man game and maze generation algorithms have been made before the design of the game. Information about the background research will be covered in Background Section.

In the design stage, three game systems, including randomly generated mazes system, shop and items system and AI systems for ghosts, are designed to be implemented in the game. In the final design document, 3 different algorithms for maze generation, 5 different kinds of items and 4 different AI algorithms for ghosts have be discussed. All details about the design will be covered in the Design section of this report.

During the implement of this project, all components covered in the design stage are realised. Part of the design has been revised to fix some problems and provide better performance. In addition, some extra components including animation, sound effect and particle system are added to the game to improve its gameplay performance. The game is developed by Unity game engine. 3D models and animations are developed by MagicaVoxel [1]. The final product is a software that can run on Windows platform.

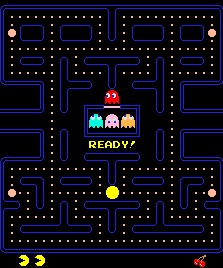
Evaluation stage is also covered in this project. The evaluation contains tests for each game components, game performance test and users’ evaluation. Testing for game components is completed inside Unity game engine. Performance test is performed on the final software and users’ evaluation is carried out by collecting feedback from users. Results about the evaluation will be covered in Evaluation section.

In addition, learning outcomes from this project and professional issues that is met during the project will also be discussed in Learning Points and Professional Issues sections.

1. **Introduction**

The objective of the “Pac-Man” project is to develop a game using the Unity game engine in which a player control Pac-Man to go through mazes and avoid ghosts who will be controlled by artificial intelligence. The project will be supervised by Alexei Lisitsa.

The original Pac-Man is an arcade game developed by Namco and first released in 1980 [2]. In that game, player should navigate a Pac-Man to go through a maze which contains Pac-Dots and ghosts. The goal of the game is to collect points by eating the Pac-Dots and avoid ghosts in the maze. Current stage will be completed and the next stage will be started after reaching the exit of maze with enough points.

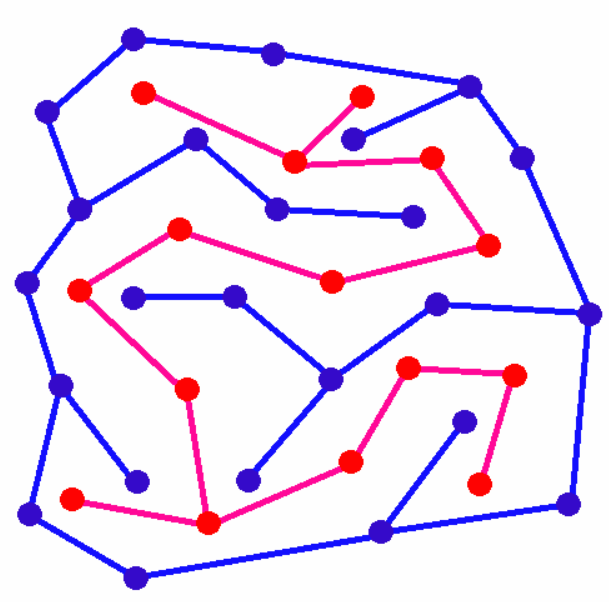
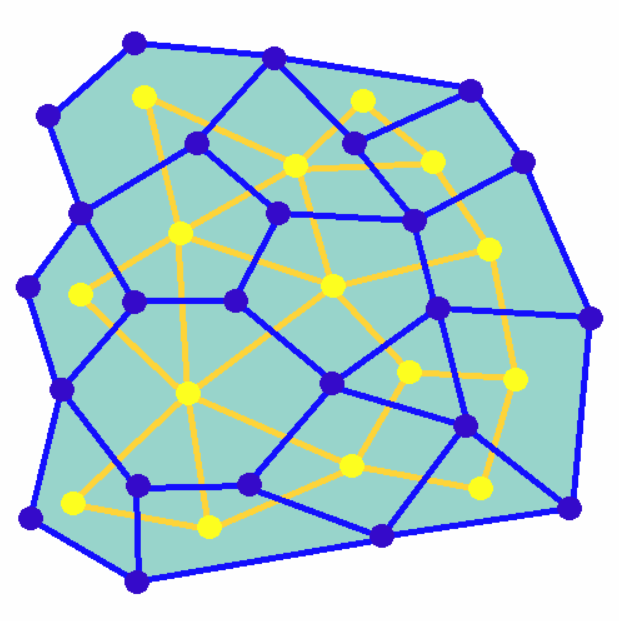


*Figure 1.1 Original Pac-Man Game*

This objective of this project is to implement the Pac-Man game or a variant thereof. This game will be based on the original Pac-Man game. Some extra features such as mazes generation and AI for different ghosts will be added to create a new gameplay experience. This game will consist of three main components: gameplay, mazes generation and AI for ghosts. They will be implemented using Unity game engine. Some other software such as GIMP, Blender and MagicaVoxel will be used to create resources for this game.

1. **Research**

A variety of different aspects which are relevant to this project have been researched. The most important research is about the maze generation algorithms. In this game, a maze will be generated automatically in each stage of the game. A maze generation algorithm starts with a predetermined arrangement of cells with walls between them, which can be considered as a connected graph with nodes representing cells and edges representing possible walls. The purpose of the generation algorithm is to go through all cells and make a path from start cell and end cell (show in figure 2.1) [3].



*Figure 2.1 Generation of a maze*

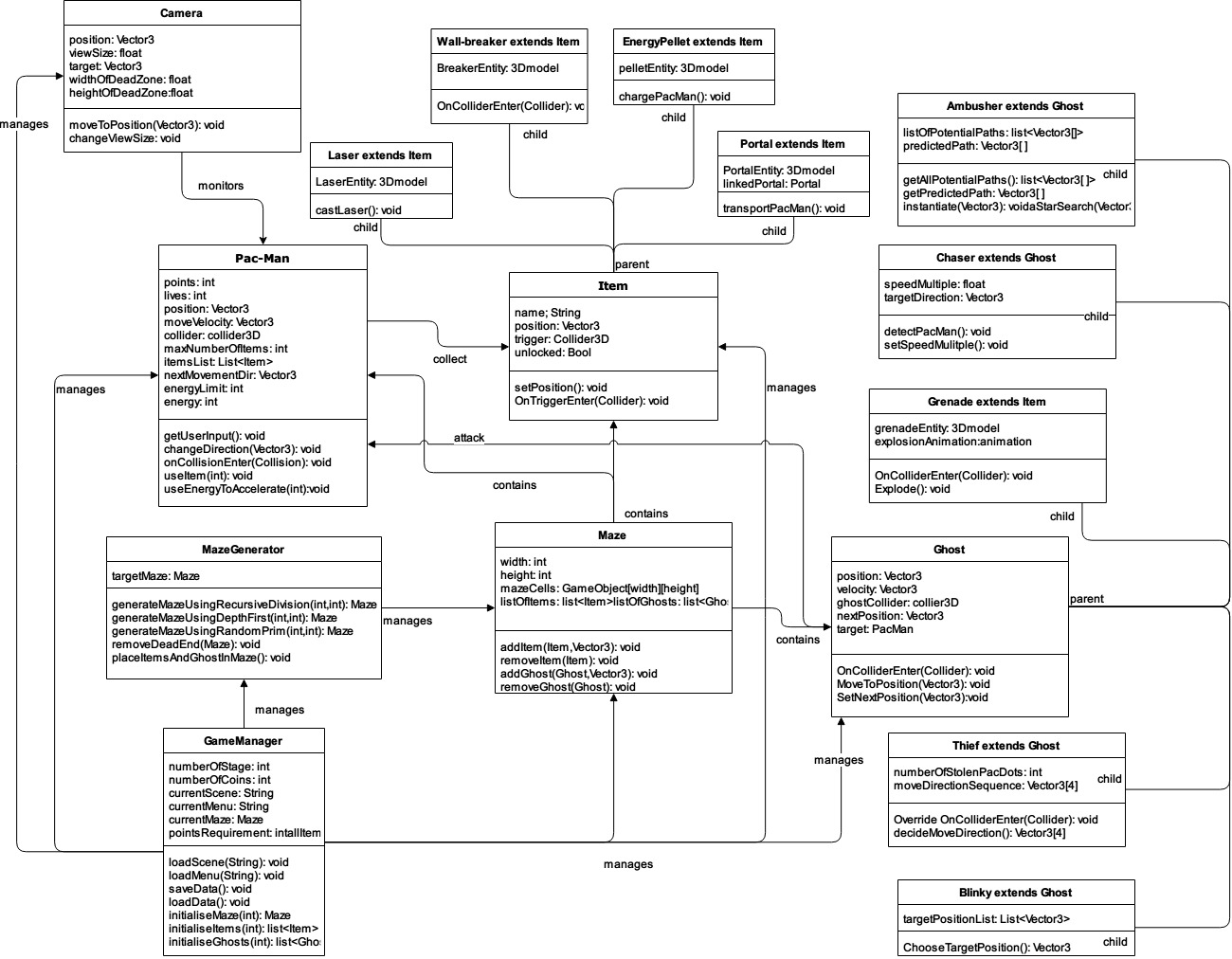
Three maze generation algorithms: Recursive backtracker, Recursive division and Randomized Prim's algorithm have been studied. Mazes generated by Recursive division have more straight roads than corners so it is easiest to go through [3]. Recursive backtracker generates mazes which contains a clear main path and long corridors so it generates a maze with medium difficulty [4]. Randomized Prim's algorithm creates most complex mazes and provides the most difficult game stage [5]. In the game, different maze generation algorithms and size of that maze will be combined to maintain an increasing game difficulty. Details including pseudo codes of these algorithms will be discussed in “Project Design” section.

The Unity game engine has also been researched. Most components of the game will be implemented by Unity and C# scripts which use Unity API will be attached to game objects to implement their control and interactions. The Unity official website provides various documentation and tutorials for beginners. Prior to this document, several tutorials have been studied and some game demos have been implemented. During the implementing stage, Unity is an aspect which will be attached importance and some time has been arranged to the study Unity game engine.

Using of 3D modelling software Blender and MagicaVoxel has also been practised. Blender is a 3D creation suite and it will be used to develop 3D assets of the game environment [6]. MagicaVoxel is a free lightweight 8-bit voxel art editor [1]. It will be used to implement the models and animations of Pac-Man and ghosts.

1. **Project Design**

The object-oriented design will be applied in this project. Object-oriented design is the discipline of defining the objects and their interactions to solve a problem [7]. Therefore, this section will start with a class diagram that contains all classes in the game. Then, details of each class and its design will be discussed.

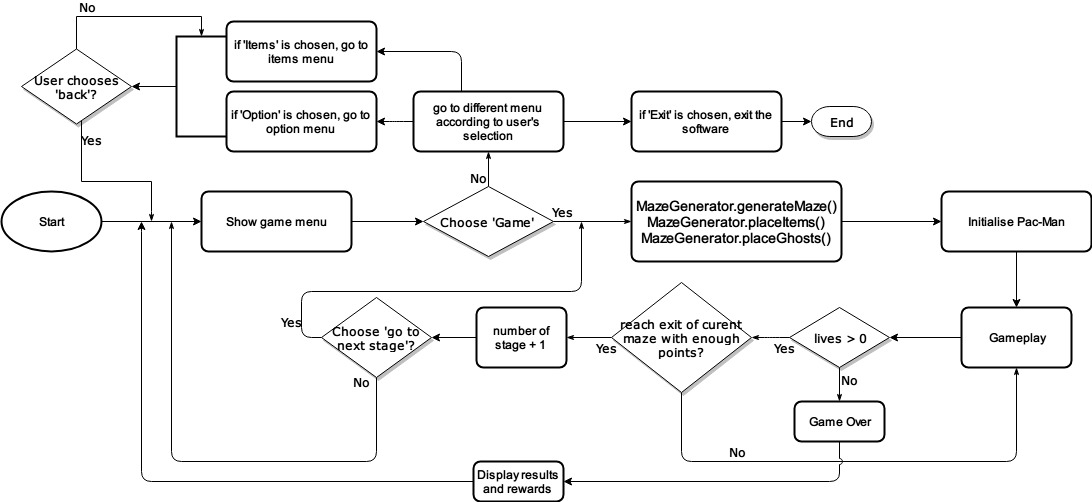


*Figure 1.1.1 Class Diagram*

* 1. **Design of GameManager**

|  |
| --- |
| **GameManager** |
| numberOfStage: int  numberOfCoins: int  currentScene: String  currentMenu: String  currentMaze: Maze  pointsRequirement: int  allItemsList: list<Item> |
| loadScene(String): void  loadMenu(String): void  saveData(): void  loadData(): void  initialiseMaze(int): Maze  initialiseItems(int): list<Item>  initialiseGhosts(int): list<Ghost>  unlockItem(String): void |

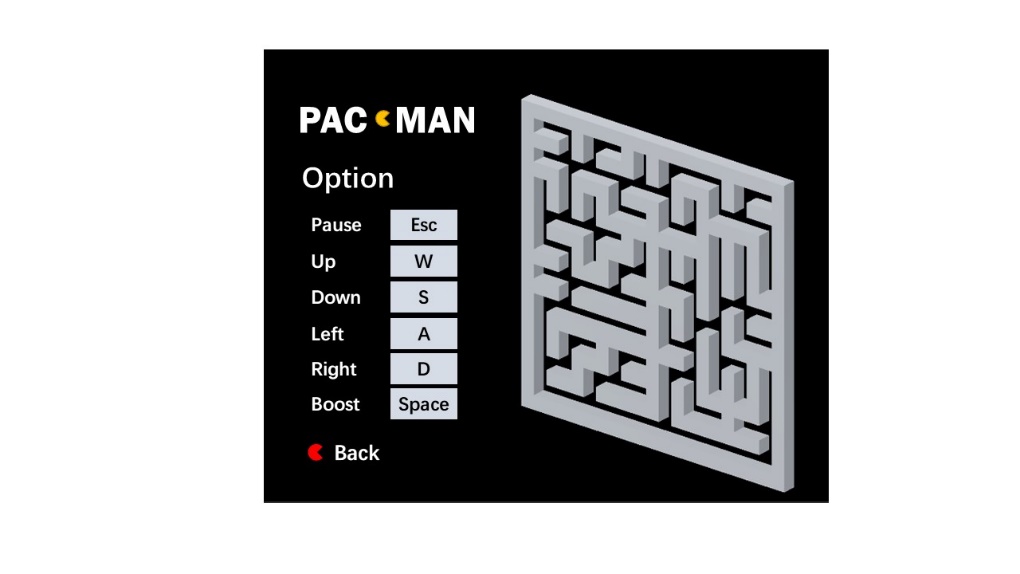
The GameManager manages the logic and content of the game. It is used to load game scenes, initialise game resources, monitor the state of a game and end a game eventually. The following flowchart shows the main procedures of the game.



*Figure 1.1.2 Flow Chart*

The main procedures of the game can be divided into two loops: user interface loop and gameplay loop.

* **User interface loop**
* When the software is launched, a user interface with some buttons will be shown. User can go to a different menu by clicking a related button. Figure 3.1.1 are examples of layout of some user interfaces. In the final product, the background of this interface is expected to be a dynamic scene that shows how a maze is generated. The font and colour in the interface can also be changed.



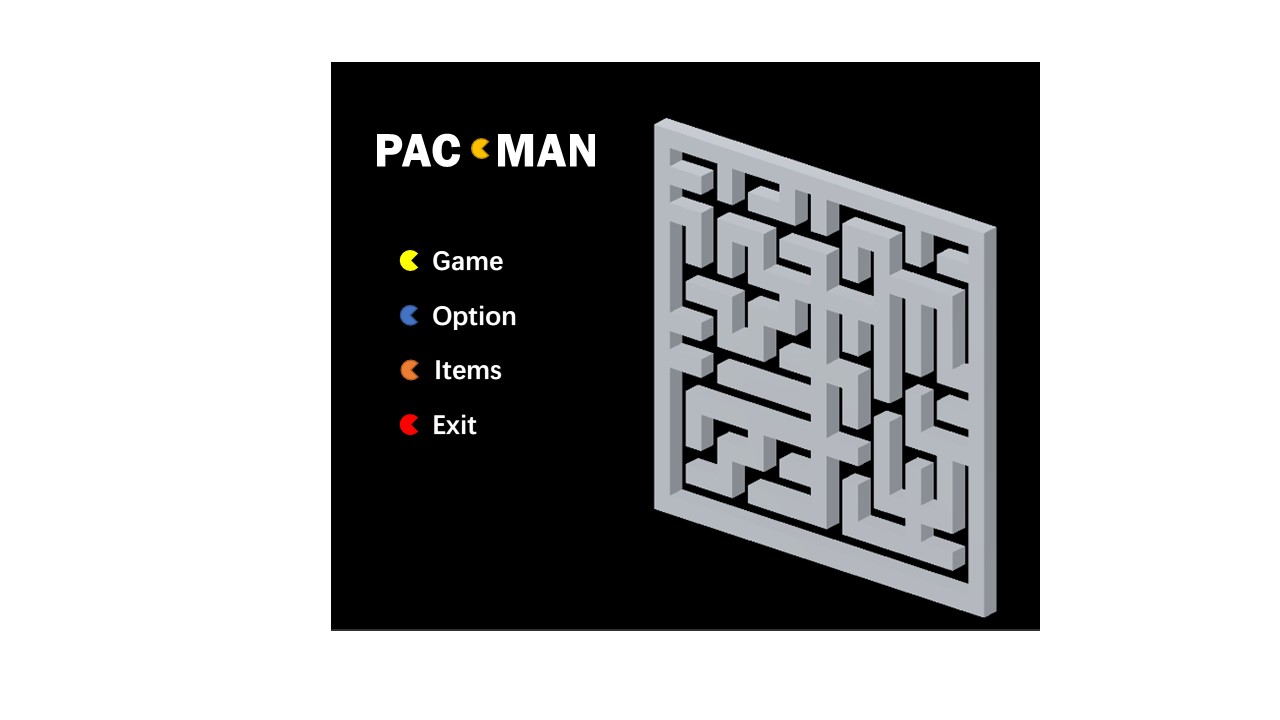


Figure 4.1.1 Examples of Layout of UI

* By clicking the ‘Game’ button, the game scene will be loaded and gameplay loop is started.
* If user chooses ‘Option’, an interface that allows user to change its game setting will appear. User can adjust controls and visual effects for the game.
* In the ‘Items’ interface, user can consume some coins to unlock new item. Coins can be obtained by playing and complete the Pac-Man game, which will be discussed in ‘gameplay loop’ section. In the ‘Item’ menu, guidelines about how to play the game and use items will be provided.
* The software will be closed if ‘Exit’ is clicked
* During game, user can pause the game by clicking a key (‘Esc’ by default). Then user can resume the game, change options or leave the game and go back to menu.
* In each interface, user can go back to the initial interface by clicking the ‘back’ button.
* **Gameplay loop**
* The gameplay loop starts when user selects ‘Game’ and ends when Pac-Man loses all it lives or players chooses to leave the game.
* In each gameplay loop, GameManager will firstly create a new MazeGenerator object. The generator will generate a maze according to current number of stage. Secondly, some items and ghosts will be placed in the maze. Algorithms for generation of mazes, items and ghosts will be discussed in ‘Design of MazeGenerator’ section. After all game environment is set up, Pac-Man will be initialised and player should navigate the Pac-Man through the maze.
* There are several ghosts in each maze. When Pac-Man touches ghosts, Pac-Man will lose one life. If Pac-Man loses all its lives, the game loop will be ended and player will go back to the menu.
* Some items that provide Pac-Man extra abilities will be placed in the maze. Pac-Man has an items list which can contain a limited number of items. If there is available space for a new item, Pac-Man can pick an item by touching it. Then player can use these items by pressing corresponding number keys. Details of functions and design of items will be discussed in ‘Design of Item’ sections.
* All maze cells which is not occupied by walls will be filled with PacDots. Pac-Man can collect them by touching them. Each PacDot will provide 1 PacDot point and 1 energy for Pac-Man. Player can consume energy to accelerate the Pac-Man temporarily.
* In the maze, the Pac-Man moves in a speed which is decided by difficulty of current stage of game. Pac-Man moves slower in a higher stage game. Player can only control Pac-Man’s direction. The objective of player is to collect enough PacDot points and reach the exit of the maze. Players should make use of items and boost skill to achieve the objective. Details about controls of Pac-Man and interactions between Pac-Man and game objects will be discussed in ‘Design of Pac-Man’ section.
* When a player completes a stage, a summary page will be displayed. In this page, player will be rewarded some coins according to the number of PacDots collected in this stage. Coins can be used to unlock new items, increase max number of items and reduce energy cost when using boost. Then, player should choose to continue or exit. If player chooses to exit, game scene will return menu interface. If player chooses to continue, number of stage will be increased and player enters next game loop.
  1. **Design of MazeManager**

In each stage of game, a MazeManager will be used to manage a maze in the game environment. It will firstly generate a maze using a maze generation algorithm and then manage all items and ghosts in that maze. Before discussing the MazeManager, a Maze class which is used to describe a maze will be discussed first.

* **Maze**

A Maze object will contain the map information of the maze. Information of all items and ghosts will also be stored in that Maze object.

|  |
| --- |
| **Maze** |
| width: int  height: int  mazeCells: GameObject[width][height]  listOfItems: list<Item>  listOfGhosts: list<Ghost> |
| addItem(Item,Vector3): void  removeItem(Item): void  addGhost(Ghost,Vector3): void  removeGhost(Ghost): void |

* A maze consists of width \* height maze cells and each maze cells is 1\*1\*1 in size. In each maze cell, there can be empty or a wall. Information about walls in the maze is stored in the mazeCells. MazeCells[i][j] means the maze cell whose x coordinate is i and y coordinate is j. If there is an empty cell, mazeCells[i][j] equals null and if there is a wall, mazeCells[i][j] will be a GameObject that represents a wall.
* A valid maze should at least satisfy following requirements:

1. There should be at least one path between any two points in the maze
2. Each position except walls in the maze should be reachable
3. Pac-Man can only move inside the maze
4. The width of the path should be 1 unit (width of Pac-Man)

* All items and ghosts that placed in the maze will be added to an item list and a ghost list. An Item or Ghost can be added to the maze after providing a valid position and GameObject entity
* **MazeGenerator**

The job of a MazeGenerator is divided into two stages: generate a maze and place items and ghosts in the maze. The structure of MazeGenerator class is shown in below table.

|  |
| --- |
| **MazeGenerator** |
| targetMaze: Maze |
| generateMazeUsingRecursiveDivision(int,int): Maze  generateMazeUsingDepthFirst(int,int): Maze  generateMazeUsingRandomPrim(int,int): Maze  removeDeadEnd(Maze): void  placeItemsAndGhostInMaze(): void |

Three different maze generation algorithm will be implemented in the MazeGenerator. They will be used in turn to generate mazes with different styles and difficulties. Usually, the order of algorithms that will be used is Recursive Division, Depth-First and then Random Prim. After all these three algorithms have been used, size of the maze will be increased to raise the difficulty. In first stage, the maze will be 15\*15 and every 3 stages, the width and height of the maze will increase 5. The followings are details about three algorithms.

* **Recursive Division[3]**

This algorithm begins with a m\*n maze whose cells are all empty cells. Then boundary walls are created for the maze. Call the empty space inside the boundary a “field”. The algorithm will bisect the field with a randomly positioned wall, either horizontally or vertically and add a “door” on the wall. This wall divides the original field into two new fields. For each field, bisecting will be done recursively until the filed cannot be divided. The followings are pseudo code for depth-first algorithm.

*Begin with an empty field. Create boundary walls for the maze*

*Function recursiveDivision(field) {*

*If this field cannot be divided, return*

*Choose a random position, create a list of walls, either horizontally or vertically*

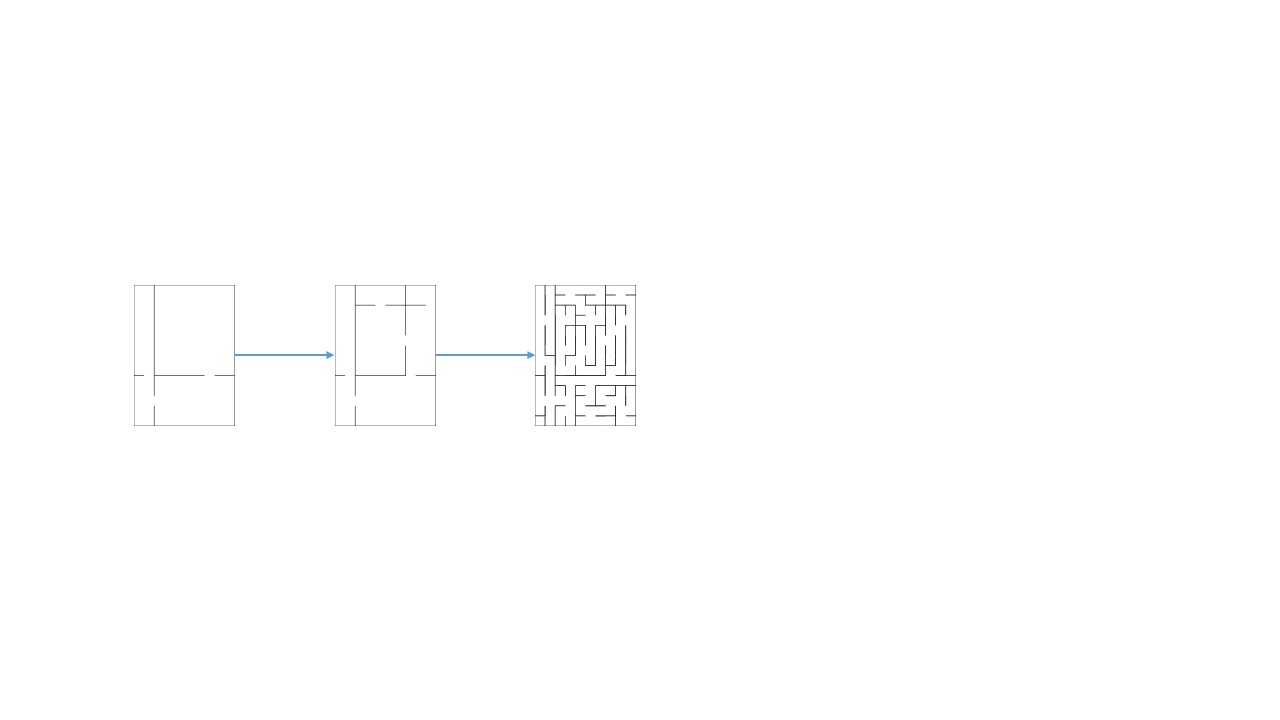
*Choose a random position on that list of walls and open a “door”*

*The original field is divided into two new fields*

*For each new field, call recursiveDivision(field)*

*Continue recursively until the maze is completed*

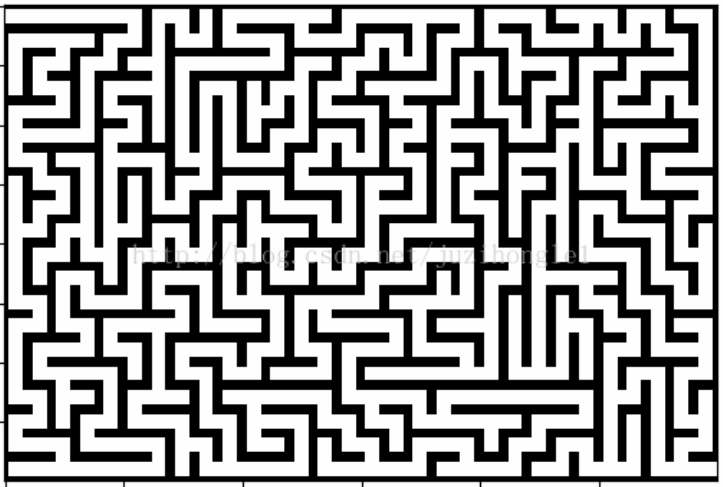
*End function*



*Figure 3.2.1 Recursive Division*

Figure 3.2.1 shows the generation of maze using recursive division. Mazes generated by this algorithm usually have fewer corners and more long, straight roads. Therefore, this kind of mazes is thought to have least difficulty.

* **Recursive backtracker(Depth-First) Algorithm [3, 4]**



*Figure 3.2.2 Sample of maze generated by recursive backtracker*

This algorithm is based on a depth-first search. Firstly, it creates a m\*n maze whose cells are all walls. Then, the algorithm starts from the start point and selects a random neighbouring cell which has not been visited. Wall between two cells will be removed. That neighbouring cell will be marked as visited and added to the stack to facilitate backtracking. This process will be continued, with a cell which has no unvisited neighbours being considered a dead-cell. If there is a dead-cell, it will backtrack through the path to find a cell with unvisited neighbour. This algorithm continues until every cell has been visited. The followings are pseudocode for depth-first algorithm.

1. *Begin with a m\*n maze whose maze cells are all walls.*
2. *Make the start point the current cell and mark it as visited*
3. *While there are unvisited cells*

*1.If the current cell has any unvisited neighbours*

*1.Choose one of the unvisited neighbours randomly*

*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*

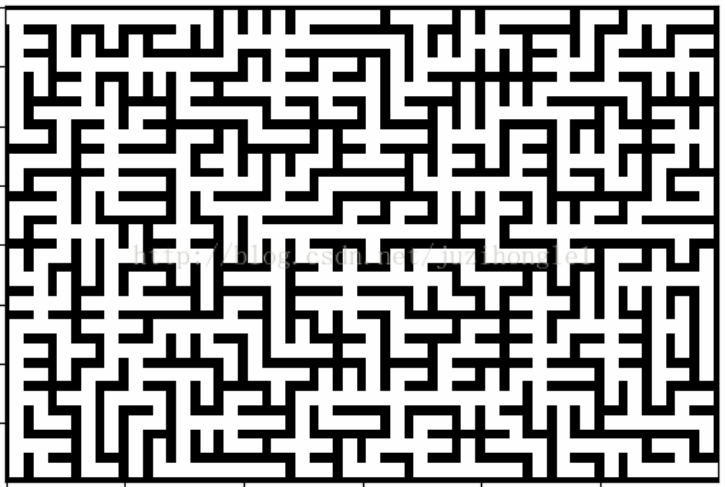
*2.Else if stack is not empty*

*1.Pop a cell from the stack*

*2.Make it the current cell*

Due to the fact that this algorithm explores as far as possible along each branch, mazes generates by this algorithm usually have a low branching factor and many long corridors, which makes this kind of mazes has a medium difficulty.

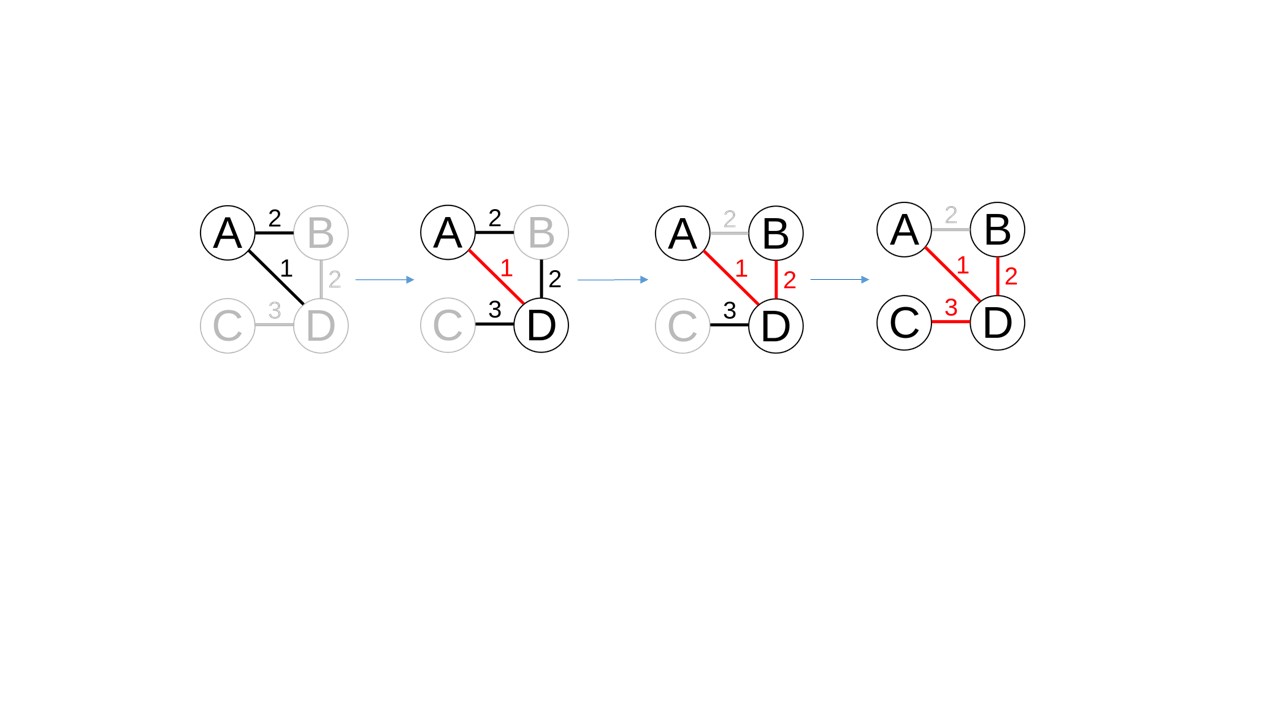
* **Randomize Prim’s algorithm[5]**



*Figure 3.2.3 Sample of maze generated by recursive backtracker*

This algorithm is based on a Prim’s algorithm. The Prim’s algorithm is used to find a minimum spanning tree for an undirected graph. Considering that the generation of a maze can be represented as the generation of a tree in a graph, Prim’s algorithm can be modified to become a suitable method for generating mazes. Following pseudo codes and chart show how original Prim’s algorithm works [8].

1. *Choose an arbitrary node from the graph, and add it to some (initially empty) set V.*
2. *Choose the edge with the smallest weight from the graph, that connects a node in V with another node not in V.*
3. *Add that edge to the minimal spanning tree, and the edge’s other node to V.*
4. *Repeat steps 2 and 3 until V includes every node in the graph.*



*Figure 3.2.4 Prim’s Algorithm*

For a maze generation, every edge has the same weight because a maze consists of many same cells. Therefore, in step 2, instead of choosing the edge with smallest weight, a random edge will be selected. Then, a new algorithm called randomize Prim’s algorithm can be used to generate mazes. The followings are pseudo code for that algorithm.

*1.Begin with a m\*n maze whose maze cells are all walls*

*2.Select a start point, break the wall on this position. Add its neighbouring walls to the wall list.*

*3.While there are walls in the wall list:*

*1.Pick a random wall from the list. If only one of the two cells that the wall divides is visited, then:*

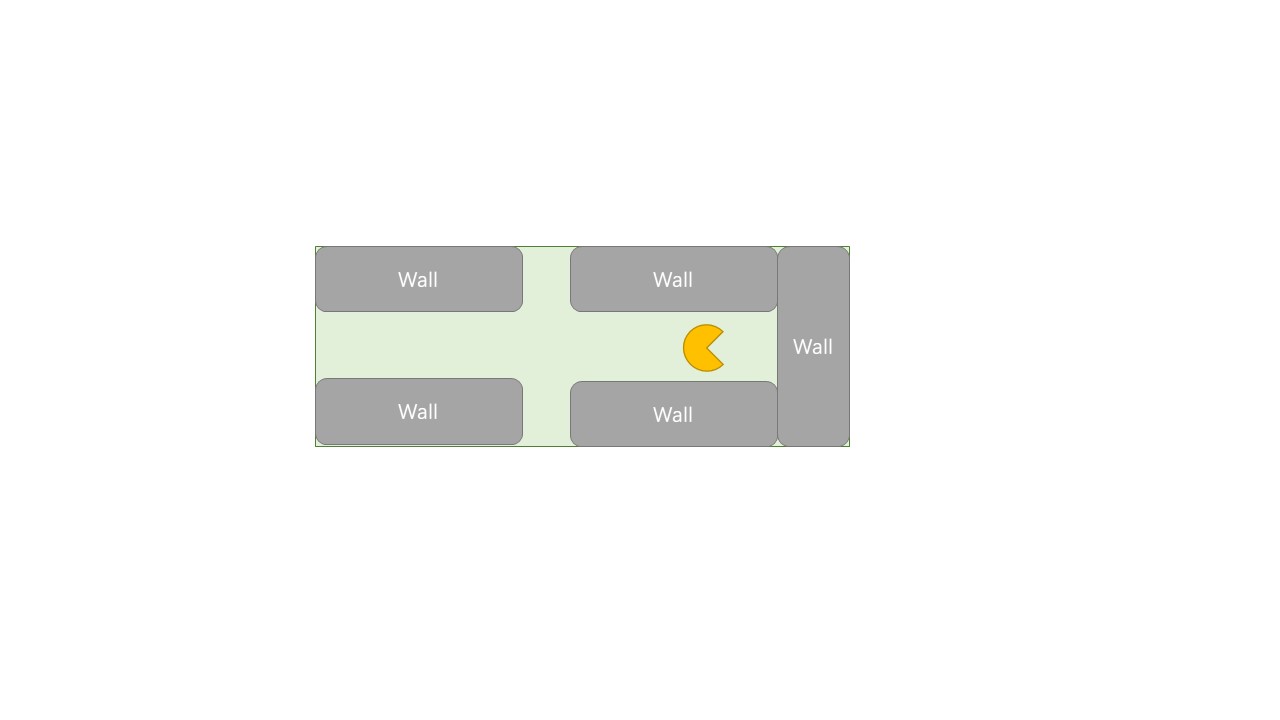
*2.Break the wall and mark the unvisited cell as part of the maze.*

*3.Add the neighbouring walls of the cell to the wall list.*

*2.Remove the wall from the list.*

In each step, randomize Prim’s algorithm chooses its target cell randomly all over the maze, while Recursive Division and Recursive backtracker select cells depending on their last selection. Therefore, compared with two other algorithms, randomize Prim’s algorithm generates most complex maze and it is the most difficult to solve.

* **RemoveDeadEnd(Maze): void**



*Figure 3.2.5 Dead End*

After a maze is generated, there will be some “dead ends” in the maze (figure 3.2.5). In some cases, PacMan may be blocked in a dead end and PacMan has no way to survive, which is not expected in the game. Therefore, all dead ends in a maze will be removed by breaking one of its three walls. RemoveDeadEnds(Maze) will be called to finish this job.

*Function removeDeadEnds(Maze) {*

*For every empty cell in the Maze {*

*If (up, right and down of the cell is blocked by walls), break it right wall*

*If (left, up and right of the cell is blocked by walls), break it up wall*

*If (up, left and down of the cell is blocked by walls), break it left wall*

*If (left, down and right of the cell is blocked by walls), break it down wall*

*End function*

* **Place Items and Ghosts**

After the construction of maze is completed, Items and Ghosts will be placed in the maze. The basic idea is to divide a maze into many 5\*5 squares. In each square, one item and one ghost will be placed in an empty maze cell. After all items and ghosts have been set up, an Ambusher ghost, which has no initial position, will be placed finally.

*Function placeItemsAndGhostInMaze() {*

*For (int i = 0; i< Maze.width; i = i + 5) {*

*For (int j = 0; j < Maze.height; j = j + 5) {*

*RandomX = random(i, i + 5)*

*RandomY = random(j, j + 5)*

*If (Maze[randomX][randomY] != wall) {*

*Place a random item which has been unlocked in Maze[randomX][randomY]*

*}*

*RandomX = random(i, i + 5)*

*RandomY = random(j, j + 5)*

*If (Maze[randomX][randomY] != wall && there is no item in Maze[randomX][randomY]) {*

*Place a random ghost in Maze[randomX][randomY]*

*}*

*} } }*

* 1. **Design of PacMan(Player)**

|  |
| --- |
| **PacMan** |
| points: int  lives: int  position: Vector3  moveVelocity: Vector3  collider: collider3D  maxNumberOfItems: int  itemsList: List<Item>  nextMovementDir: Vector3  energyLimit: int  energy: int |
| getUserInput(): void  changeDirection(Vector3): void  onCollisionEnter(Collision): void  useItem(int): void  useEnergyToAccelerate(int): void |

In this game, player will control and navigate the Pac-Man through the maze. The control of the Pac-man consists of three aspects: movement, interaction with game objects and use of items and energy.

* **Movement**
* The attribute “position” indicates the location of Pac-Man in the game environment. It is represented as a Vector3 object. Vector3 in Unity is a class for representation of 3D vectors and points, which is suitable for the position information of game objects [9].
* The attribute “moveVelocity” indicates both speed and direction of movement of Pac-Man. It is also represented as a Vector3 object.
* In the game, Pac-Man will move at a speed that controlled by the game manager. Speed of the Pac-Man will be determined according to current number of levels in order to provide different difficulties for player. In a higher level game, Pac-Man move at a lower speed. Player can only change its move direction or consume energy to accelerate temporarily. *getUserInput()* and *changeDirection(Vector3)* function will be placed under a *FixedUpdate()* function. The *FixedUpdate()* in Unity is called every fixed framerate by the Pac-Man object [10]. Therefore, the Pac-Man can keep getting operations from users and change its direction when there is no wall at that direction. The followings are pseudo code for movement of Pac-Man.

*Function FixedUpdate() {*

*Move with current velocity*

*getUserInput()*

*changeDirection(Vector3)*

*}*

*Function getUserInput() {*

*If ‘W’ is entered then nextMovementDir = new Vector3(0,1,0)*

*If ‘S’ is entered then nextMovementDir = new Vector3(0, -1,0)*

*If ‘A’ is entered then nextMovementDir = new Vector3(-1,0,0)*

*If ‘D’ is entered then nextMovementDir = new Vector3(1,0,0)*

*}*

*Function changeDirection(Vector3) {*

*If (nextMovementDir==(0,1,0) && noWallOnTopOfPacMan) then change direction to up*

*If(nextMovementDir==(0,-1,0) && noWallOnBottomOfPacMan) then change direction to bottom*

*If (nextMovementDir==(-1,0,0) && noWallOnLeftOfPacMan) then change direction to left*

*If (nextMovementDir==(1,0,0) && noWallOnRightOfPacMan) then change direction to right*

*}*

* **Interactions with game objects**
* The interaction among Pac-Man and game objects are mainly implemented using collider. Collider3D, a Unity component which is used to detect collision between game objects, will be attached to all game objects in this game. Colliders of walls are set to be static so they will not move even if a collision happens. When Pac-Man collides with Ghosts, Pac-Man will be destroyed and loses one life. When Pac-Man collides with an Item, item will be added to itemsList and player can use it later. If Pac-Man touches a PacDot, it will be rewarded some points and energy, then the PacDot will be destroyed. In Unity, when a collision happens, *OnCollisionEnter(Collision)* will be called automatically by the object that contains a collider component [11]. The followings are pseudo code for interactions of Pac-Man.

*Function OnCollisionEnter(Collision) {*

*If (Collision.collider.type==wall) then do nothing*

*If (Collision.collider.type==ghost) {*

*lives = lives -1*

*if(lives > 0) then respawn the Pac-Man*

*}*

*If (Collision.collider.type==item && itemsList is not full) {*

*add item to list*

*destroy the item*

*}*

*If (Collision.collider.type==PacDot) {*

*If (energy < energyLimit), then energy = energy + 1*

*points = points + 1*

*destroy the PacDot*

*}*

*}*

* **Use items and energy**
* Pac-Man can pick items in the maze and store it in an item list. At first, a Pac-Man can have at most 1 item at the same time. Player can consume coins to upgrade its maximum number of items after completing each stage. Details about upgrading has be discussed in ‘Design of GameManager’ section. If there is available item in the list, player can use these items by press number keys. Functions of items will be discussed in ‘Design of Items’ section.

*//Example pseudo code for using of Items*

*Function Update(){*

*If (a number key is pressed && there is an item in the list) {*

*Use that item*

*}*

*}*

* Pac-Man can charge its energy by collecting PacDots in the maze. Then player can press ‘Space’ button to consume energy and accelerate the movement of Pac-Man temporarily. The increment of speed and the number of energy that will be consumed will be determined in implement stage.

*Function Update() {*

*If (‘Space’ button is presses and there is enough energy){*

*MoveSpeed = moveSpeed + speedIncrement*

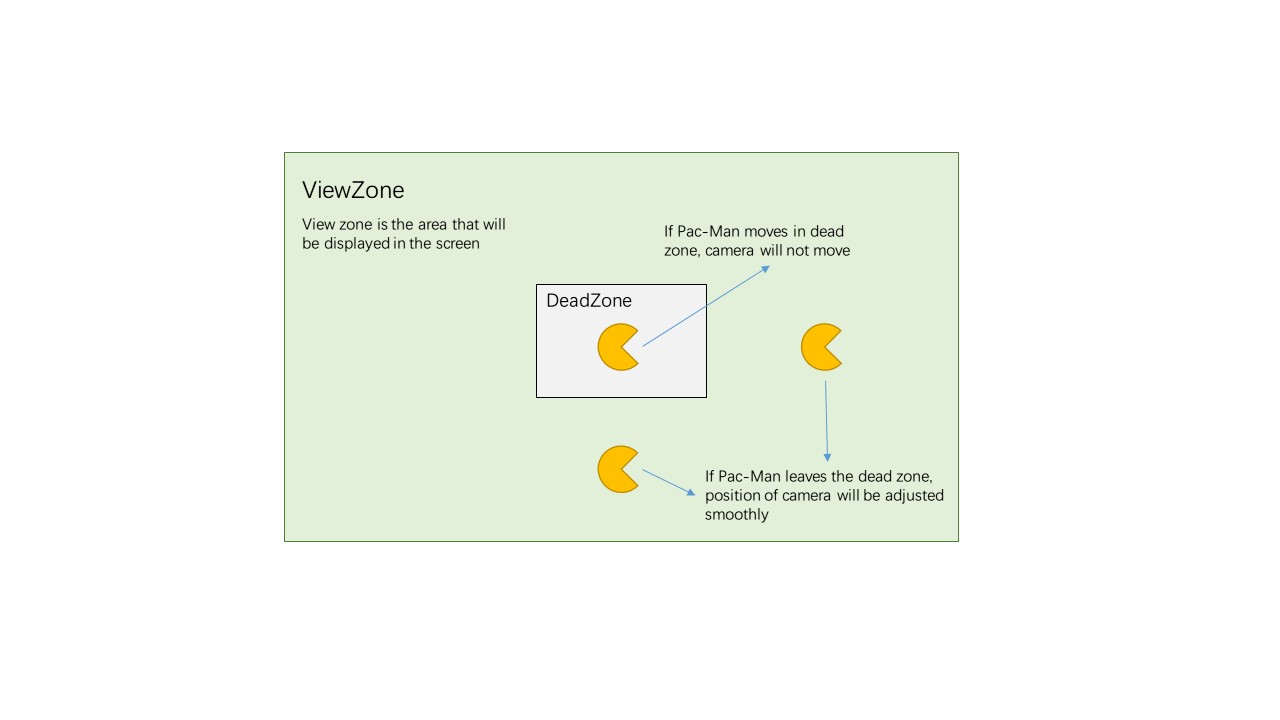
*}*

*}*

* 1. **Design of Camera**

|  |
| --- |
| **Camera** |
| position: Vector3  viewSize: float  target: Vector3  widthOfZone: float  heightOfDeadZone: float |
| moveToPosition(Vector3): void  changeViewSize: void |

The view of camera is divided into two parts: view zone and dead zone. The structure of camera view is shown in the figure3.3.1. This kind of design is to provide a smooth movement of camera. Compared with the camera that follows the target, this kind of camera reduces the shaking of view and players can be more focused on the game.



*Figure 3.4.1 Camera View*

The followings are pseudo code for camera.

*Function Update() {*

*If target is inside the dead zone {*

*do nothing*

*} else if target leaves the dead zone {*

*move towards the direction of target*

*}*

*}*

* 1. **Design of Items**

Some items will be placed in the maze. Pac-Man can pick them up and use them later in the game. Each item can provide a unique ability or bonus for Pac-Man, which is helpful when solving the maze. Prior to this document, five different items have been designed and all of them extends from a parent class: Item.

|  |
| --- |
| **Item** |
| Name: String  position: Vector3  trigger: Collider3D  unlocked: Bool |
| setPosition(): void  OnTriggerEnter(Collider): void |

* **Item**
* The position of an item in the maze is represented as a 3D vector
* Each item has a trigger attribute. Trigger is a special kind of collider and it does not behave as s solid object and will allow other colliders to pass through. A trigger will call the *OnTriggerEnter(Collider)* function when a collider enters its space [12]. Therefore, a trigger is a good way for the detection of interactions between Pac-Man and items.

*Function OnTriggerEnter(Collider) {*

*If(Collider is Pac-Man && Pac-Man’s items list is not full) {*

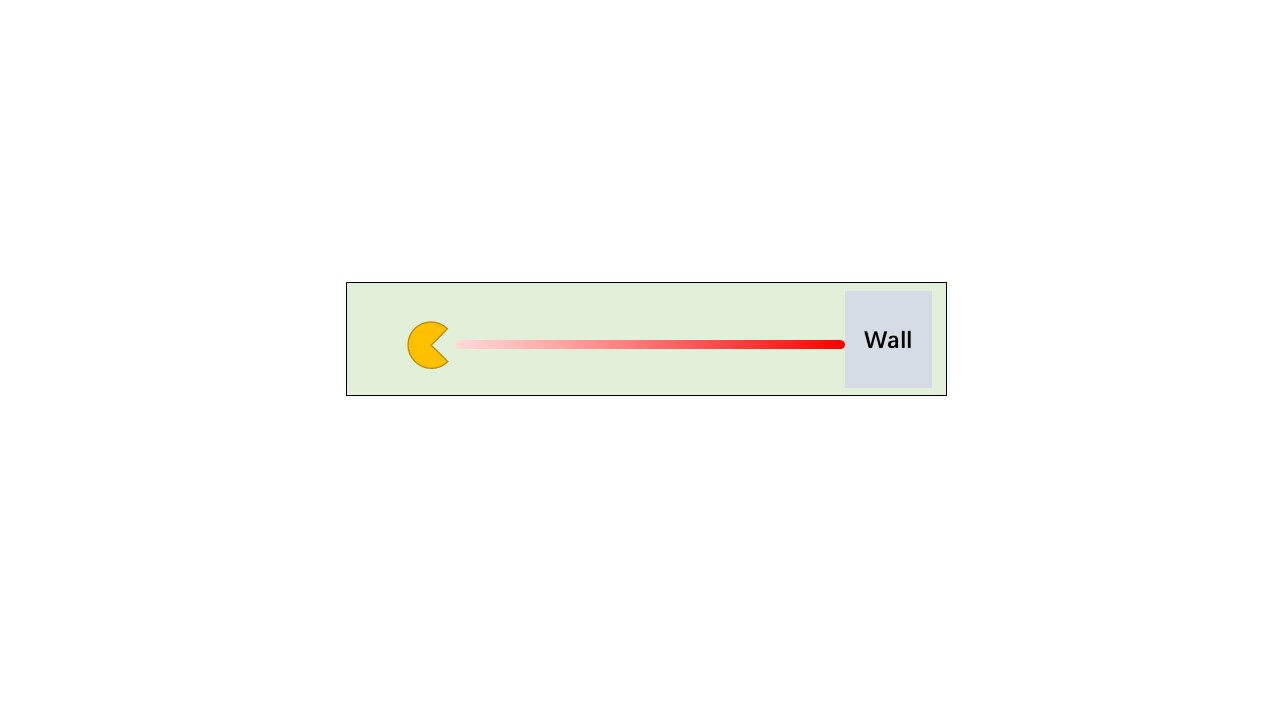
*Add item to Pac-Man’s items list*

*Remove item from the maze*

*}*

*}*

* All items are locked if it is the first time playing this game. Players should earn coins by playing the game and consume coins to unlock items.
* **Laser**



When player uses a laser item, a laser will be cast from the head of Pac-Man, in direction of Pac-Man’s movement. When the laser touches a ghost, it will disable that ghost for 15 seconds. The laser will last 5 seconds and it cannot get through the wall. A laser item is implemented by a Laser class.

|  |
| --- |
| **Laser extends Item** |
| LaserEntity: 3Dmodel |
| castLaser(): void |

* This laser item extends Item class and has a new function *castLaser()*. This function is called when player chooses to use a laser item. Raycast() function will be used to cast a ray from Pac-Man, in direction of Pac-Man’s movement. The following pseudo codes show how a laser behave.

*Function castLaser() {*

*RayCast(originPoint, Direction)*

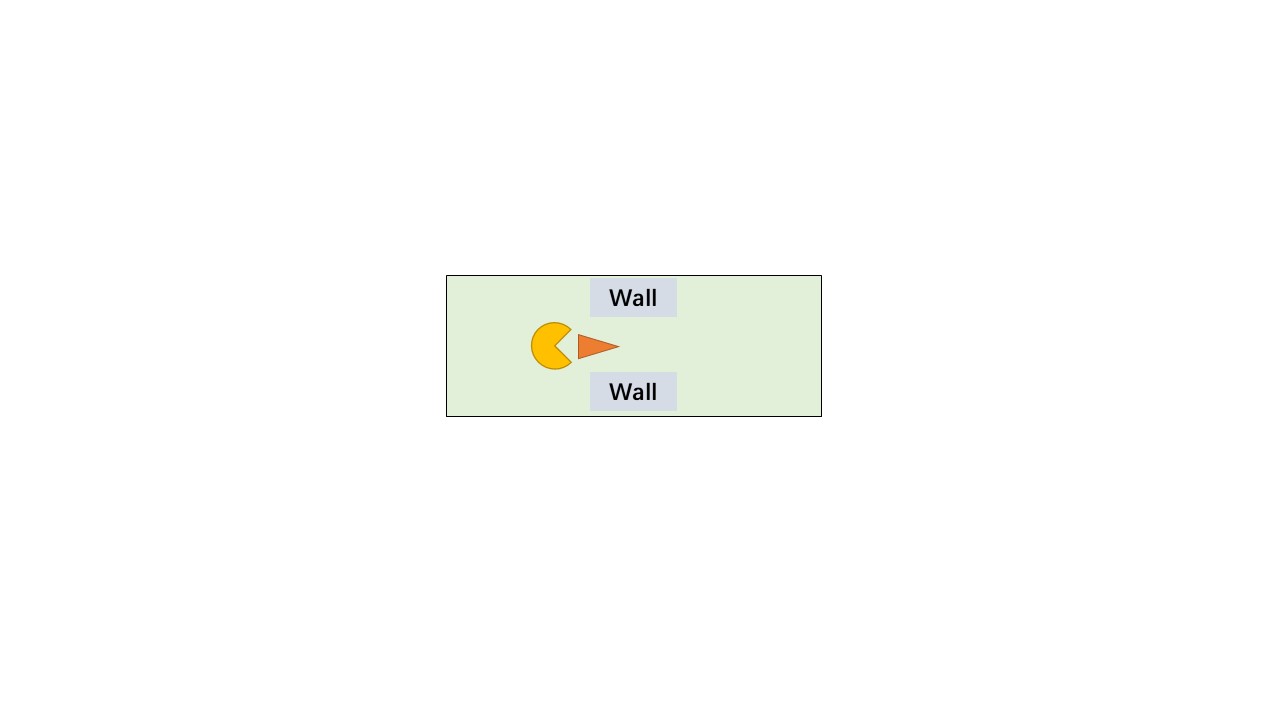
*If ray hits a wall, then the laser will stop in front of the wall*

*If ray hits a ghost, then the ghost will be disabled for 15 seconds*

*After 5 seconds, the laser object will be destroyed.*

*}*

* **Wall-breaker**



Using this item will generate a wall-breaker object in front of the Pac-Man. When the breaker colliders with a wall, both of wall and breaker will be destroyed. If the breaker colliders with a ghost, the breaker will be destroyed and the ghost will be disabled permanently.

|  |
| --- |
| **Wall-breaker extends Item** |
| BreakerEntity: 3Dmodel |
| OnColliderEnter(Collider): void |

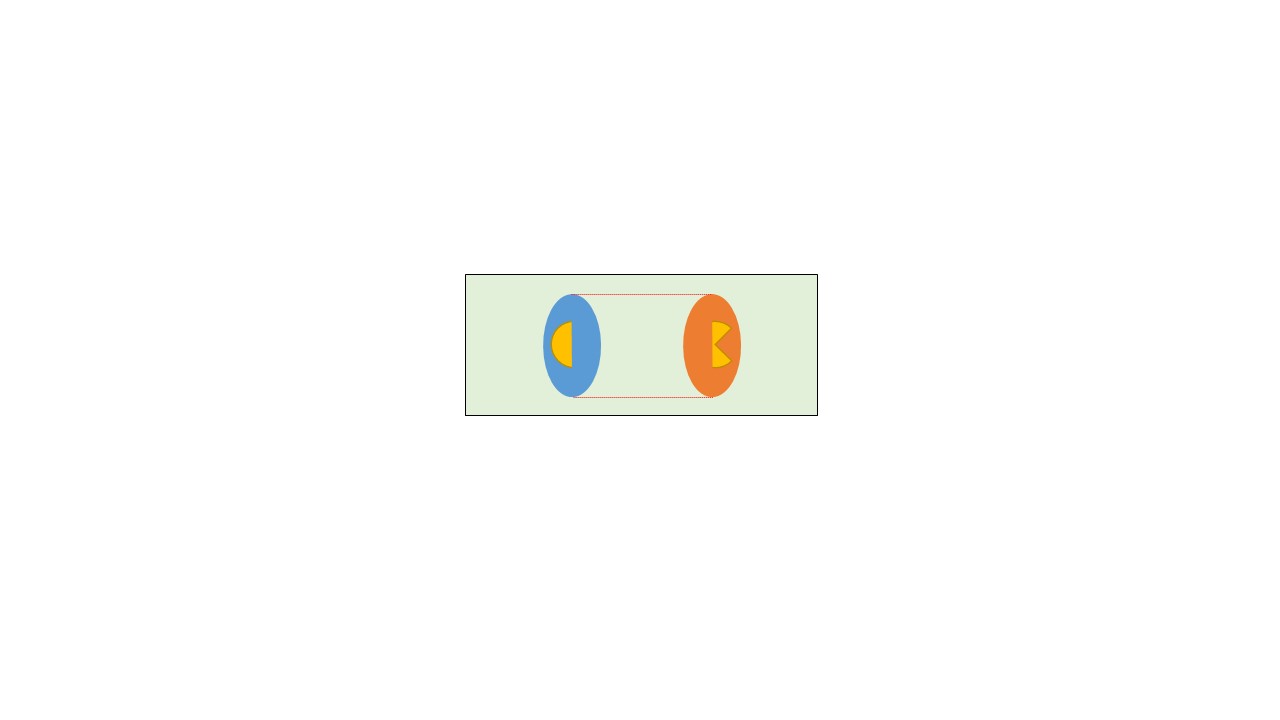
*Function OnColliderEnter(Collider) {*

*If Collider.type == wall, then destroy the wall and this breaker*

*If Collider.type == ghost, then destroy this breaker and disable the ghost permanently*

*}*

* **Portal**



Portals should appear in pairs in the maze so each portal should link with another portal. When the player first time uses the portal item, the first portal will be created at the position of the Pac-Man. Then, player can use the item again to create another portal and two portals will be linked. When a Pac-Man pass through one of the two portals, it will be transported to the position of the linked portal.

|  |
| --- |
| **Portal extends Item** |
| PortalEntity: 3Dmodel  linkedPortal: Portal |
| transportPacMan(): void |

*If player uses the item for the first time, then create a portal at the position of the Pac-Man*

*If (player uses the item for the second time && current position != position of first portal) {*

*Create another portal and link two portals*

*}*

*Function Update() {*

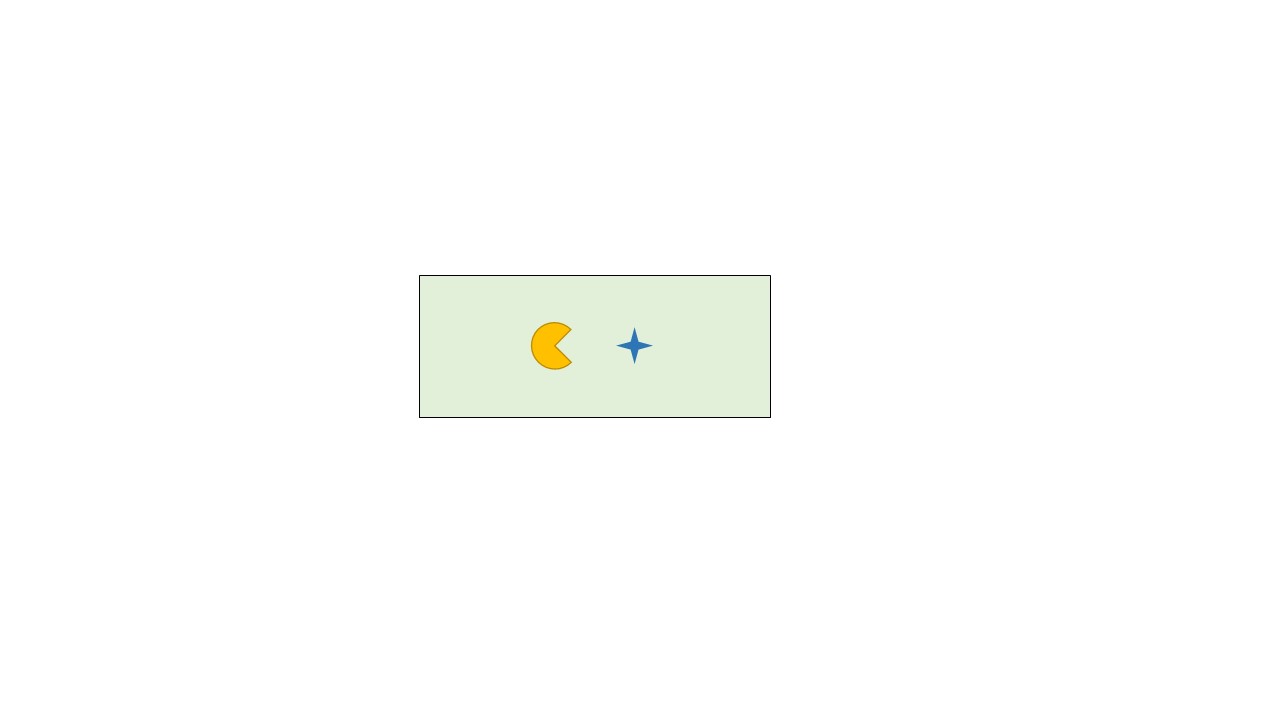
*If (Pac-Man arrives at a portal and the portal has a linked portal) {*

*Transport the Pac-Man to the position of linked portal*

*}*

*}*

* **Energy pellet**



Pac-Man can get 100 points of energy supply by using the energy pellet. This item allows number of energy to exceed player’s energy limit temporarily. Energy can be used to boost the Pac-Man.

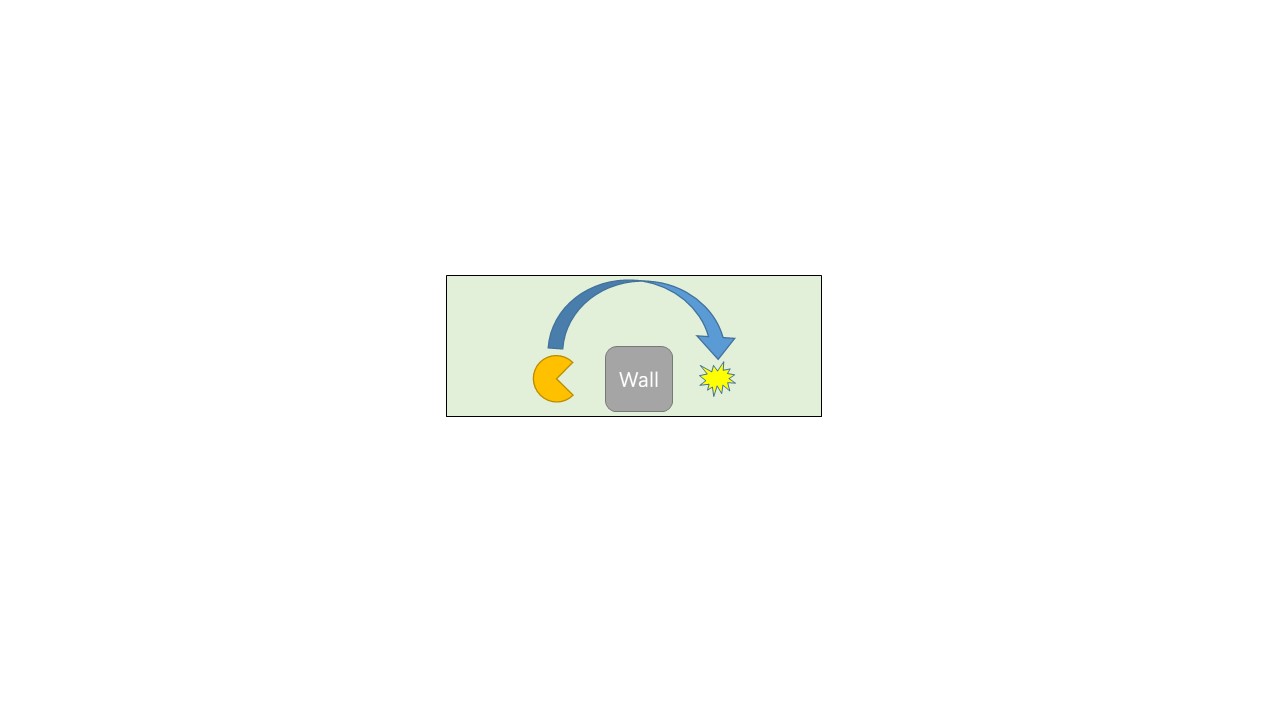
|  |
| --- |
| **EnergyPellet extends Item** |
| pelletEntity: 3Dmodel |
| chargePacMan(): void |

*Function chargePacMan() {*

*If player uses energy pellet, then Pac-Man.energy = Pac-Man.energy + 100*

*}*

* **Grenade**



Player can throw a grenade over a wall and attack a ghost. The grenade will explode when it collides with ghost and disable that ghost for 15 seconds. If it does not touch any ghost, it will stay on the ground for 3 seconds and Pac-Man can recollect it by touching it during that 3 seconds. It will explode if it is not collected after 3 seconds.

|  |
| --- |
| **Grenade extends Item** |
| grenadeEntity: 3Dmodel  explosionAnimation: animation |
| OnColliderEnter(Collider): void  Explode(): void |

*//Example of pseudo code for grenade item*

*Grenade.position = PacMan.position + PacMan.direction\*2*

*While(grenade stays on the ground for less than 3 seconds) {*

*If(grenade collides with any ghost), then explore()*

*If(grenade collides with PacMan), then PacMan collect the grenade item*

*}*

*If(grenade survives for more than 3 seconds), then explore()*

*Function explode() {*

*PlayExplosionAnimation*

*Disable all ghosts within one unit for 15 seconds*

*If PacMan is hit by the explosion, then PacMan loses one life*

*Self.destory()*

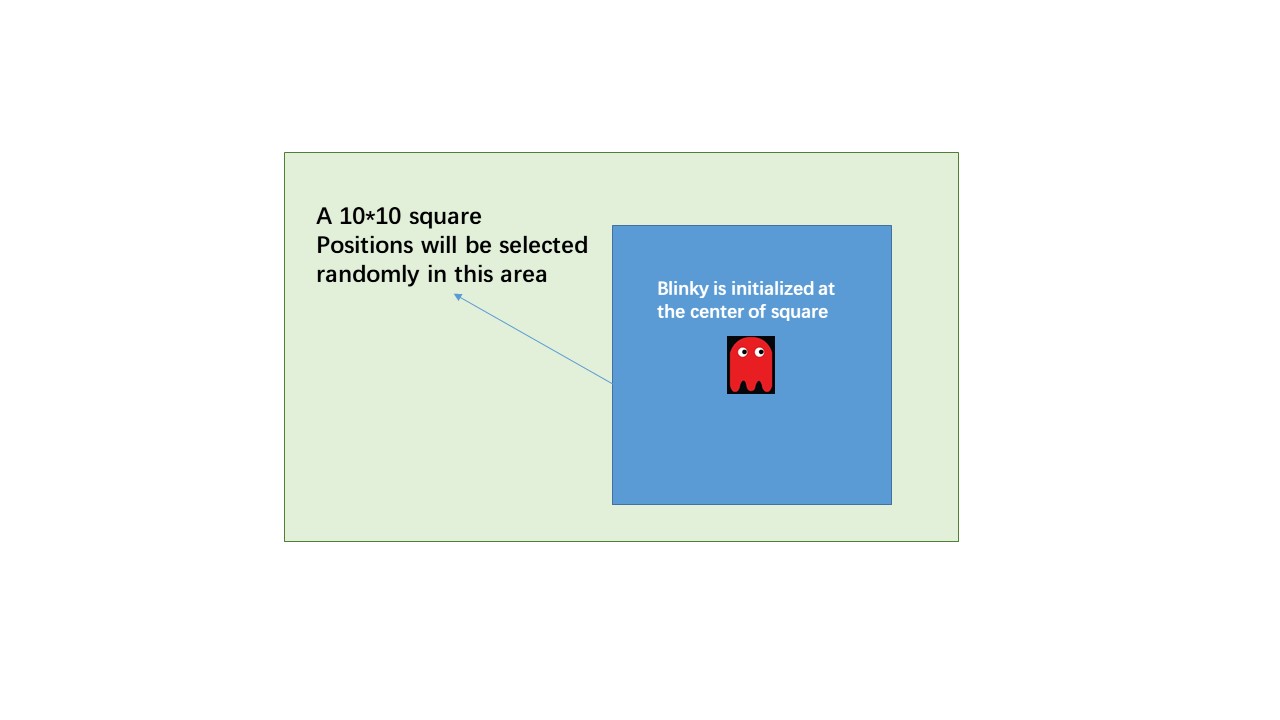
*}*

* 1. **Design of Ghosts**

This section will discuss several kinds of ghost that moving in the maze and interact with the Pac-Man. Currently, 5 different types of ghost have been designed, including Blinky, Chaser, Ambusher and Thief. Each ghost has its unique behaviour logic and aim of ghosts except Thief is to defeat Pac-Man by colliding with it. The aim of Thief is to steal PacDots from the maze to prevent Pac-Man from collecting enough points. All kinds of ghost inherit a Ghost class.

|  |
| --- |
| **Ghost** |
| position: Vector3  velocity: Vector3  ghostCollider: collier3D  nextPosition: Vector3  target: PacMan |
| OnColliderEnter(Collider): void  MoveToPosition(Vector3): void  SetNextPosition(Vector3): void |

* **Blinky**



*Figure 3.6.1 Blinky*

Blinky is the simplest kind of ghost. It moves among a list of predetermined target positions. When a Blinky is initialised, it will select 10 positions randomly. All these positions should inside a 10\*10 square whose centre is the Blinky (figure 3.6.1). After the game is started, Blinky will move between those positions in the list randomly. A Blinky class inherits Ghost and has one extra attribute and method.

|  |
| --- |
| **Blinky extends Ghost** |
| targetPositionList: List<Vector3> |
| ChooseTargetPosition(): Vector3 |

*//Example of pseudo code for Blinky*

*Choose 10 number from (position.x - 5, position.x + 5) as X coordinate of target positions*

*Choose 10 number from (position.y - 5, position.y + 5) as Y coordinate of target positions*

*Add those positions to target position list*

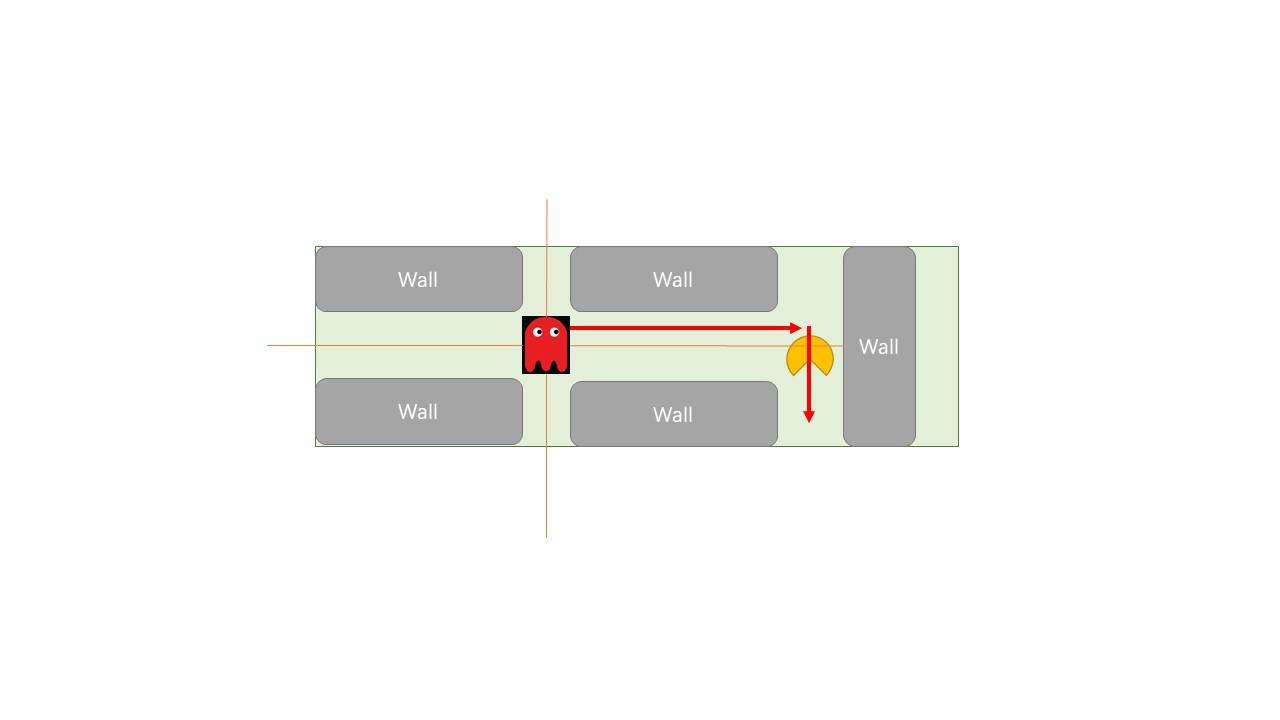
*While(true) {*

*Choose a position from the list, move to the position*

*After reaching the target position, choose another one from the list*

}

* **Chaser**



*Figure 3.6.2 Chaser*

This kind of ghost will detect the appearance of PacMan in four directions. It a PacMan is found, Chaser will record its position and direction of movement. Then Chaser will move to that position and move along the direction of PacMan. A Chaser’s speed will reduce while it is moving and it needs rest to recover its speed. Its maximum speed will range from 2.5 times of speed of PacMan to 1.5 times of speed of PacMan. The following is structure of Chaser class.

|  |
| --- |
| **Chaser extends Ghost** |
| speedMultiple: float  targetDirection: Vector3 |
| detectPacMan(): void  setSpeedMulitple(): void |

* *Raycast()* will be used to detect the appearance of PacMan. *Raycast()* casts a ray to a direction and reports when ray hits a collider [13]. *detectPacMan()* will cast four rays to directions including up, down, left and right. If any ray hits a PacMan, the position and direction of PacMan’s movement will be recorded.
* The speed of a Chaser is adjusted by *setSpeedMulitple()*. When a Chaser is moving, in each second, speedMulitple will reduce 0.1. the speedMulitple will recover while a Chaser is static. The speedMultiple ranges from 2.5 to 1.5 and speed of a Chaser equals speed of PacMan \* speedMultiple.

*//Example of pseudo code for Chaser*

*While(True) {*

*Cast rays to up, down, right, left four directions*

*If(ray hit a PacMan) {*

*TargetPosition = PacMan.position, targetDirection = PacMan.direction*

*MoveToPosition(targetPosition)*

*After arriving the position, turn to the target direction and move ahead*

*}*

*If(Chaser hit a wall && targetDirection is null), then stop and scan PacMan again*

*}*

*Function setSpeedMulitple() {*

*While(True) {*

*If(Chaser is moving && speedMulitple>=1.5) {*

*After each second, speedMulitple = speedMultiple – 0.1*

*} else if(Chaser is static && speedMulitple<=2.5) {*

*After each second, SpeedMulitple = speedMultiple + 0.1*

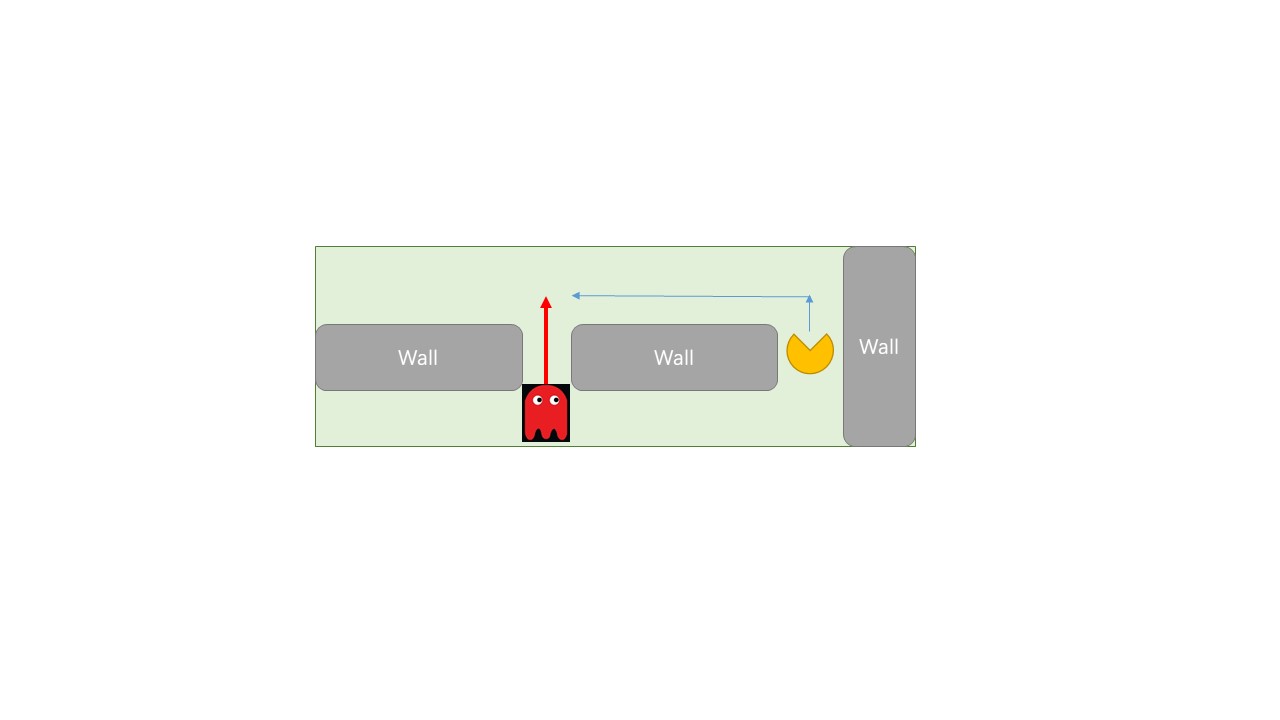
*}*

*Self.speed = PacMan.speed \* speedMultiple*

*}*

*}*

* **Ambusher**

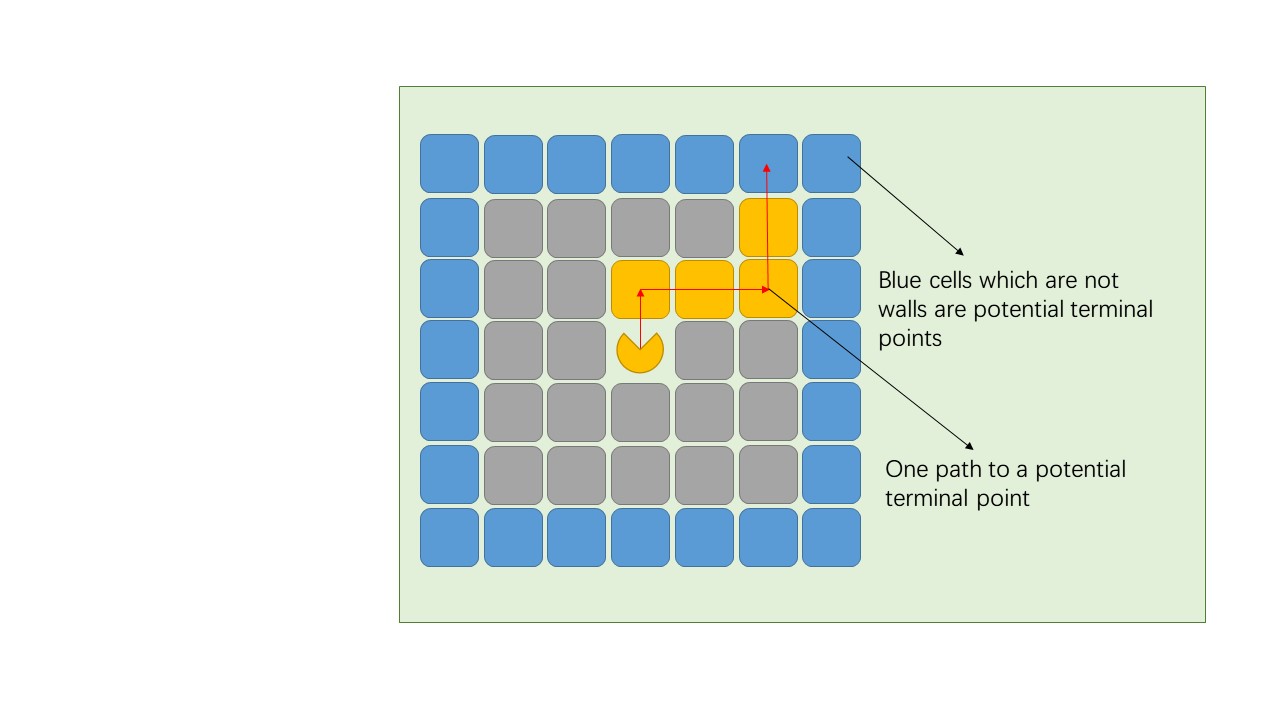


*Figure 3.6.3 Ambusher*

An Ambusher will not appear in the maze at the beginning of each game. In every 30 seconds, the Ambusher will try to predict the movement of the PacMan and appear near the place that the PacMan might be. To implement this feature, some extra attributes and methods will be applied in Ambusher class.

|  |
| --- |
| **Ambusher extends Ghost** |
| listOfPotentialPaths: list<Vector3[ ]>  predictedPath: Vector3[ ] |
| getAllPotentialPaths(): list<Vector3[ ]>  getPredictedPath: Vector3[ ]  instantiate(Vector3): void  aStarSearch(Vector3): Vector3[ ] |

* *GetAllPotentialPaths()* will firstly draw a square whose width is 7 units and centre is the PacMan. Then, all boundary cells which are not wall will be marked as potential terminal point. A potential terminal point indicates a position that PacMan may arrive recently (figure 3.6.4). Finally, an A\* algorithm will be applied to compute all paths which start from PacMan to every potential terminal point.



*Figure 3.6.4 Potential terminal points*

*//Example pseudo code for getAllPotentialPaths()*

*Function getAllPotentialPaths() {*

*List listOfPotentialTerminalPoints = list<Vector3>*

*For (int i =self.position.x – 3; i<= self.position.x + 3; i ++){*

*If(cell at (i, self.position.y + 3) is not wall), add it to listOfPotentialTerminalPoints*

*If(cell at (i, self.position.y - 3) is not wall), add it to listOfPotentialTerminalPoints*

*}*

*For (int i =self.position.y – 3; i<= self.position.y + 3; i ++){*

*If(cell at (self.position.x + 3, i) is not wall), add it to listOfPotentialTerminalPoints*

*If(cell at (self.position.x - 3, i) is not wall), add it to listOfPotentialTerminalPoints*

*}*

*For each point in listOfPotentialTerminalPoints{*

*Path = aStarSearch(point)*

*listOfPotentialPaths.add(Path)*

*}*

*}*

*Function aStarSearch(Vector3 end){*

*Vector3[ ] path*

*Put position of PacMan into a OPEN list, OPEN list stores unvisited cells*

*A CLOSE list will be used to store visited cells*

*F(n) of the each point = actual distance from the start point to this point + (abs(end.x – startPoint.x) + abs(end.y – startPoint.y))*

*While(OPEN != Null){*

*CurrentPoint = the point which has the least f(n)*

*If(currentPoint == end), then break*

*For (every non-wall neighbour cell X of current cell){*

*Compute its f(X)*

*If(this cell is in OPEN){*

*If(new f(X) < old f(n) in the OPEN list){*

*Set n as the parent of X*

*Update f(n) in the OPEN list*

*}*

*}else if (this cell is in CLOSE) {*

*Continue*

*}else if ( X is not in OPEN or CLOSE) {*

*Set n as the parent of X*

*Compute f(X)*

*Add X to OPEN list*

*}*

*}*

*Remove n from OPEN list and add it to CLOSE list*

*}*

*path = points that move along parent cell from end point to the start point*

*Return path*

*}*

* *GetPredictedPosition()* will firstly calculate the expected profit for each path in the potential path list. The expected profit of a path = number of PacDots in the path + number of items in the path \* 10. The path which has most expected profit is the predicted path and the terminal point of the path is the predicted destination of the PacMan. Then the Ambusher instance will be instantiated at the position of the predicted destination. If an Ambusher instance does not meet the PacMan in 5 seconds, it will disappear from the maze and it needs 30 seconds to prepare its next ambush.

*//Example of pseudo code for behaviour of an Ambusher*

*While(True) {*

*Get all potential paths of the PacMan*

*For each path in potential paths list {*

*Expected profit = number of PacDots in path + 10 \* number of Items in path*

*}*

*Predicated path = potential path with most profit*

*Instantiate the Ambusher instance at the terminal point of the predicated path*

*The Ambusher instance will survive for 5 seconds*

*Destroy the Ambusher instance*

*Disable the Ambusher object for 30 seconds*

*}*

* **Thief**

The final kind of ghost is called Thief. This kind of ghost will be activated when the distance between it and the PacMan is less than 10 units. After it is activated, it will try to avoid PacMan instead of hunting it. This kind of ghost will also collect PacDots when it collides with them. PacMan can recollect PacDots that stolen a Thief by colliding with it or hitting it with any aggressive item (Laser, Wall-breaker or grenade). To implement this Ghost, a Thief will inherit a Ghost and has some extra attributes and methods.

|  |
| --- |
| **Thief extends Ghost** |
| numberOfStolenPacDots: int  moveDirectionSequence: Vector3[4] |
| Override OnColliderEnter(Collider): void  decideMoveDirection(): Vector3[4] |

* *OnColliderEnter(Collider)* function in Thief will be overridden. When it collides with a PacDot, it will collect the PacDot. If it collides with PacMan, the Thief will be destroyed and give all stolen PacDots to PacMan.
* *DecideMoveDirection()* will create a sequence of direction including up, down, left and right according to the position of Thief and PacMan. Thief can avoid PacMan by moving in order of the sequence. For example, if its sequence is [up, left, down, right]. The Thief will firstly try to move up. If up is blocked by walls, it will try to move left, then down, then right. The algorithm for this function will be shown below and it will ensure that Thief will always try to move away from PacMan.

*//Example of pseudo code for behaviour of a Thief*

*Override OnColliderEnter(Collider){*

*If (Collider.type == PacDot){*

*NumberOfStolenPacDots ++*

*Destory that PacDot*

*}Else if (Collider.type == PacMan) {*

*Self.destroy()*

*PacMan.points = PacMan.points + numberOfStolenPacDots*

*PacMan.energy = PacMan.energy + numberOfStolenPacDots*

*}*

*}*

*Function decideMoveDirection() {*

*Vector3 relativePosition = PacMan.position – Thief.position*

*If(relativePosition.x>=0&&relativePosition.y<=0&&relativePosition.x.abs< relativePosition.y.abs) {*

*MoveDirectionSequence = [up, left, right, down]*

*}*

*If(relativePosition.x>=0&& relativePosition.y<=0&& relativePosition.x.abs>relativePosition.y.abs) {*

*MoveDirectionSequence = [up, left, right, down]*

*}*

*If(relativePosition.x>=0&&relativePosition.y<=0&& relativePosition.x.abs> relativePosition.y.abs) {*

*MoveDirectionSequence = [up, left, right, down]*

*}*

*If(relativePosition.x>=0&&relativePosition.y<=0&& relativePosition.x.abs< relativePosition.y.abs) {*

*MoveDirectionSequence = [up, left, right, down]*

*}*

*If(relativePosition.x<=0&&relativePosition.y<=0&& relativePosition.x.abs<relativePosition.y.abs) {*

*MoveDirectionSequence = [up, right, left, down]*

*}*

*If(relativePosition.x<=0&&relativePosition.y<=0&& relativePosition.x.abs>relativePosition.y.abs) {*

*MoveDirectionSequence = [right, up, down, left]*

*}*

*If(relativePosition.x<=0&&relativePosition.y>=0&& relativePosition.x.abs< relativePosition.y.abs) {*

*MoveDirectionSequence = [down, right, left, up]*

*}*

*If(relativePosition.x<=0&&relativePosition.y>=0&& relativePosition.x.abs> relativePosition.y.abs) {*

*MoveDirectionSequence = [left, down, up, left]*

*}*

*}*

*Function Update() {*

*DecideMoveDirection()*

*Move in direction MoveDirectionSequence[0]*

*If moveDirection is blocked by walls,, moveDirection = MoveDirectionSequence[1]*

*If moveDirection is blocked by walls,, moveDirection = MoveDirectionSequence[2]*

*If moveDirection is blocked by walls,, moveDirection = MoveDirectionSequence[3]*

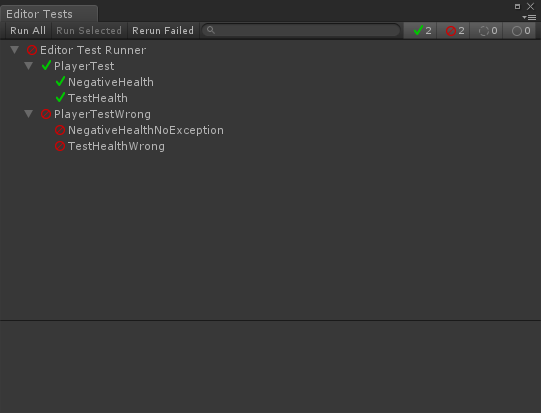
*}*

1. **Evaluation Design**

In evaluation stage will be divided into three parts: unit tests, performance tests and player’s feedback. Unit tests will be completed using Unity Test Runner. Unity performance testing extension will be used to conduct the performance test. Players’ feedback will be collected using a built-in questionnaire. Players can complete this questionnaire the results will be sent to my email box.

* 1. **Unit tests**

Unit test will be done for the final product of the project. It will check the validation and function of each component in the game. Unity has a Unit Test Runner which can be used to deal with all unit tests [14]. Figure 4.1.1 shows the interface of a Unit Test Runner.



*Figure 4.1.1 Unit Test Runner*

The following table shows all tests and test data that will be checked in the final unit tests.

|  |  |  |  |
| --- | --- | --- | --- |
| **Test ID** | **Description of test** | **Test data** | **Expected Result** |
| 1 | Load of the main menu: when loadMenu(“main”) is called, the main menu should appear |  | All menu items in the main menu should appear on the screen |
| 2 | Load of “Option” menu: when loadMenu(“option”) is called, the “Option” menu should appear |  | All menu items in the option menu should appear on the screen |
| 3 | Load of “Items” menu: when loadMenu(“items”) is called, the “Items” menu should appear |  | All menu items in the items menu should appear on the screen |
| 4 | A stage of game is ready to start: when loadScene(“game”) is called, load the game environment and ready for a game |  | All UI items in a game are ready.  A MazeGenerator object is ready for this stage of game |
| 5. | Test the mazes that generated by the MazeGenerator. MazeGenerator should generate valid mazes according to number of stage. A\* algorithm is used to find a way from the start point to the end point. Then all position of walls, items and ghosts will be checked. All of them should be inside the maze and should not overlap with each other. Existence of dead ends will also be checked. | A set of numbers that represents the number of stage | All mazes should satisfy following requirements:  1.There should be at least one path between any two points in the maze can be found  2.Each position except walls in the maze should be reachable  3.Pac-Man can only move inside the maze.  4.The width of the path should be 1 unit (width of Pac-Man)  5.All items and ghosts appear on the empty cells in the maze.  6.No dead ends appear in the maze |
| 6 | Movement of PacMan. PacMan should move according to the input of player. PacMan will also be blocked by walls in a maze. The test provides a pre-generated maze and a set of player’s input. Finally, PacMan should arrive the expected position. | A pre-generated maze  A set of player’s input | The PacMan should finally arrive the expected position. |
| 7 | Interactions between PacMan and PacDots. PacMan will collide with an amount of PacDots in the test | Number of PacDots | 1. PacDots will be destroyed when colliding with PacMan 2. PacMan get 1 point when colliding with a PacDot 3. PacMan get 1 energy when colliding with a PacDot and report a “full energy” after the number of energy reaches 100 |
| 8 | Interactions between PacMan and Items.  PacMan will collide with some items in a maze | Different kinds of items | 1. If PacMan’s item box is not full, PacMan will pick items up when colliding with them 2. If PacMan’s item box is full, items will be left on the ground |
| 9 | Interactions between PacMan and Ghosts (except Thief). PacMan will collide with ghost in a maze | Different kinds of ghosts | PacMan will be destroyed when colliding with a Ghost. The number of lives will be reduced 1. If the number of lives is less than 1, Game Over will be reported |
| 10 | Interactions between PacMan and Thief. PacMan will collide with Thief in a maze | Several Thief with different number of stolen PacDots | The Thief will be destroyed. The points will be added to the PacMan according to the number of PacDots that stolen by the thief |
| 11 | Test that the PacMan arrive the exit of the maze. Only PacMan with enough points can complete the stage | A set of PacMan with different number of points | PacMan with enough points reports a Stage Clear and display a stage clear menu  PacMan without enough number of points is not allowed to complete the stage |
| 12 | Interactions between stage clear menu and player. Player can consume coins to upgrade its items box or energy. Player can also choose to continue the game or leave the game. | A set of players’ input | When player chooses to upgrade its items box or energy, if there are enough coins, consume coins to upgrade the box, otherwise a warning message will be reported  If player chooses to continue the game loop, number of stages + 1 and start next game loop  If player chooses to return main menu, destroy the game scene and display main menu |
| 13 | Test that when the number of PacMan’s lives is less than 1,a game over menu will be displayed |  | When the number of PacMan’s lives is less than 1, a game over menu will be displayed. |
| 14 | Interactions between game over menu and player. Player can choose to return to main menu or start another game | A set of players’ input | If player chooses to return to main menu, destroyed the game scene and display main menu  If player chooses to start another game, restart the game loop at stage 1 |
| 15 | Test that a laser can be used to disable a ghost. Send a laser to a ghost and the ghost should be disabled. | Player uses a laser | The ghost should be disabled for 15 seconds |
| 16 | Test that a wall-breaker can be used to break a wall. Send a wall-breaker to a wall and the wall should be destroyed. | Player uses a wall-breaker | The wall should be destroyed |
| 17 | Test that a portal can be used to transport the PacMan. Set a pair of portal in a maze and let a PacMan walk to one of it. |  | The PacMan should be transported to the position of the other portal. Any attribute except position of the PacMan should not be changed |
| 18 | Test that an energy pellet can be used to add energy to the PacMan. Set several energy pellets in a maze and let a PacMan walk to one of it. |  | The PacMan get 100 points of energy when colliding with one energy pellet. Energy that obtained by this way is allowed to exceed the energy limit. Then the energy pellet will be destroyed. |
| 19 | Test that a grenade can be used to throw over a wall and disable a ghost. PacMan is asked to throw a grenade over a wall to a ghost behind the wall. | Player uses a grenade | The grenade hits the ghost behind the wall. The ghost is disabled for 15 seconds. |
| 20 | Test that a Blinky can move according to the list of positions | A list of position | Blinky should appear within the positions that belong to the list. |
| 21 | Test that a Chaser can chase the PacMan in an expected way. A chaser is placed in a pre-generated maze and PacMan with different moving routines will then be placed in the maze | A set of PacMan which has its own moving routine and an expected routine for Chaser | For each PacMan, the movement of Chaser should satisfy the expected routine |
| 22 | Test that an Ambusher can appear in an expected position. A set of PacMan which has its pre-determined routine will be added to the maze. The ambusher should react to these PacMan | A set of PacMan which has its pre-determined routine | For each PacMan, the Ambusher should appear in an expected position. |
| 23 | Test that a Thief can collect PacDots when colliding with it. The Thief is placed in a maze that contains many PacDots. | A set of PacDots | When Thief collides with a PacDot, it will collect it and get one point. |
| 24 | Test the interactions between Thief and PacMan. | A set of Thieves that contain some points of PacDots | When PacMan collides with a Thief, the Thief will be destroyed and its points of PacDots will be added to the PacMan. |
| 25 | Test the movement of a Thief. A Thief is placed in a pre-generated maze. A set of PacMan that contains its moving routine will be added to the maze. Thief should react to these PacMan | A set of PacMan which has its pre-determined routine | For each PacMan, the Thief should appear in an expected position. |
| 26 | Test that if the “Exit” button works. | Player uses “Exit” button | The software is terminated. |

* 1. **Performance test**

Performance test will be completed using Unity Performance Testing Extension [14]. This tool can monitor the frames per second and start up time during the game.

* To pass this test, in a computer with 2.6 GHz CPU frequency, 12G RAM and GTX 860M graphics card, the average frames per second should not be below 30 and the minimum frames per second should be above 24.
* To pass this test, the maximum time that is spent on loading a stage should not exceed 5 seconds.
  1. **Players’ feedback**

A questionnaire will be designed to get players’ idea about this game. All participants are voluntary and they are free to decide if they want to fill the questionnaire. Participants will also be given a Student Project 3rd Party Evaluator Information Sheet and a participant consent form to make sure an ethical use of human participants.

The followings are design of the questionnaire.

1. What do you think about the overall game?

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Very bad | Bad | Neutral | Good | Very good |
|  |  |  |  |  |

1. What do you think about the user interface of the game?

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Very bad | Bad | Neutral | Good | Very good |
|  |  |  |  |  |

1. What do you think about the graphics effects of the game?

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Very bad | Bad | Neutral | Good | Very good |
|  |  |  |  |  |

1. What do you think about the control of Pac-Man in this game?

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Very bad | Bad | Neutral | Good | Very good |
|  |  |  |  |  |

1. What do you think about the mazes in the game?

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Very bad | Bad | Neutral | Good | Very good |
|  |  |  |  |  |

1. What do you think about the design of items in the game?

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Very bad | Bad | Neutral | Good | Very good |
|  |  |  |  |  |

1. What do you think about the design of ghosts in the game?

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Very bad | Bad | Neutral | Good | Very good |
|  |  |  |  |  |

1. What do you think about the difficulty of the game?

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Too easy | Easy | Normal | Difficult | Too difficult |
|  |  |  |  |  |

1. Do you have any suggestions to the user interface?

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. Do you have any suggestions to the control of Pac-Man?

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. Do you have any suggestions to the mazes?

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. Do you have any suggestions to the design of items?

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. Do you have any suggestions to the design of ghosts?

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. Do you have any suggestions to difficulty of the game?

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. Which aspect of the game is your favourite and/or least favourite?

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

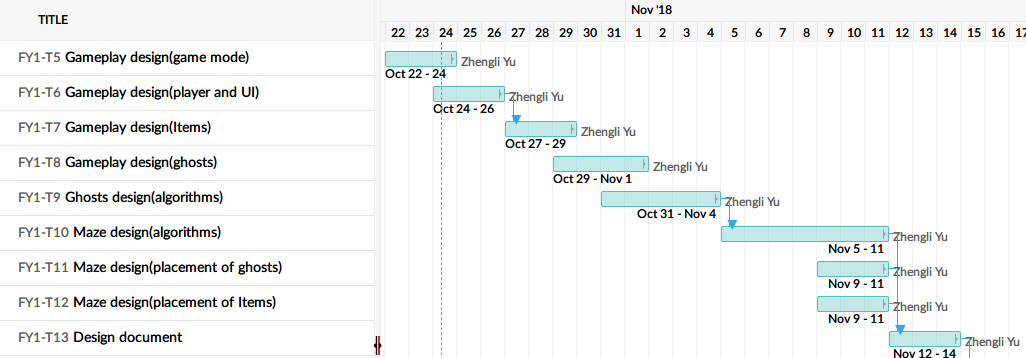
1. Do you have any extra comments and suggestions?

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. **Gantt Chart**

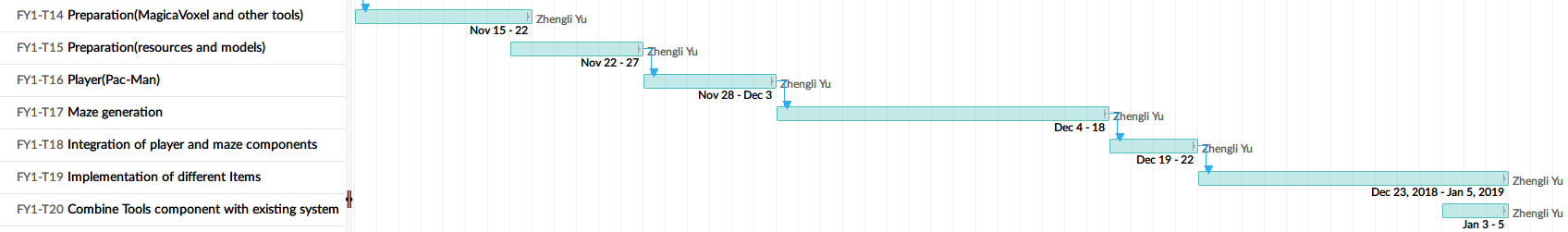
This is the Gantt Chart produced as part of the design specification, showing the schedule of the whole project. The chart is divided into 3 parts to make it easy to read.

* Part1\_Design stage



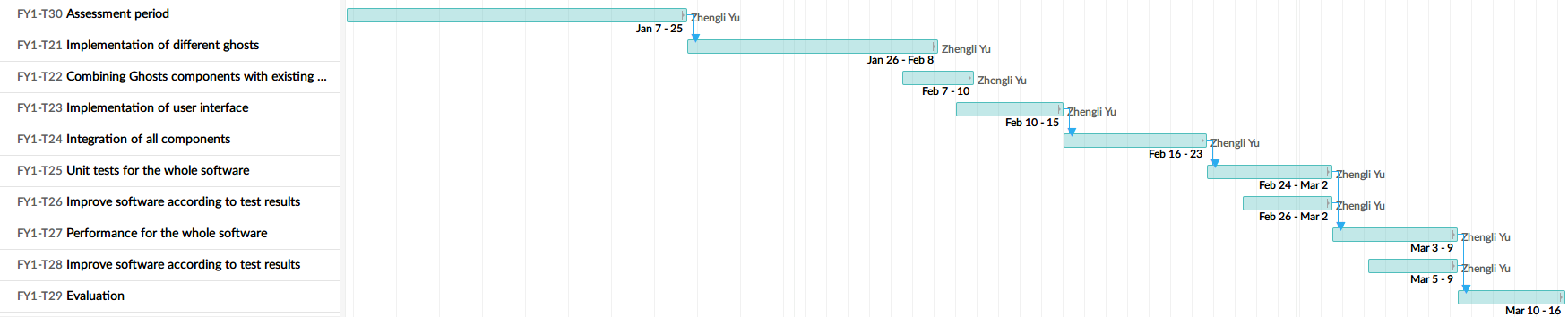
*Figure 5.1 GanttChart\_part1*

* Part2\_Implementation stage



*Figure 5.2 GanttChart\_part2*

* Part3\_Implementation, testing and evaluation stage



*Figure 5.3 GanttChart\_part3*

1. **Summary**

This document describes the design of Pac-Man project. An object-oriented design is applied to this project. In this document, all classes that will used in the software and their relations are discussed. Some pseudocode for some key methods are also given. Finally, design of evaluation stage is also covered in this article. All this design may be modified during the implement stage to improve the game and make a better product.

**Bibliography**

[1] MagicaVoxel. (2018, 1 Nov). *MagicaVoxel*. Available: <https://ephtracy.github.io/index.html?page=mv_main#>

[2] Wikipedia. (2018, 14 Oct). *Pac-Man*. Available: <https://en.wikipedia.org/wiki/Pac-Man>

[3] Wikipedia. (2018, 16 Oct). *Maze generation algorithm*. Available: <https://en.wikipedia.org/wiki/Maze_generation_algorithm>

[4] J. Buck. (2010, 27 Dec). *Maze Generation: Recursive Backtracking*. Available: <http://weblog.jamisbuck.org/2010/12/27/maze-generation-recursive-backtracking>

[5] J. Buck. (2011, 10 Jan). *Maze Generation: Prim's Algorithm*. Available: <http://weblog.jamisbuck.org/2011/1/10/maze-generation-prim-s-algorithm>

[6] Blender. (2018, 5 Nov). *Blender*. Available: <https://www.blender.org/>

[7] Wikipedia. (2018, 15 Oct). *Object-oriented design*. Available: <https://en.wikipedia.org/wiki/Object-oriented_design>

[8] R. C. Prim, "Shortest Connection Networks And Some Generalizations," *Bell System Technical Journal,* vol. 36, pp. 1389-1401, 1957.

[9] Unity. (2018, 2 Nov). *Vector3*. Available: <https://docs.unity3d.com/2018.1/Documentation/ScriptReference/Vector3.html>

[10] Unity. (2018, 2 Nov). *MonoBehaviour.FixedUpdate()*. Available: <https://docs.unity3d.com/2018.1/Documentation/ScriptReference/MonoBehaviour.FixedUpdate.html>

[11] Unity. (2018, 2 Nov). *Collider.OnCollisionEnter(Collision)*. Available: <https://docs.unity3d.com/ScriptReference/Collider.OnCollisionEnter.html>

[12] Unity. (2018, 4 Nov). *Colliders*. Available: <https://docs.unity3d.com/Manual/CollidersOverview.html>

[13] Unity. (2018, 6 Nov). *Physics.Raycast*. Available: <https://docs.unity3d.com/ScriptReference/Physics.Raycast.html>

[14] Unity. (2018 10, Nov). *UnityTestTools*. Available: <https://bitbucket.org/Unity-Technologies/unitytesttools/overview>