

*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 aim of this “Pac-Man” project is to develop a Pac-Man game or a variant thereof. In the game, player can control PacMan to go through mazes and avoid ghosts. The game is developed by Unity Engine and ghosts are controlled by artificial intelligence. The project is supervised by Alexei Lisitsa.

This game is 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 consists of three main components: gameplay, mazes generation and AI for ghosts. The maze generation system is thought to be the most important component in the game because it is the basic system for gameplay.

In the final product, this game is developed using Unity game engine. Some other software such as GIMP, Blender and MagicaVoxel are also used to create resources for this game. All requirements mentioned in the design document have been met successfully. Three different maze generation algorithms, seven different items and four different ghosts are implemented in the game. Some extra features including animation system, particle system and sound effects are also added to the game. Details about the final product can be found in Realisation section. In the evaluation stage, the final product passes the test scenes and performance test. This game also receives a positive feedback from players’ evaluation. Details about the evaluation results can be found in “Evaluation” section.

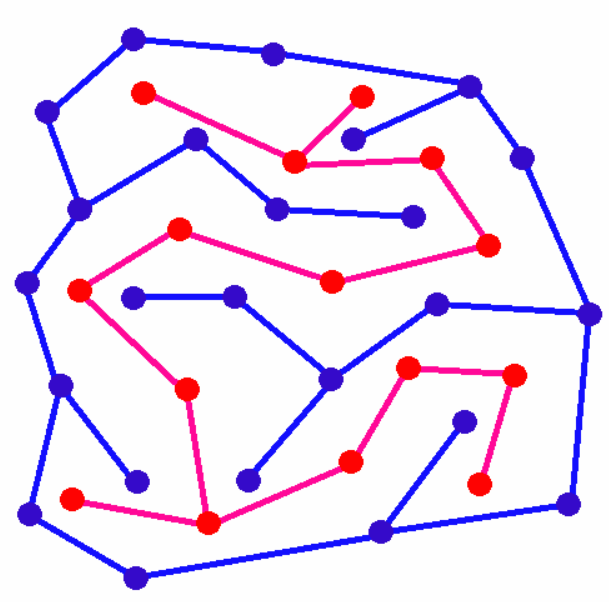
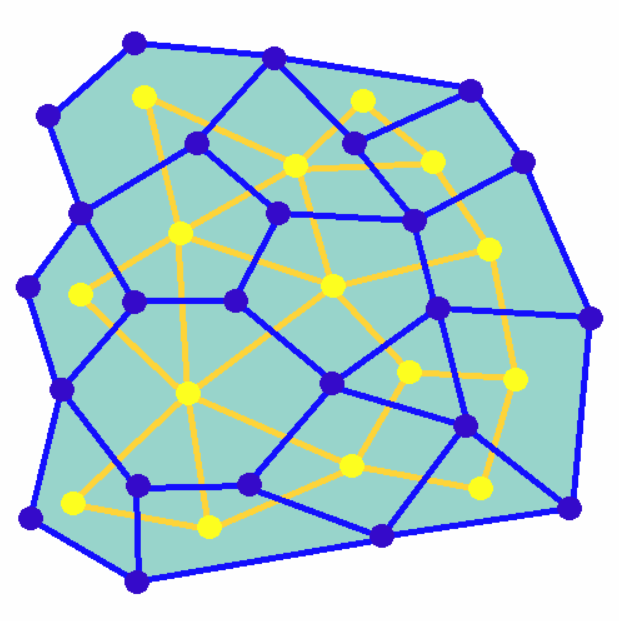
1. **Background**

**3.1 Original Pac-Man game:** The original Pac-Man is an arcade game developed by Namco and first released in 1980 [2]. In that game, players can navigate a PacMan to go through a maze which contains Pac-Dots and ghosts. The goal of the game is to avoid ghosts and collect Pac-Points by eating the Pac-Dots. Pacman can clear current stage and start next game after reaching the exit of maze with enough points.



*Figure 3.1 Original Pac-Man Game*

**3.2 Mazes generation algorithms:** Another important research is about the maze generation algorithms. In this game, a maze will be generated automatically for 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. In this graph, nodes represent cells and edges represent possible walls in the maze. The purpose of the maze generation algorithm is to traverse all cells and make a path from start cell to end cell (show in figure 3.2) [3].



*Figure 3.2 Generation of a maze*

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

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

3D modelling software such as 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]. They are used to implement the models and animations of the Pac-Man project.

1. **Data Required**

No human data and human participants are needed during in the implement of this project.

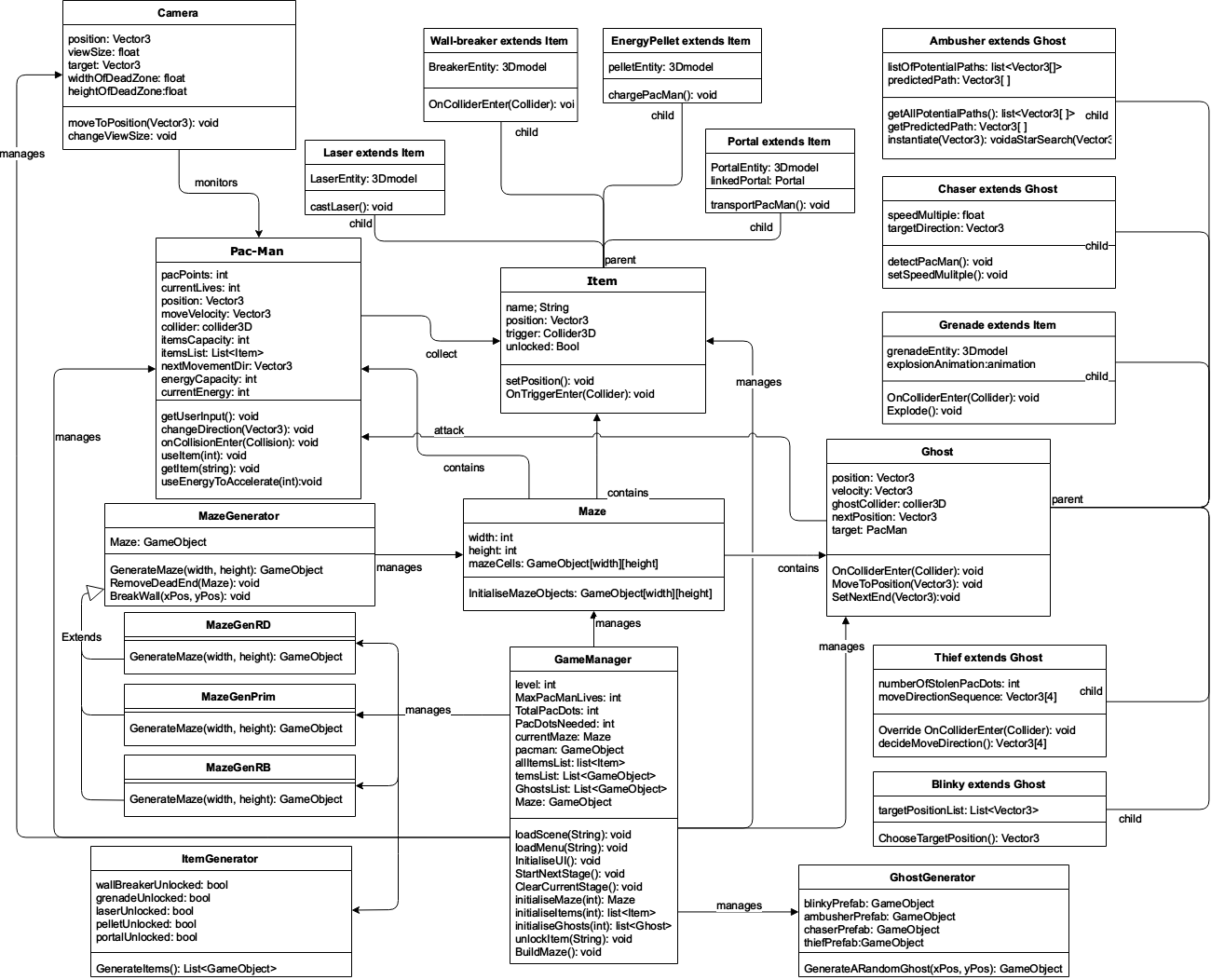
All background music used in the final product come from Jamendo Music [7]. This website provides royalty free music so they can be used freely in this non-commercial project. In addition, titles of music and the name of artist will be displayed on the bottom-left of the screen.

In terms of other resources used in the project, the images of PacMan and ghosts take original PacMan game as a reference. In addition, most of resources used in the project are created independently and some resources such as sound effects come from free and public assert package. The following table shows all resources which are provided by assert package.

|  |  |
| --- | --- |
| ***Resource*** | ***Source of the resource*** |
| Sound effects of all items | Unity official standard assert package |
| Part of textures and materials of particle system | Unity official standard assert package |
| Font of user interface text | Unity official standard assert package |
| Part of materials used on Portal item | Free “Gem” unity assert store package |
| Images of “Item Capacity” and “Energy Capacity” in shop interface | Unity official standard assert package |

1. **Project Design**

The object-oriented design is applied in this project. Object-oriented design is the discipline of defining the objects and interactions between them to solve a certain problem [8]. 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.

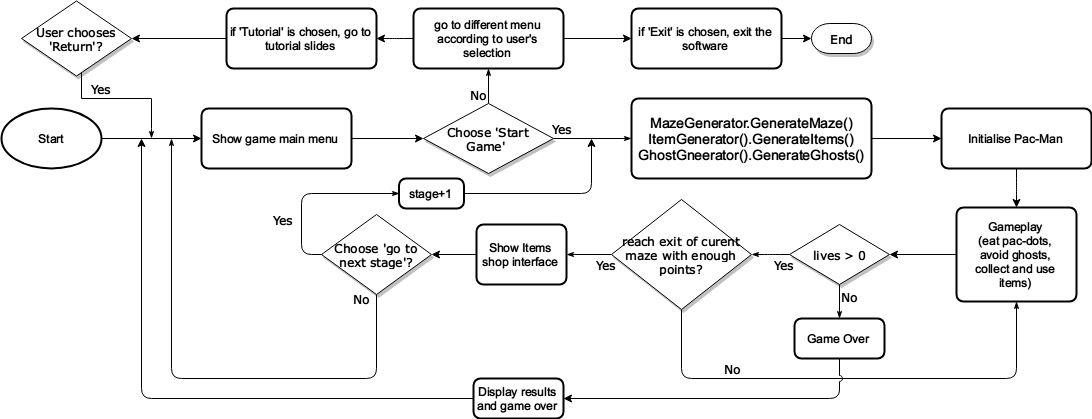


*Figure 5.1 Class Diagram*

* 1. **Design of GameSceneManager**

|  |
| --- |
| **GameSceneManager** |
| level: int  MaxPacManLives: int  TotalPacDots: int  PacDotsNeeded: int  currentMaze: Maze  pacman: GameObject  allItemsList: list<Item> |
| loadScene(String): void  loadMenu(String): void  InitialiseUI(): void  StartNextStage(): void  ClearCurrentStage(): void  initialiseMaze(int): Maze  initialiseItems(int): list<Item>  initialiseGhosts(int): list<Ghost>  unlockItem(String): void |

The GameManager is used to manage 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.

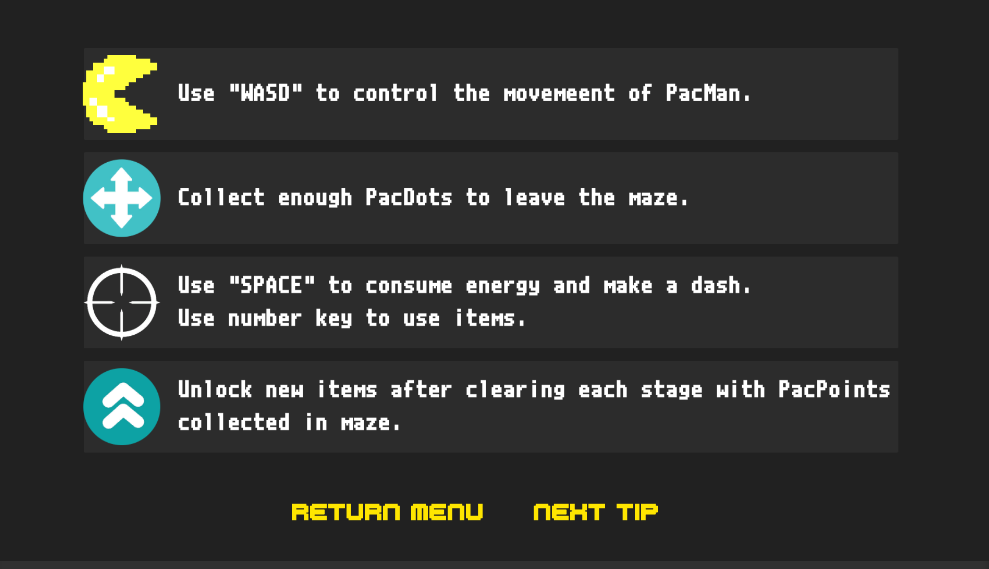


*Figure 5.1.1 Basic Game Loop*

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 5.1.2 are examples of layouts of the several different user interfaces.





*Figure 5.1.2 Examples of Layout of UI*

* By clicking the ‘Start Game’ button, the game scene will be loaded and gameplay loop is started.
* In the ‘Tutorial’ interface, user can information about how to play this game. Guidelines about control, items and ghosts will be provided.
* The software will be closed if ‘Quit’ is clicked
* During game, user can exit the game by clicking a ‘Exit’ button. Player can also turn off the music by clicking the ‘Mute’ button.
* In each interface, user can go back to the initial interface by clicking the ‘Return’ button.
* **Gameplay loop**
* The gameplay loop starts when user selects ‘Start 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 up 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 game, pacman will move with a certain speed and player can use ‘WASD’ to 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 clears current stage, a summary and shop page will be displayed. In this page, player will be rewarded some PacPoints according to the number of PacDots collected in this stage. PacPoints can be used to unlock new items, increase max capacity of items and boost energy. Then, player should choose to continue or exit. If player chooses to exit, game scene will return main menu interface. If player chooses to continue, number of stage will be increased and player enters next game loop.
  1. **Design of Maze**

In each stage of game, a Maze will be used to store information of the maze in the game environment. Some extra attributes and functions in the GameSceneManager are also used to manage the maze. 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 game objects information of the maze. Information of all items and ghosts will also be stored in the GameSceneManager.

|  |  |  |
| --- | --- | --- |
| **Maze** |  | **GameSceneManager(part for maze)** |
| width: int  height: int  mazeObjects: GameObject[width][height] | ItemsList: List<GameObject>  GhostsList: List<GameObject>  Maze: GameObject |
| InitialiseMazeObjects: GameObject[width][height] | BuildMaze(): void |

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

1. There should be more than one path between any two positions in the maze
2. Each position which is not wall in the maze should be reachable
3. Pac-Man should not move outside the maze
4. The width of the path should be 1 unit (the 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 generation of a complete maze is divided into two stages: generate a maze and place items and ghosts into the maze. Several different classes and objects are cooperated to build the maze. This section will discuss MazeGenerator first.

|  |
| --- |
| **MazeGenerator** |
| Maze: GameObject |
| GenerateMaze(width, height): GameObject  RemoveDeadEnd(Maze): void  BreakWall(xPos, yPos): void |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **MazeGenRD: MazeGenerator** |  | **MazeGenPrim: MazeGenerator** |  | **MazeGenRB: MazeGenerator** |
|  |  |  |
| GenerateMaze(width,height): GameObject | GenerateMaze(width,height): GameObject | GenerateMaze(width,height): GameObject |

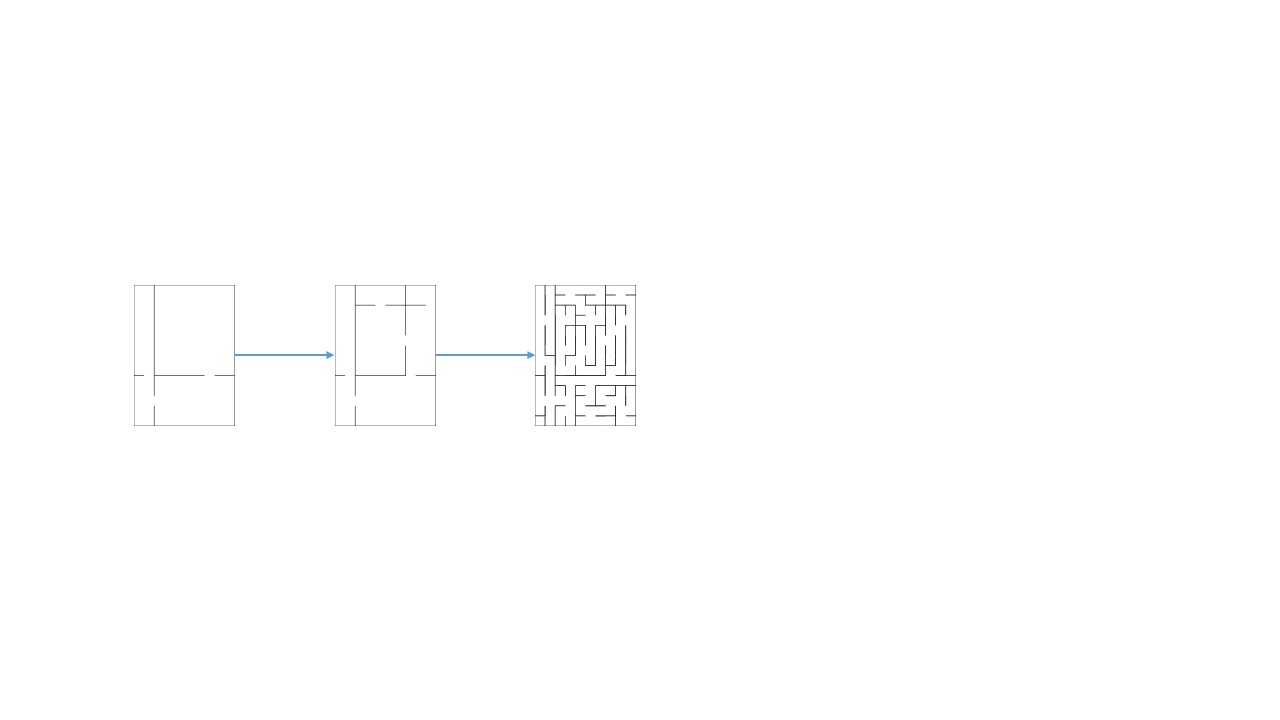
The MazeGenerator is the super class for MazeGenRD, MazeGenPrim and MazeGenRB. MazeGenRD can generate a maze using the recursive division. MazeGenPrim will generate a maze using randomized prim algorithm and MazeGenRB creates a maze using recursive backtracker algorithm. This structure makes it easy and clear for GameSceneManager to generate different kinds of maze.

Three different maze generation algorithms 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, Randomized Prim algorithm and Recursive Backtracker. 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 9\*9 and every 3 stages, the width and height of the maze will be increased. 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.

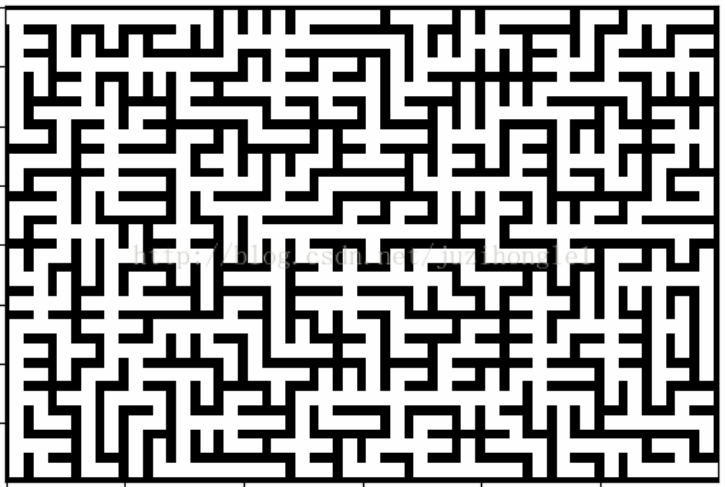
1. Begin with an empty field. Create boundary walls for the maze
2. Function recursiveDivision(field) {
3. If this field cannot be divided, return
4. Choose a random position, create a list of walls, either horizontally or vertically
5. Choose a random position on that list of walls and open a “door”
6. The original field is divided into two new fields
7. For each new field, call recursiveDivision(field)
8. Continue recursively until the maze is completed
9. End function



*Figure 5.2.1 sample of Recursive Division algorithm*

Figure 5.2.1 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.

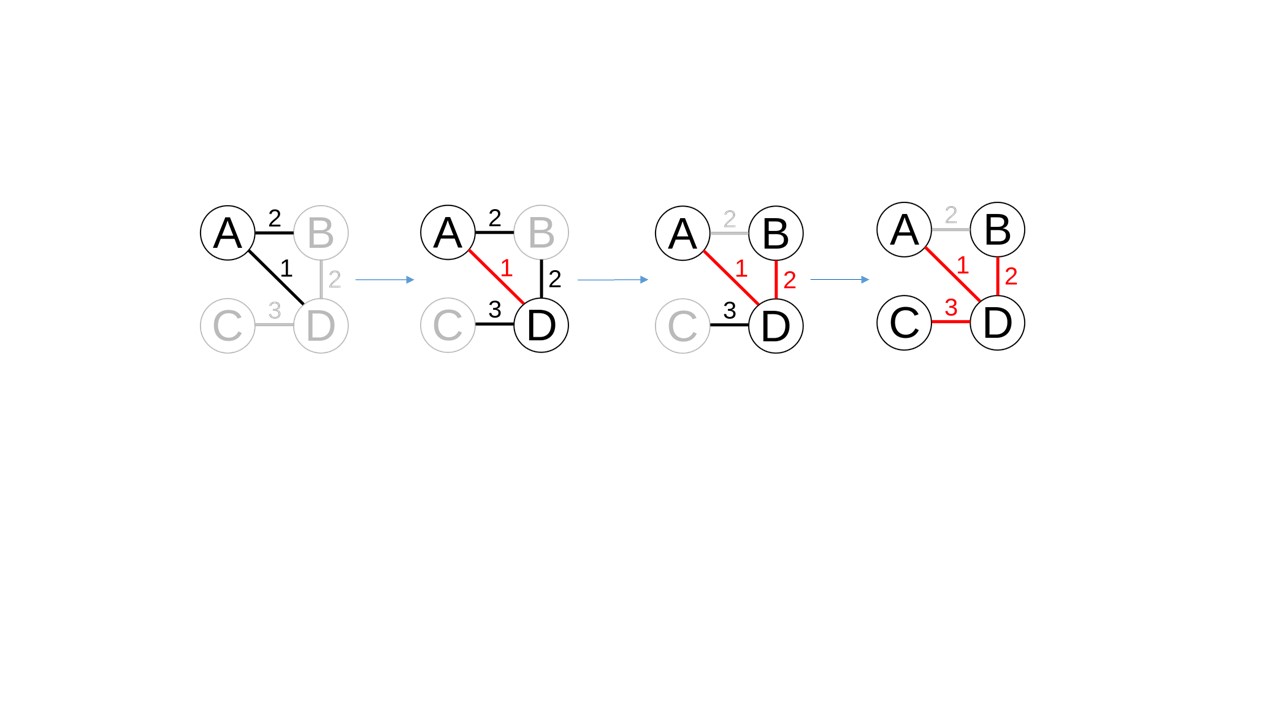
* **Randomized Prim’s algorithm[5]**

****

*Figure 5.2.2 Sample of maze generated by Randomized Prim’s algorithm*

This algorithm is based on a tree generation 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 [9].

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 5.2.3 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. 1.Begin with a m\*n maze whose maze cells are all walls
2. 2.Select a start point, break the wall on this position. Add its neighbouring walls to the wall list.
3. 3.While there are walls in the wall list:
4. 1.Pick a random wall from the list. If only one of the two cells that the wall divides is visited, then:
5. 2.Break the wall and mark the unvisited cell as part of the maze.
6. 3.Add the neighbouring walls of the cell to the wall list.
7. 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 contains many long, straight roads. Therefore, compared with Recursive Division, randomized prim provides a more complex and difficult maze.

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



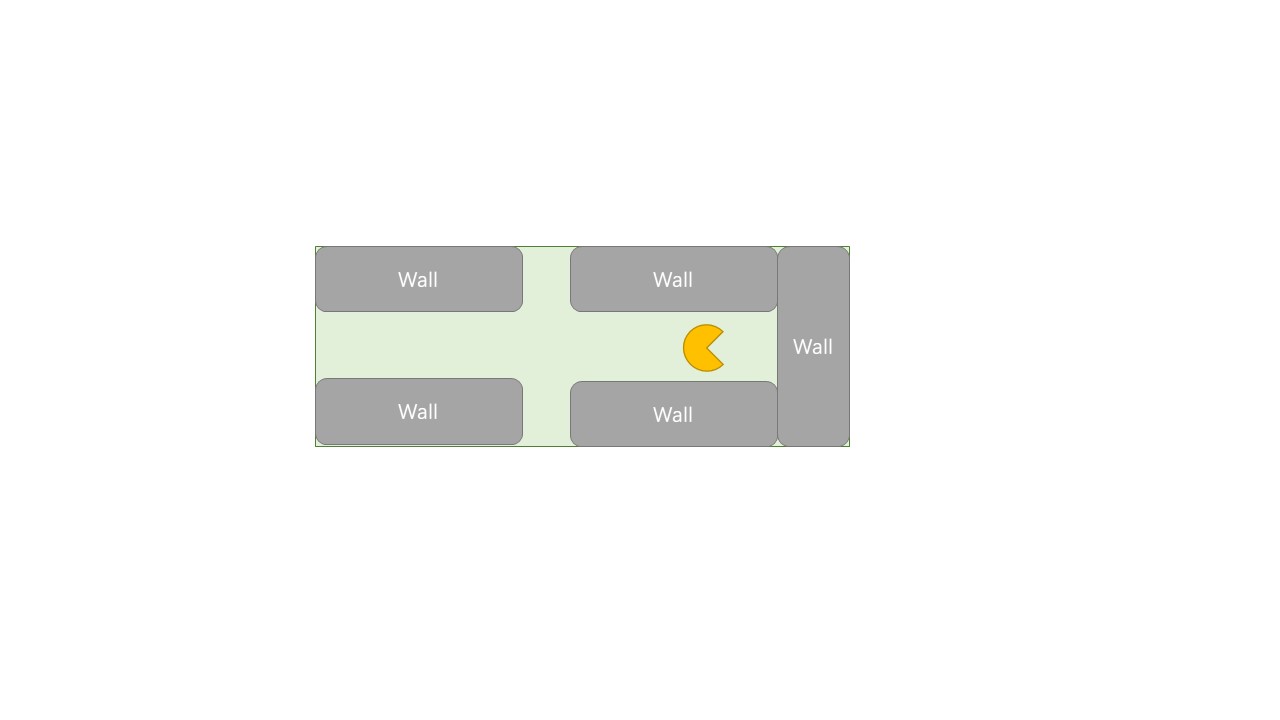
*Figure 5.2.4 Sample of maze generated by Recursive Backtracker*

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. 1.  Begin with a m\*n maze whose maze cells are all walls.
2. 2.  Make the start point the current cell and mark it as visited
3. 3.  While there are unvisited cells
4. 1.If the current cell has any unvisited neighbours
5. 1.Choose one of the unvisited neighbours randomly
6. 2.Push the current cell to the stack
7. 3.Remove the wall between the current cell and the chosen cell
8. 4.Make the chosen cell the current cell and mark it as visited
9. 2.Else if stack is not empty
10. 1.Pop a cell from the stack
11. 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 continuous corners, which makes this kind of mazes become the most difficult kind of maze for the PacMan game.

* **RemoveDeadEnd(Maze): void**



*Figure 5.2.5 Dead End*

After a maze is generated, there will be some “dead ends” in the maze (figure 5.2.5). 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, such kind of dead ends in a maze will be removed by breaking one of its three walls. To avoid making the maze too easy, not all dead ends will be removed. For every two dead ends, only one dead end will be chosen to be removed. RemoveDeadEnds(Maze) will be called to complete this job.

1. Function removeDeadEnds(Maze) {
2. Bool breakNext = true;
3. For every empty cell in the Maze {
4. If(breakNext = false), continue; breakNext = true;
5. If (up, right and down of the cell is blocked by walls), break it right wall; breakNext = false;
6. If (left, up and right of the cell is blocked by walls), break it up wall; breakNext = false;
7. If (up, left and down of the cell is blocked by walls), break it left wall; breakNext = false;
8. If (left, down and right of the cell is blocked by walls), break it down wall; breakNext = false;
9. 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.

|  |  |  |
| --- | --- | --- |
| **ItemGenerator** |  | **GhostGenerator** |
| wallBreakerUnlocked: bool  grenadeUnlocked: bool  laserUnlocked: bool  pelletUnlocked: bool  portalUnlocked: bool | blinkyPrefab: GameObject  ambusherPrefab: GameObject  chaserPrefab: GameObject  thiefPrefab: GameObject |
| GenerateItems(): List<GameObject> | GenerateARandomGhost(xPos, yPos): GameObject |

1. Function placeItemsAndGhostInMaze() {
2. For (int i = 0; i**<** **Maze.width**; i = i + 5) {
3. For (int j = 0; j **<** **Maze.height**; j = j + 5) {
4. RandomX = random(i, i + 5)
5. RandomY = random(j, j + 5)
6. If (Maze[randomX][randomY] != wall) {
7. Place a random item which has been unlocked in Maze[randomX][randomY]
8. }
9. RandomX = random(i, i + 5)
10. RandomY = random(j, j + 5)
11. If (Maze[randomX][randomY] != wall && there is no item in Maze[randomX][randomY]) {
12. Place a random ghost in Maze[randomX][randomY]
13. }
14. }   }   }
    1. **Design of PacMan(Player)**

|  |
| --- |
| **PacMan** |
| pacPoints: int  currentLives: int  position: Vector3  moveVelocity: Vector3  collider: collider3D  itemsCapacity: int  itemsList: List<Item>  nextMovementDir: Vector3  energyCapacity: int  currentEnergy: int |
| getUserInput(): void  changeDirection(Vector3): void  onCollisionEnter(Collision): void  useItem(int): void  getItem(string): void  useEnergyToAccelerate(int): void |

In this game, player can control and navigate the Pac-Man to go through the maze. The control of the Pac-man consists of three parts: 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 [10].
* 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. Player can change its move direction by consuming 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 [11]. Therefore, the Pac-Man can keep getting input from users and change its direction when there is no wall at that direction. The followings are pseudo code for movement of Pac-Man.

1. Function FixedUpdate() {
2. Move with current velocity
3. getUserInput()
4. changeDirection(Vector3)
5. }
7. Function getUserInput() {
8. If ‘W’ is entered then nextMovementDir = new Vector3(0,1,0)
9. If ‘S’ is entered then nextMovementDir = new Vector3(0, -1,0)
10. If ‘A’ is entered then nextMovementDir = new Vector3(-1,0,0)
11. If ‘D’ is entered then nextMovementDir = new Vector3(1,0,0)
12. }
14. Function changeDirection(Vector3) {
15. If (nextMovementDir==(0,1,0) && noWallOnTopOfPacMan) then change direction to up
16. If(nextMovementDir==(0,-1,0) && noWallOnBottomOfPacMan) then change direction to bottom
17. If (nextMovementDir==(-1,0,0) && noWallOnLeftOfPacMan) then change direction to left
18. If (nextMovementDir==(1,0,0) && noWallOnRightOfPacMan) then change direction to right
19. }

* **Interactions with game objects**
* The interaction among Pac-Man and game objects are mainly implemented using collider. Collider3D is a Unity component which is used to detect collisions between game objects. Collider components 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 will be able to use it later. If Pac-Man touches a PacDot, PacMan will be rewarded one point and boost 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 [12]. The followings are pseudo code for interactions of Pac-Man.

1. Function OnCollisionEnter(Collision) {
2. If (Collision.collider.type==wall) then do nothing
3. If (Collision.collider.type==ghost) {
4. lives = lives -1
5. if(lives **>** 0) then respawn the Pac-Man
6. }
7. If (Collision.collider.type==item && itemsList is not full) {
8. add item to list
9. destroy the item
10. }
11. If (Collision.collider.type==PacDot) {
12. If (energy **<** **energyLimit**), then energy = energy + 1
13. points = points + 1
14. destroy the PacDot
15. }
16. }

* **Use items and energy**
* Pac-Man can pick up items in the maze and store it in an item list. At first, a Pac-Man can have at most 3 item at the same time. Player can consume pacpoints to upgrade its maximum capacity of items after each stage of game. If there is any available item in the list, player can use these items by pressing according number keys. Function of each item will be discussed in ‘Design of Items’ section.

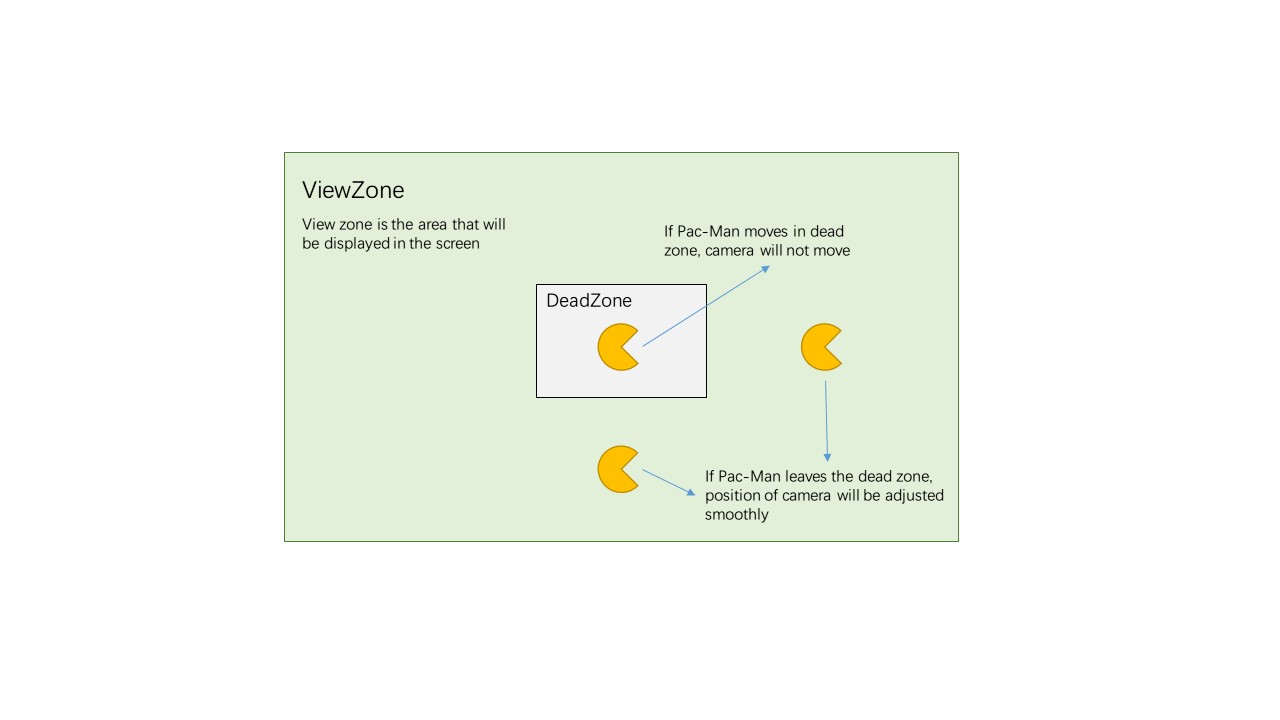
1. //Example pseudo code for using of Items
2. Function Update(){
3. If (a number key is pressed && there is an item in the list) {
4. Use that item
5. }
6. }

* Pac-Man can charge its boost 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.

1. Function Update() {
2. If (‘Space’ button is presses and there is enough energy){
3. MoveSpeed = moveSpeed + speedIncrement
4. }
5. }
   1. **Design of Camera**

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

The view of camera is divided into two areas: view zone and dead zone. The structure of camera view is shown in the figure 5.4.1. This kind of design is to provide a smooth movement of camera. Compared with the camera that follows the target directly, this kind of camera reduces the shaking of camera and players can have better gameplay experience.



*Figure 5.4.1 Camera View*

The followings are pseudo code for camera.

1. Function Update() {
2. If target is inside the dead zone {
3. do nothing
4. } else if target leaves the dead zone {
5. move towards the direction of target
6. }
7. }
   1. **Design of Items**

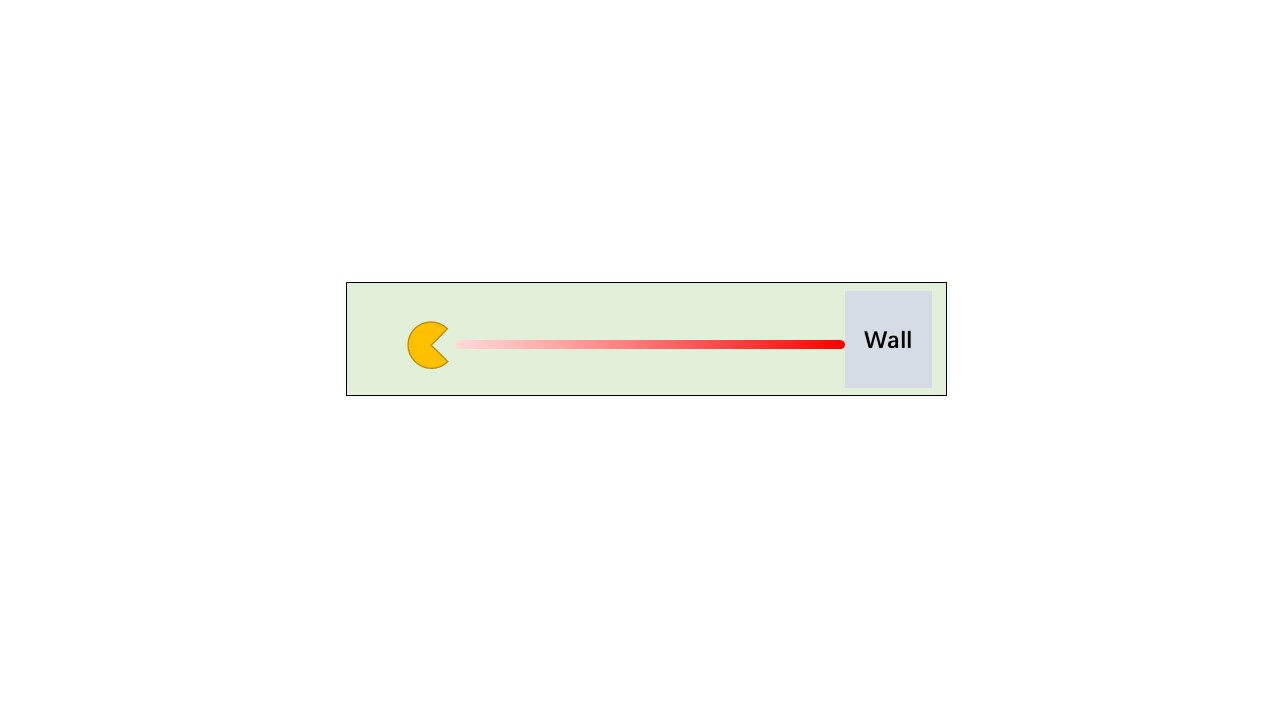
Some items will be placed in the maze. Pac-Man can pick them up and use them later in the game. Each item can provide a unique ability or bonus for Pac-Man, which is helpful when solving the maze. In the design stage, five different items have been designed and all of them extends from a super class: 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. It does not behave as s solid object and a trigger will allow other colliders to pass through. A trigger will call the OnTriggerEnter(Collider) function when a collider enters its collider [13]. Therefore, a trigger is a good way for the detection of interactions between Pac-Man and items.

1. Function OnTriggerEnter(Collider) {
2. If(Collider is Pac-Man && Pac-Man’s items list is not full) {
3. Add item to Pac-Man’s items list
4. Remove item from the maze
5. }
6. }

* All items are locked if it is the first stage of this game. Players should earn pacpoints by playing the game and consume PacPoints to unlock new items.
* **Laser**



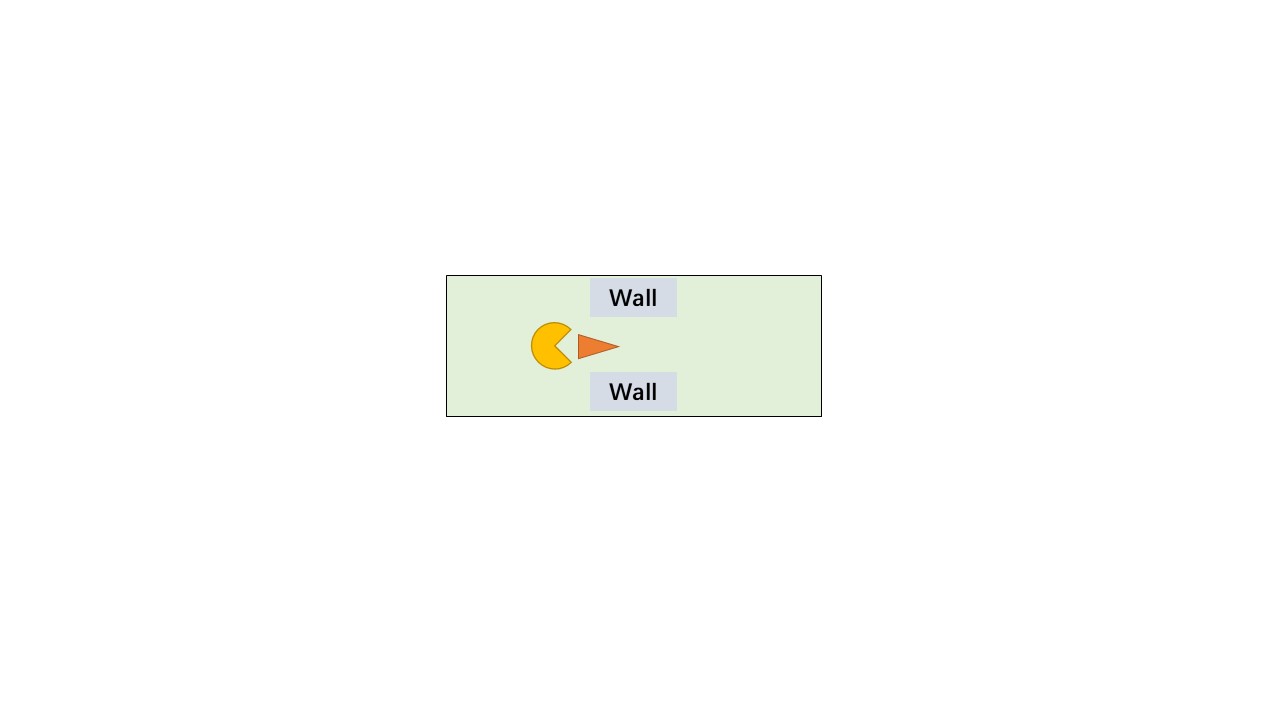
When player uses a laser item, a laser will be cast from the head of Pac-Man, in direction of Pac-Man’s movement. When the laser hits a ghost, it will disable that ghost for 15 seconds. The laser will last 5 seconds and it cannot get through the wall. A laser item is implemented by a Laser class.

|  |
| --- |
| **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.

1. Function castLaser() {
2. RayCast(originPoint, Direction)
3. If ray hits a wall, then the laser will stop in front of the wall
4. If ray hits a ghost, then the ghost will be disabled for 15 seconds
5. After 5 seconds, the laser object will be destroyed.
6. }
7. Function castLaser() {
8. RayCast(originPoint, Direction)
9. If ray hits a wall, then the laser will stop in front of the wall
10. If ray hits a ghost, then the ghost will be disabled for 15 seconds
11. After 5 seconds, the laser object will be destroyed.
12. }

* **Wall-breaker**

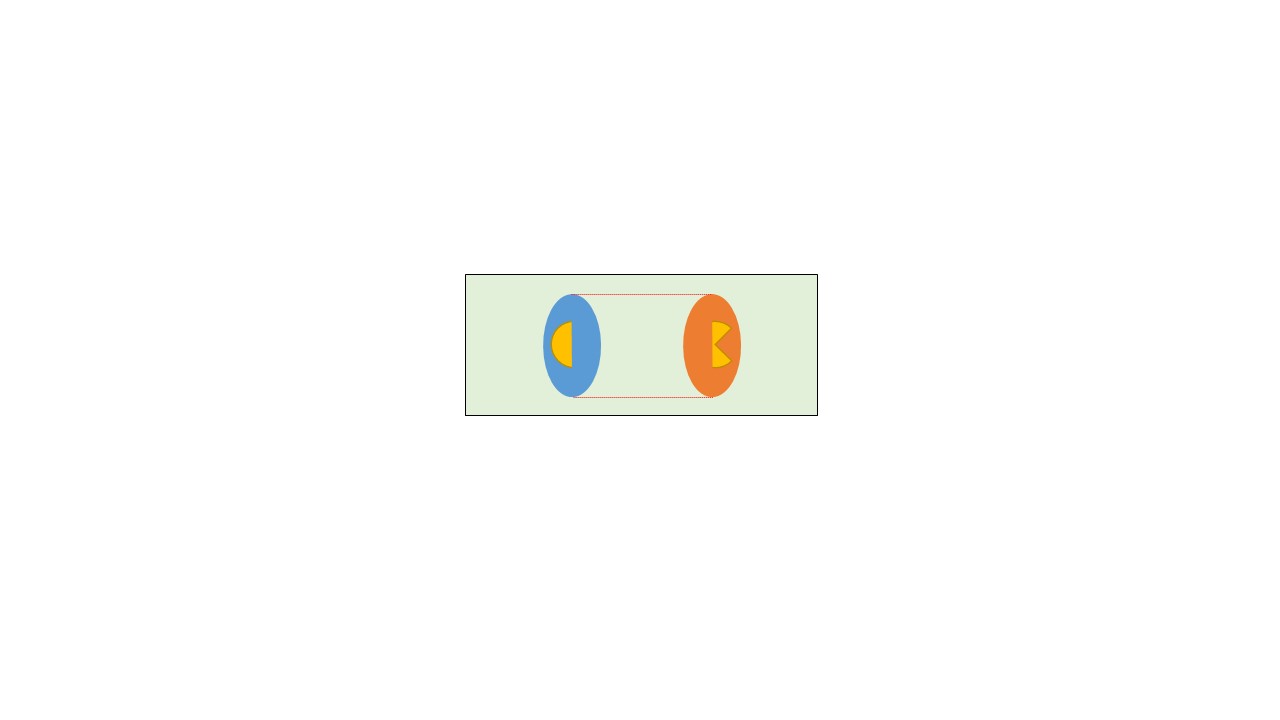


* Using this item will generate a wall-breaker object in front of the Pac-Man. When the breaker colliders with a wall, both of wall and breaker will be destroyed. If the breaker colliders with a ghost, the breaker will be destroyed and the ghost will be disabled permanently.

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

1. Function OnColliderEnter(Collider) {
2. If Collider.type == wall, then destroy the wall and this breaker
3. If Collider.type == ghost, then destroy this breaker and disable the ghost permanently
4. }

* **Portal**

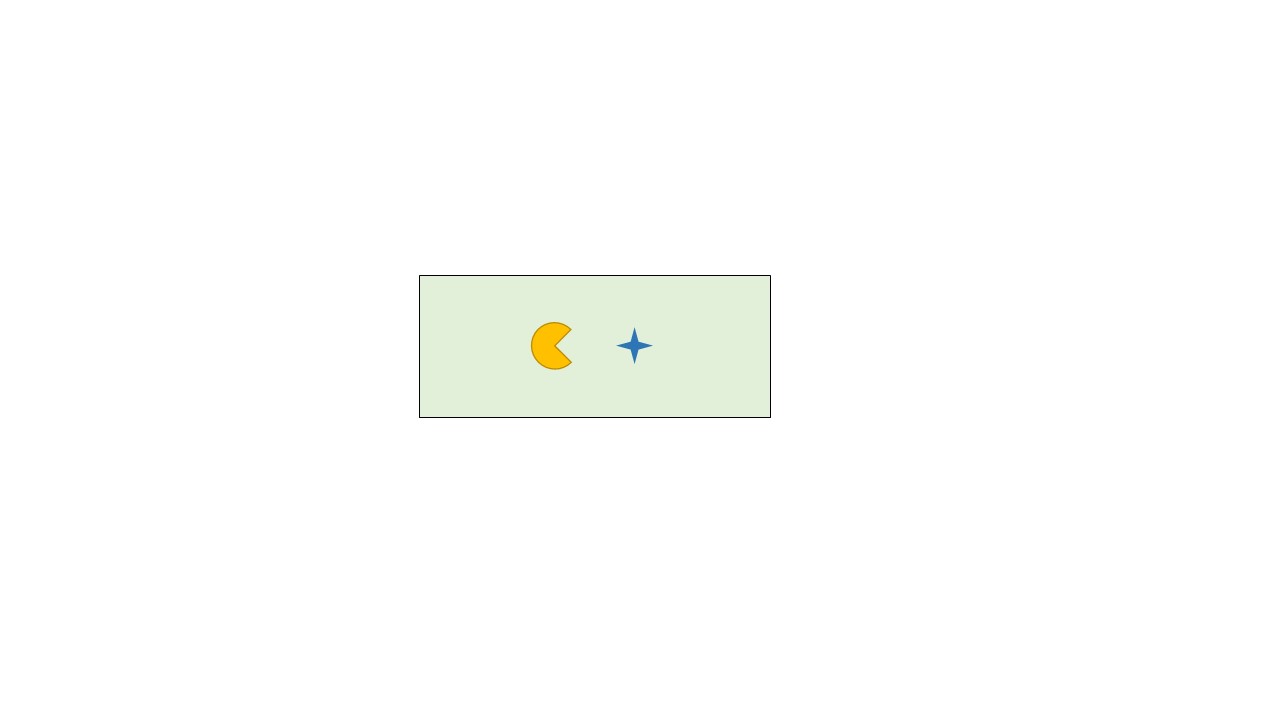


Portals should appear in pairs in the maze so each portal should link with another portal. When the player first time uses the portal item, the first portal will be created at the position of the Pac-Man. Then, player can use the item again to create another portal and two portals will be linked. When a Pac-Man pass through one of the two portals, it will be transported to the position of the linked portal. Portals in the maze also have a cooldown time so PacMan cannot use a portal continuously. The cooldown time will be displayed on the top of the portal.

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

1. If player uses the item for the first time, then create a portal at the position of the Pac-Man
2. If (player uses the item for the second time && current position != position of first portal) {
3. Create another portal and link two portals
4. }
6. Function Update() {
7. If (Pac-Man arrives at a portal and the portal has a linked portal) {
8. Transport the Pac-Man to the position of linked portal
9. }
10. }

* **Energy pellet**

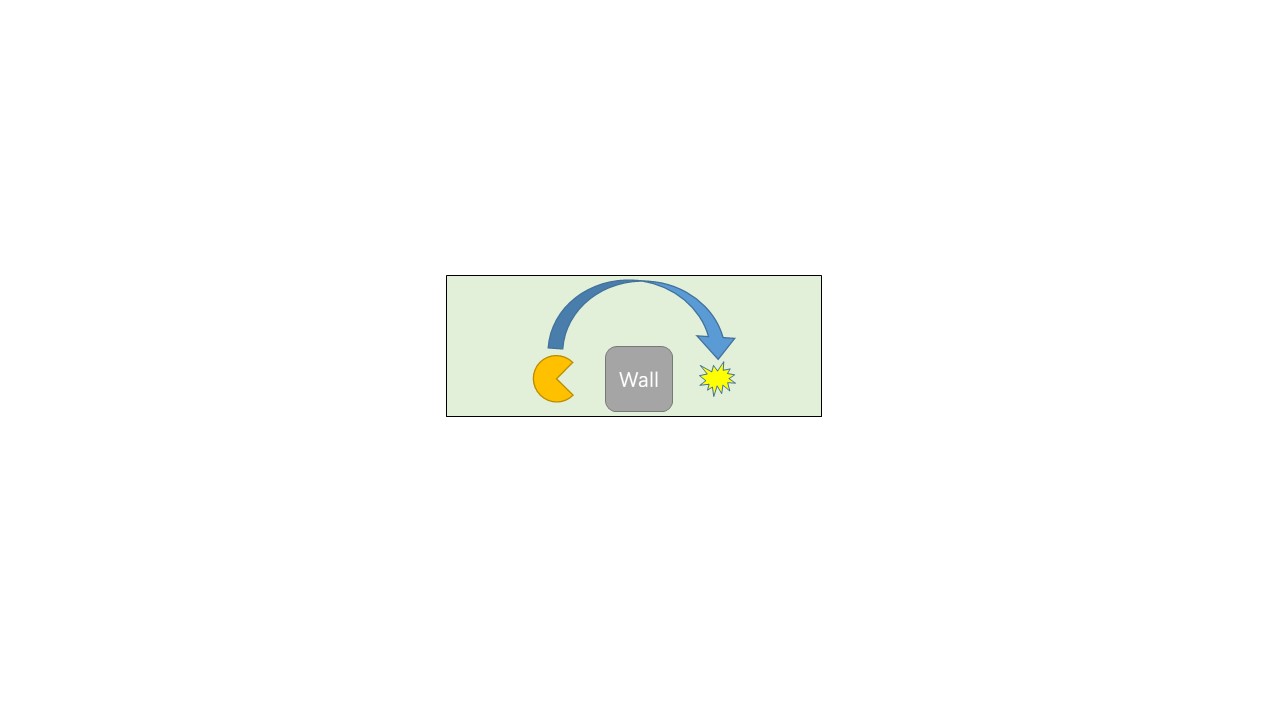


Pac-Man can get 100 points of boost energy by eating the energy pellet. This item will ignore the energy capacity of the PacMan.

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

1. Function chargePacMan() {
2. If player uses energy pellet, then Pac-Man.energy = Pac-Man.energy + 100
3. }

* **Grenade**



Player can throw a grenade over a wall and attack a ghost. The grenade will explode when it collides with ghost and destroy that ghost. If it does not touch any ghost, it will keep moving forward. If it hit a wall, it will stay near the wall and PacMan can pick it up again.

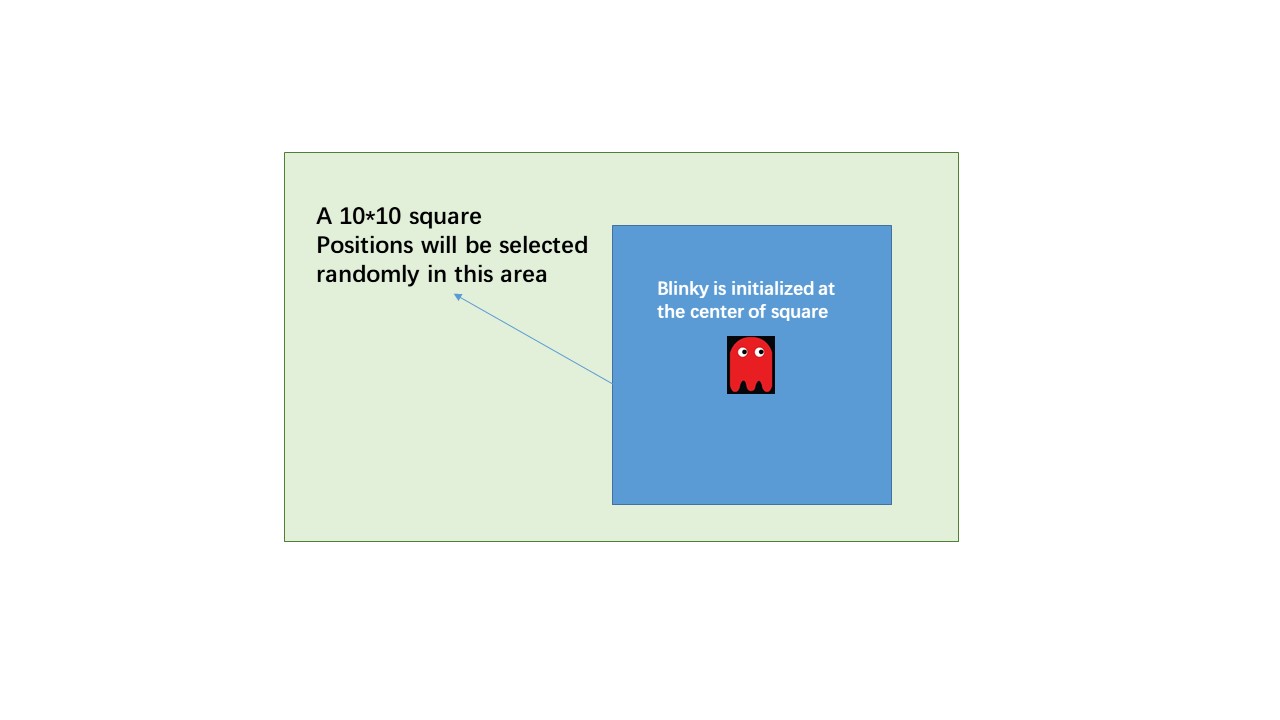
|  |
| --- |
| **Grenade extends Item** |
| grenadeEntity: 3Dmodel  explosionAnimation: animation |
| OnColliderEnter(Collider): void  Explode(): void |

1. //Example of pseudo code for grenade item
2. Grenade.position = PacMan.position + PacMan.direction\*2
4. Update() {
5. If(grenade collides with any wall), then stay near the wall
6. If(grenade collides with any ghost), then explore()
7. If(grenade collides with PacMan), then PacMan collect the grenade item
8. }
10. Function explode() {
11. PlayExplosionAnimation
12. Destroy the ghost
13. Self.destory()
14. }
    1. **Design of Ghosts**

This section will discuss several kinds of ghost that moving in the maze and interact with the Pac-Man. In the design stage, 4 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  SetNextEnd(Vector3): void |

* **Blinky**



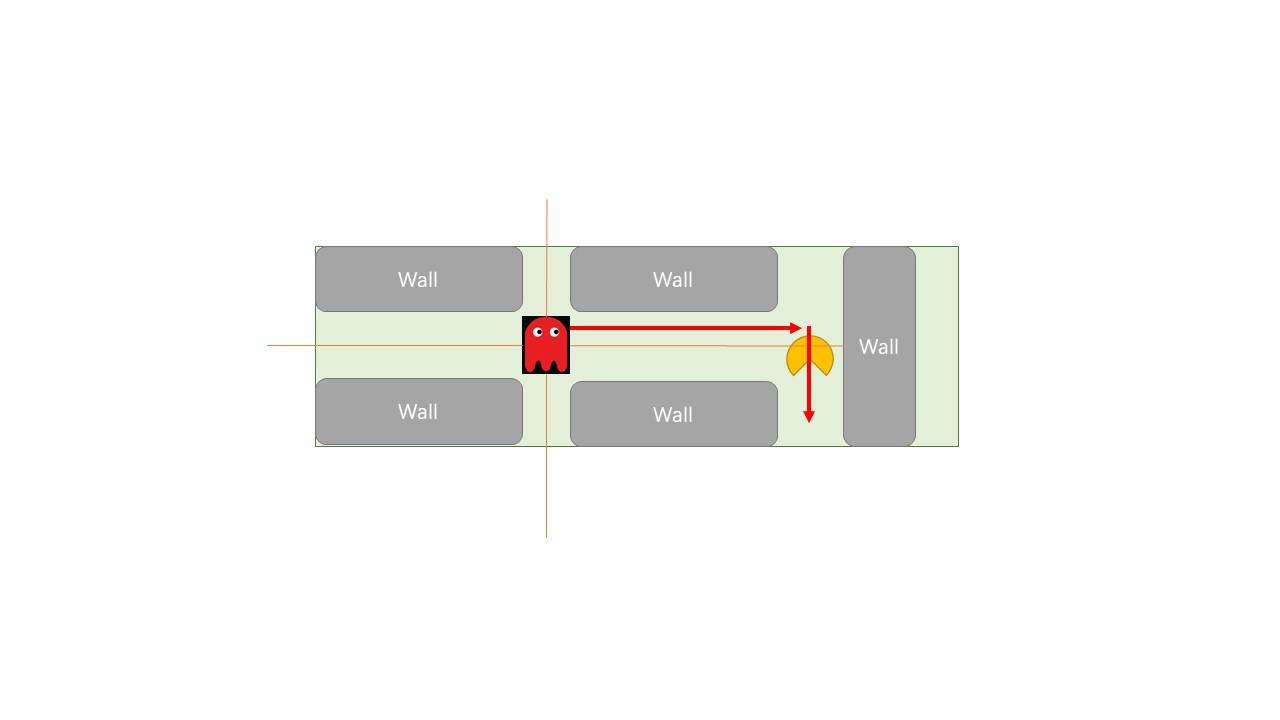
*Figure 5.6.1 Blinky*

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 |

1. //Example of pseudo code for Blinky
2. Choose 10 number from (position.x - 5, position.x + 5) as X coordinate of target positions
3. Choose 10 number from (position.y - 5, position.y + 5) as Y coordinate of target positions
4. Add those positions to target position list
5. While(true) {
6. Choose a position from the list, move to the position
7. After reaching the target position, choose another one from the list
8. }

* **Chaser**



*Figure 5.6.2 Chaser*

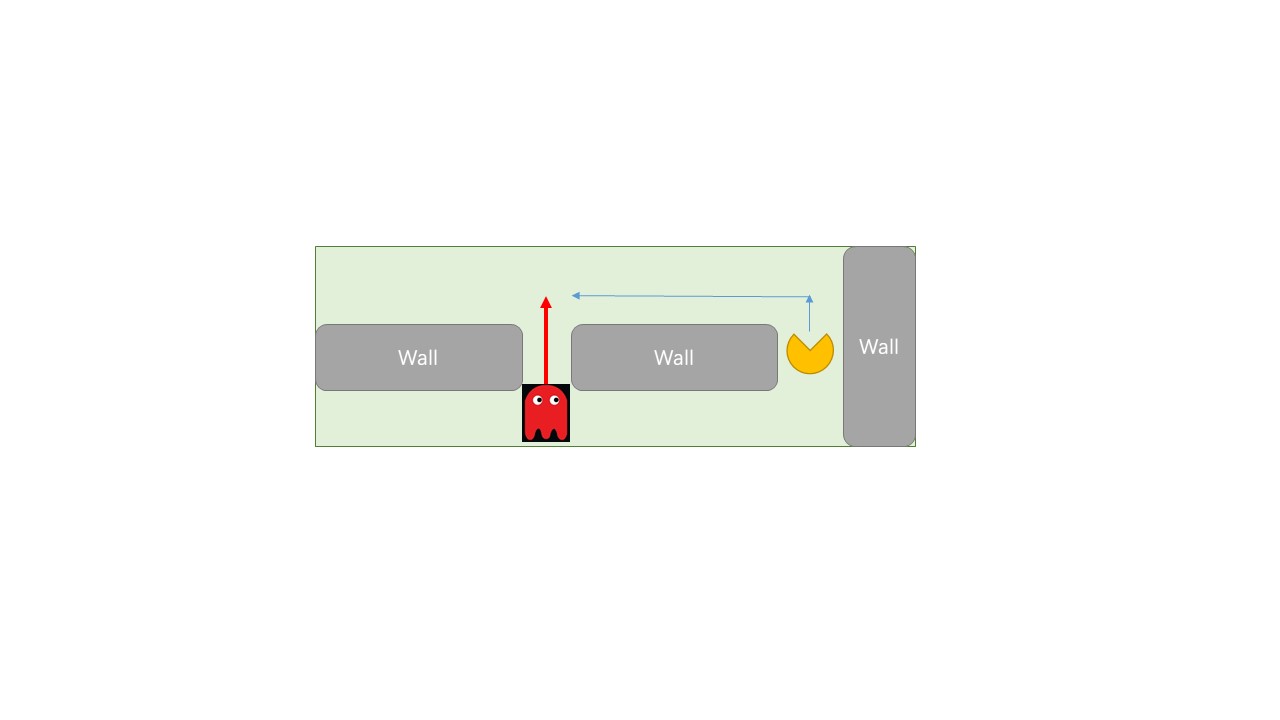
This kind of ghost will detect the appearance of PacMan in four directions. It a PacMan is found, Chaser will record its position and direction of movement. Then Chaser will move to that position and move along the direction of PacMan. A chaser can move faster than the PacMan. The following is structure of Chaser class.

|  |
| --- |
| **Chaser extends Ghost** |
| speedMultiple: float  targetDirection: Vector3 |
| detectPacMan(): 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 [14]. *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.

1. //Example of pseudo code for Chaser
2. Update() {
3. Cast rays to up, down, right, left four directions
4. If(ray hit a PacMan) {
5. TargetPosition = PacMan.position, targetDirection = PacMan.direction
6. MoveToPosition(targetPosition)
7. After arriving the position, turn to the target direction and move ahead
8. }
9. If(Chaser hit a wall && targetDirection is null), then stop and scan PacMan again
10. }

* **Ambusher**

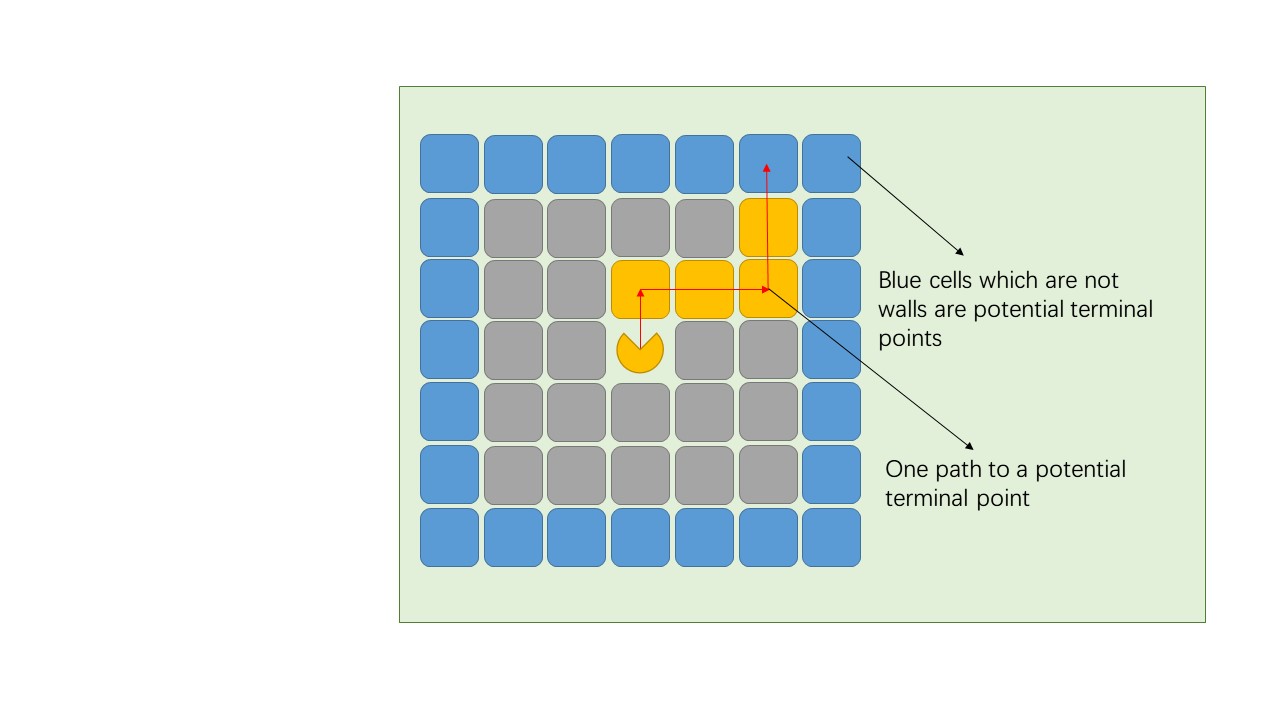


*Figure 5.6.3 Ambusher*

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 5.6.4 Potential terminal points*

1. //Example pseudo code for getAllPotentialPaths()
2. Function getAllPotentialPaths() {
3. List listlistOfPotentialTerminalPoints = list**<Vector3>**
4. For (int i =self.position.x – 3; i**<**= self.position.x + 3; i ++){
5. If(cell at (i, self.position.y + 3) is not wall), add it to listOfPotentialTerminalPoints
6. If(cell at (i, self.position.y - 3) is not wall), add it to listOfPotentialTerminalPoints
7. }
8. For (int i =self.position.y – 3; i**<**= self.position.y + 3; i ++){
9. If(cell at (self.position.x + 3, i) is not wall), add it to listOfPotentialTerminalPoints
10. If(cell at (self.position.x - 3, i) is not wall), add it to listOfPotentialTerminalPoints
11. }
12. For each point in listOfPotentialTerminalPoints{
13. Path = aStarSearch(point)
14. listOfPotentialPaths.add(Path)
15. }
16. }
18. Function aStarSearch(Vector3 end){
19. Vector3[ ] path
20. Put position of PacMan into a OPEN list, OPEN list stores unvisited cells
21. A CLOSE list will be used to store visited cells
22. 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))
23. While(OPEN != Null){
24. CurrentPoint = the point which has the least f(n)
25. If(currentPoint == end), then break
26. For (every non-wall neighbour cell X of current cell){
27. Compute its f(X)
28. If(this cell is in OPEN){
29. If(new f(X) **<** **old** f(n) in the OPEN list){
30. Set n as the parent of X
31. Update f(n) in the OPEN list
32. }
33. }else if (this cell is in CLOSE) {
34. Continue
35. }else if ( X is not in OPEN or CLOSE) {
36. Set n as the parent of X
37. Compute f(X)
38. Add X to OPEN list
39. }
40. }
41. Remove n from OPEN list and add it to CLOSE list
42. }
43. path = points that move along parent cell from end point to the start point
44. Return path
45. }

* *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 move to the predicated destination.

1. //Example of pseudo code for behaviour of an Ambusher
2. While(True) {
3. Get all potential paths of the PacMan
4. For each path in potential paths list {
5. Expected profit = number of PacDots in path + 10 \* number of Items in path
6. }
7. Predicated path = potential path with most profit
8. Set the end of predicated path as Ambusher’s next destionation
9. }

* **Thief**

The final kind of ghost is called Thief. After it is activated, it will try to move away from PacMan instead of hunting it. This kind of ghost can 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 (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] |

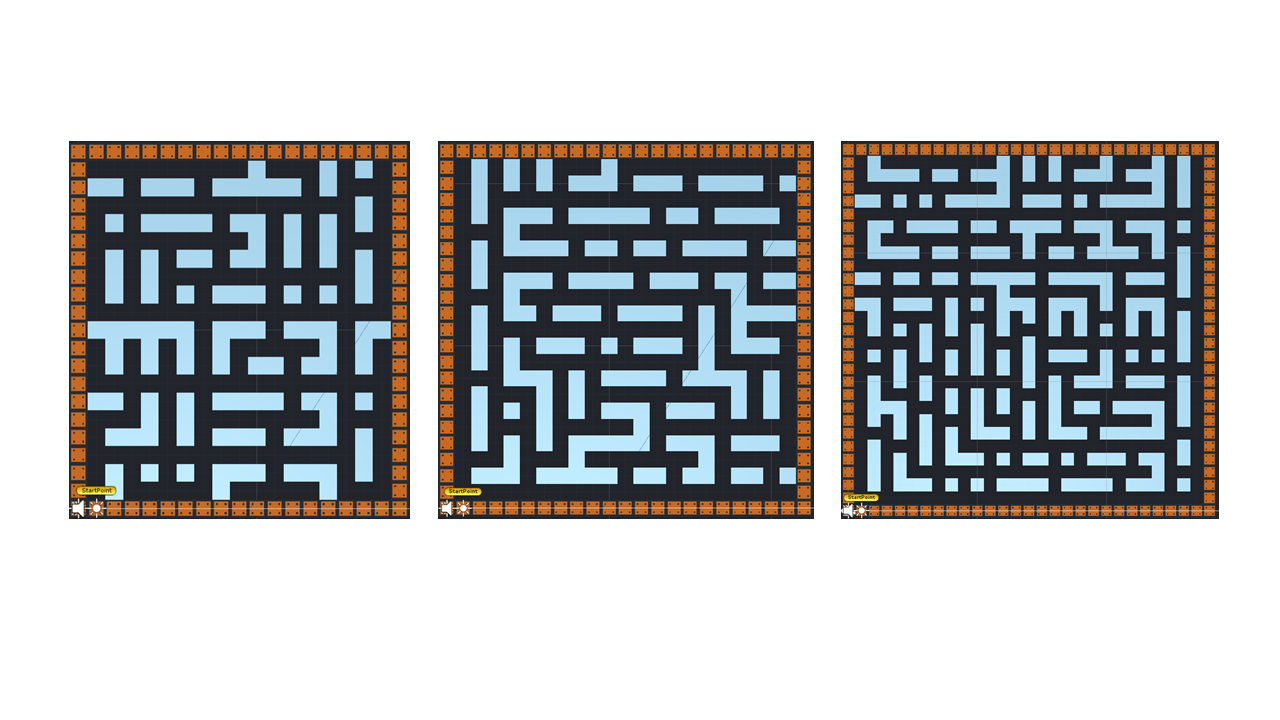
* *OnColliderEnter(Collider)* function in Thief will be overridden. When it collides with a PacDot, it will collect the PacDot. If it collides with PacMan, the Thief will be destroyed and give all stolen PacDots to PacMan.
* *DecideMoveDirection()* will create a sequence of direction including up, down, 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.

1. //Example of pseudo code for behaviour of a Thief
2. Override OnColliderEnter(Collider){
3. If (Collider.type == PacDot){
4. NumberOfStolenPacDots ++
5. Destory that PacDot
6. }Else if (Collider.type == PacMan) {
7. Self.destroy()
8. PacMan.points = PacMan.points + numberOfStolenPacDots
9. PacMan.energy = PacMan.energy + numberOfStolenPacDots
10. }
11. }
12. Function decideMoveDirection() {
13. Vector3 relativePosition = PacMan.position – Thief.position
14. If(relativePosition.x**>**=0&&relativePosition.y**<**=0&&relativePosition.x.abs**<** **relativePosition.y.abs**) {
15. MoveDirectionSequence = [up, left, right, down]
16. }
17. If(relativePosition.x**>**=0&& relativePosition.y**<**=0&& relativePosition.x.abs**>**relativePosition.y.abs) {
18. MoveDirectionSequence = [up, left, right, down]
19. }
20. If(relativePosition.x**>**=0&&relativePosition.y**<**=0&& relativePosition.x.abs**>** relativePosition.y.abs) {
21. MoveDirectionSequence = [up, left, right, down]
22. }
23. If(relativePosition.x**>**=0&&relativePosition.y**<**=0&& relativePosition.x.abs**<** **relativePosition.y.abs**) {
24. MoveDirectionSequence = [up, left, right, down]
25. }
26. If(relativePosition.x**<**=0&&relativePosition.y**<**=0&& relativePosition.x.abs**<relativePosition.y.abs**) {
27. MoveDirectionSequence = [up, right, left, down]
28. }
29. If(relativePosition.x**<**=0&&relativePosition.y**<**=0&& relativePosition.x.abs**>**relativePosition.y.abs) {
30. MoveDirectionSequence = [right, up, down, left]
31. }
32. If(relativePosition.x**<**=0&&relativePosition.y**>**=0&& relativePosition.x.abs**<** **relativePosition.y.abs**) {
33. MoveDirectionSequence = [down, right, left, up]
34. }
35. If(relativePosition.x**<**=0&&relativePosition.y**>**=0&& relativePosition.x.abs**>** relativePosition.y.abs) {
36. MoveDirectionSequence = [left, down, up, left]
37. }
38. }
39. Function Update() {
40. DecideMoveDirection()
41. Move in direction MoveDirectionSequence[0]
42. If moveDirection is blocked by walls, moveDirection = MoveDirectionSequence[1]
43. If moveDirection is blocked by walls, moveDirection = MoveDirectionSequence[2]
44. If moveDirection is blocked by walls, moveDirection = MoveDirectionSequence[3]
45. }
46. **Realisation**

In the implement stage, most game objects meet the requirements of design and all expect functions are realised in the final product. Some modifications have also been made to the ‘Item’ game object to make it fit requirements of this game better. Some extra components including animation system and particle system are also added to provide better gameplay experience. This section will discuss the implement for each component in detail.

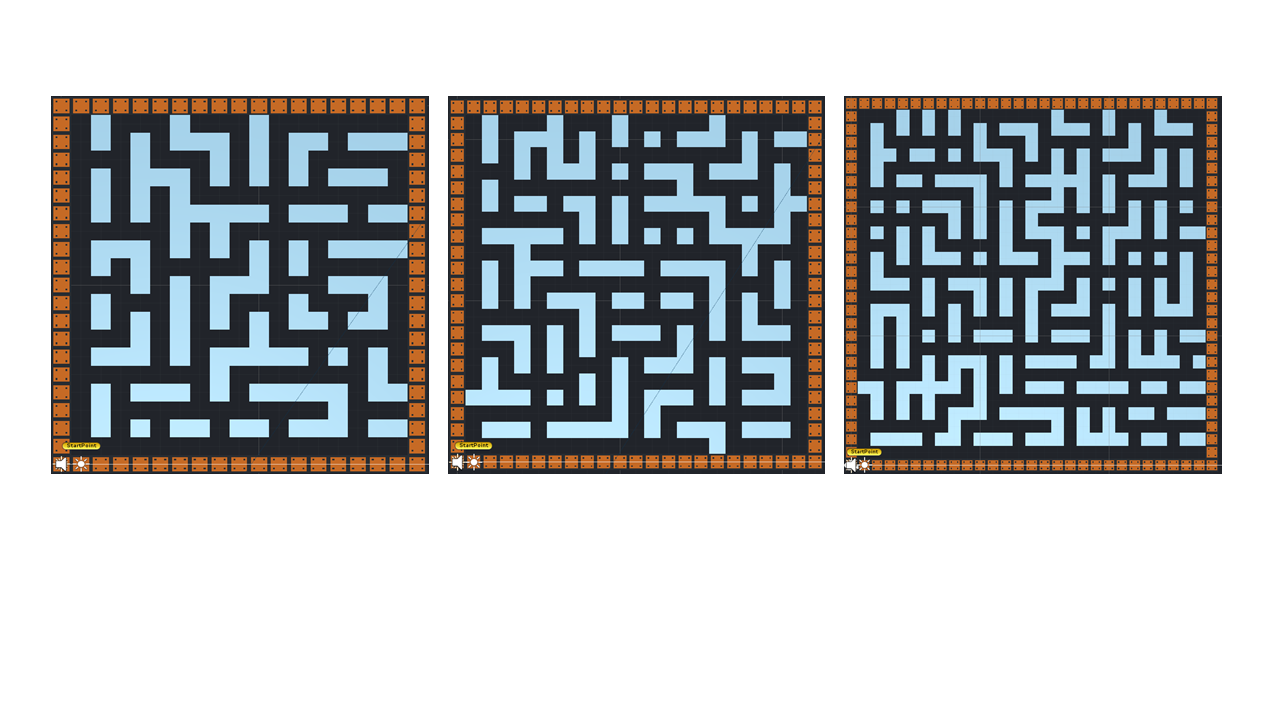
* 1. **Maze Generation**

Since the maze generation system is the basic and key component for the game, it should be completed first. All three algorithms mentioned in the ‘Design’ section are implemented in the Unity environment. An abstract super class “MazeGenerator” is created as the parent of all maze generators. The MazeGenerator class also contains necessary API for the generation of mazes. Then, three C# scripts are created to implement the logic of each maze generation algorithm. Finally, an empty GameObject is created in the game scene and three MazeGenerator scripts are attached to that object. When *MazeGenerator.GenerateMaze(width, height)* function is called, mazes with different types and sizes will be generated. Figure 6.1.1, 6.1.2 and 6.1.3 shows mazes generated by different algorithms. Complete codes for maze generation can be found in appendices.



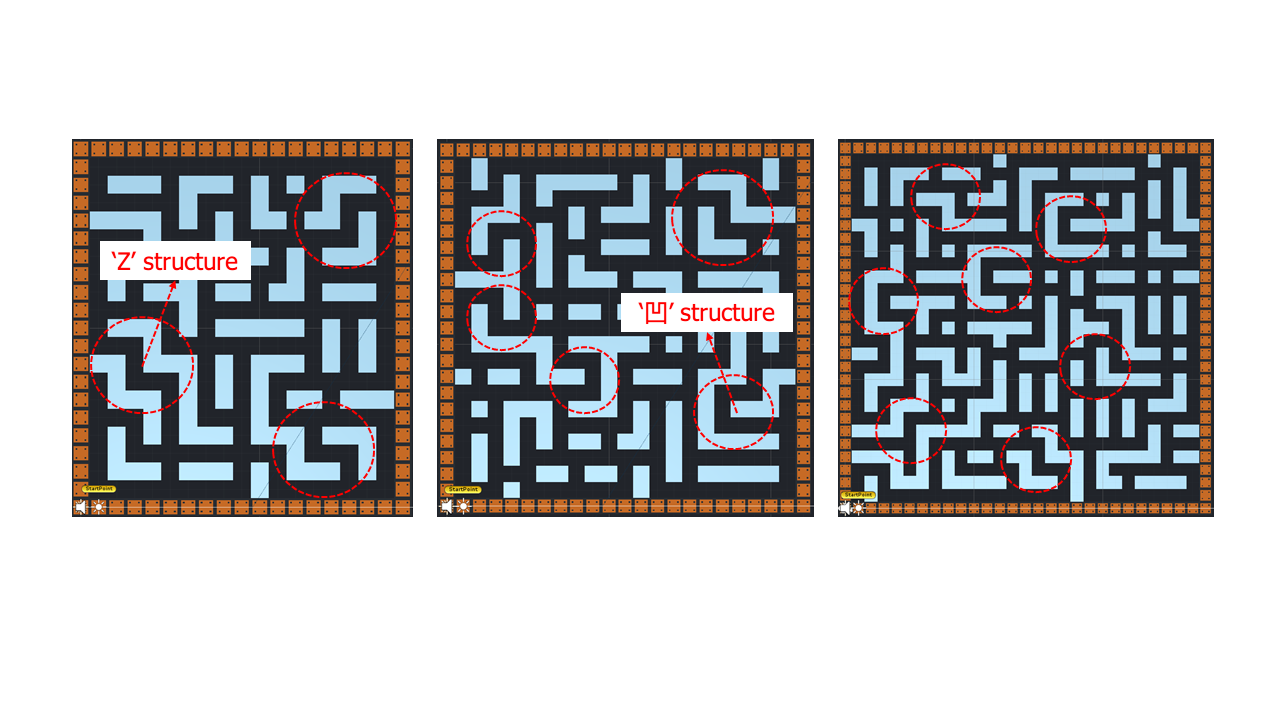
*Figure 6.1.1 Mazes generated by recursive division*

Mazes in Figure 6.1.1 are generated by recursive division. Mazes generated by this kind of algorithm usually consist of many short vertical and horizontal walls. There are also many long, straight roads, which makes this maze easy to solve. Therefore, this kind of maze generator are used to create the simplest maze in the game.



*Figure 6.1.2 Mazes generated by randomized Prim’s algorithm*

These mazes shown in Figure 6.1.2 make use of the randomized Prim’s algorithm. Compared with recursive division, this algorithm can generate mazes which have more corners. Players need to adjust PacMan’s direction more frequently and try to find different way to solve the maze. This maze generator is used to provide mazes with medium difficulty.

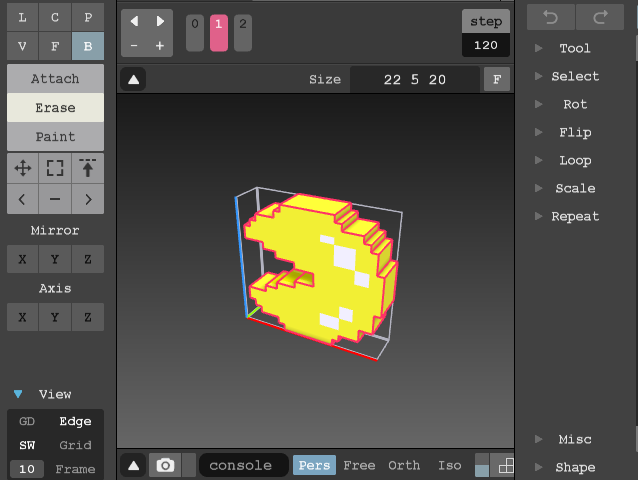


*Figure 6.1.3 Mazes generated by recursive backtracker algorithm*

Mazes generated by recursive backtracker algorithm contain many ‘Z’ and ‘凹’ structure (Figure 6.1.3). Both ‘Z’ and ‘凹’ structure have a feature that if a PacMan enter this structure, there will be only one exit. In this game, ghosts will be placed in the maze to hunt PacMan. Therefore, such kind of structure can be dangerous because PacMan will have no other ways to run. With this feature, recursive backtracker algorithm is used to generate those most difficult mazes.

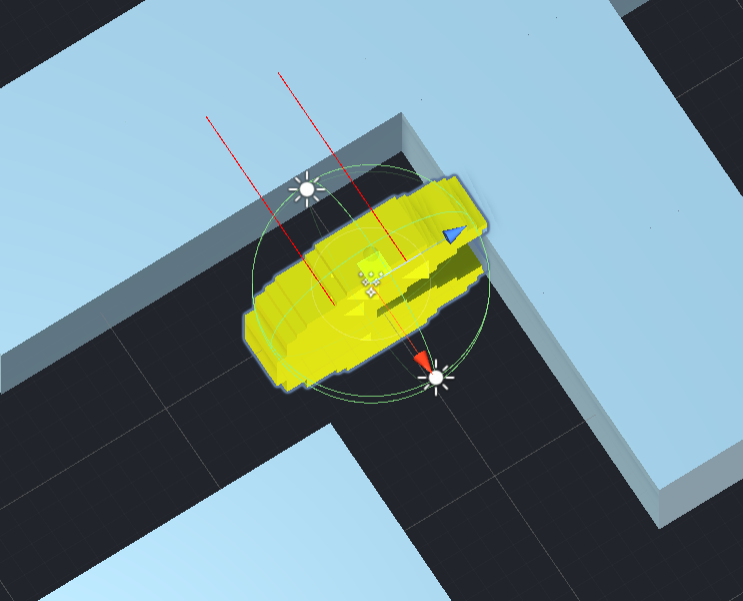
* 1. **Player**

Player (PacMan) is implemented according to the design document. Pseudo codes for the logic of PacMan can be found in ‘Design’ section. The 3D model for the PacMan is created using MagicaVoxel [1]. Collider and Linecast are used to detect the interaction between PacMan and other GameObject. Several C# scripts are attached to the PacMan to handle user’s input and PacMan’s game logic. Complete codes for PacMan can be found in appendices.

**

*Figure 6.2.1 Making of PacMan in MagicaVoxel*

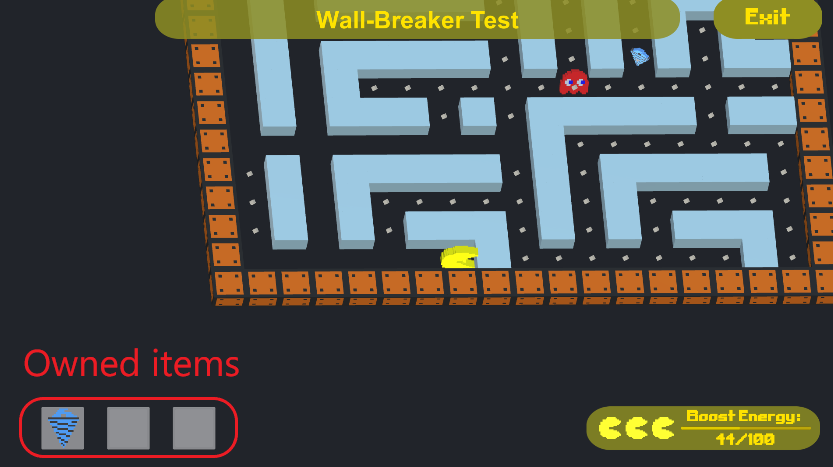
Figure 6.2.1 shows the making of PacMan’s 3D model in MagicaVoxel. MagicaVoxel is a modelling tool that provides modelling, rendering, painting and animation functions. All models used in the game are created using MagicaVoxel.



*Figure 6.2.2 PacMan in Unity game engine*

Figure 6.2.2 shows the PacMan in the Unity game engine. The red lines represent the Linecast in Unity. The Linecast is used to detect the existence of other colliders and it is used when PacMan tries to change its direction. For example, in figure 6.2.5, PacMan tries to turn left but Linecast finds that there is a wall (collider)on the left. Then, Linecast will return this result to the script and script will not allow PacMan to turn left.

The green lines represent the sphere collider of the GameObject. The collider is used to detect the interaction between PacMan and other GameObjects such as Ghosts and Items. When PacMan collides with another collider or trigger, the *OnColliderEnter(Collision)* function in PacMan’s script will be called [2]. Information about this collision and the GameObject is also passed to the Player script and PacMan will have corresponding actions. For example, when PacMan collides with a PacDot, the Player script will firstly get information about collision and PacMan knows that it collides with a PacDot. Then, PacMan will get one PacPoint and Boost Energy. Finally, the PacDot will be destroyed from the game scene.



*Figure 6.2.3 Owned items*

When player collides with an item object in the maze and player has an empty item box, PacMan will pick this item up and store it (except Energy Pellet) in the item box (figure 6.2.3). Player can use items after obtaining them. Each item has a unique function and details about items will be discussed in ‘Items’ section.

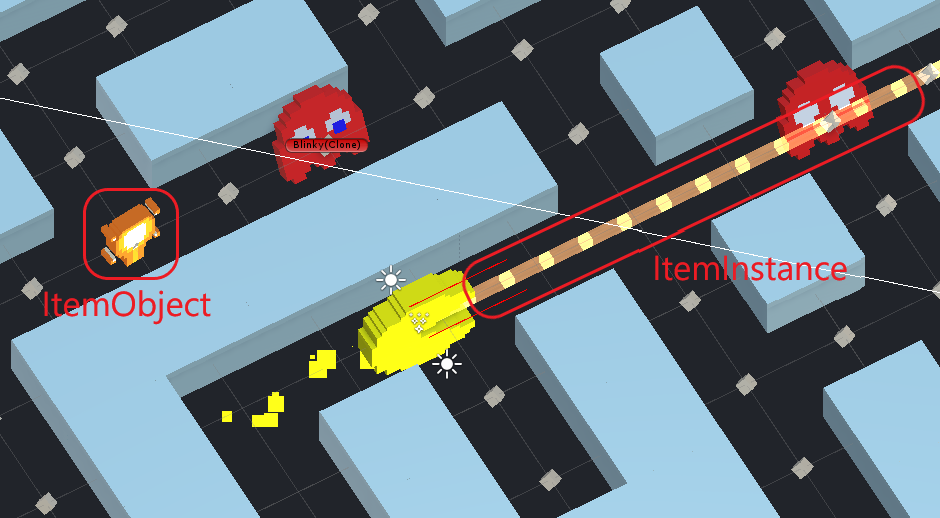


*Figure 6.2.4 Boost and boost energy*

Player can press ‘Space’ key and consume boost energy to make PacMan move faster. Figure 6.2.4 shows PacMan in boost mode and boost energy. PacMan can get boost energy by eating PacDot or energy pellet. Each PacDot provides 1 boost energy and each energy pellet provides 100 boost energy. In the boost mode, PacMan can move in 1.5 times of its original speed and each second in boost mode will consume 50 boost energy.

* 1. **Items**

During the implement stage, the design of items has been revised to meet requirements of the game better. In the design stage, a class: Item is designed to implement the functions of items. In the final product, Item class is removed and two class: ItemObject and ItemInstance are added to implement the item system. ItemObject describes GameObjects which have not been picked up by PacMan. ItemInstance describes GameObjects which are generated after using corresponding items. Figure 6.3.1 shows ItemObject and ItemInstance for Laser.



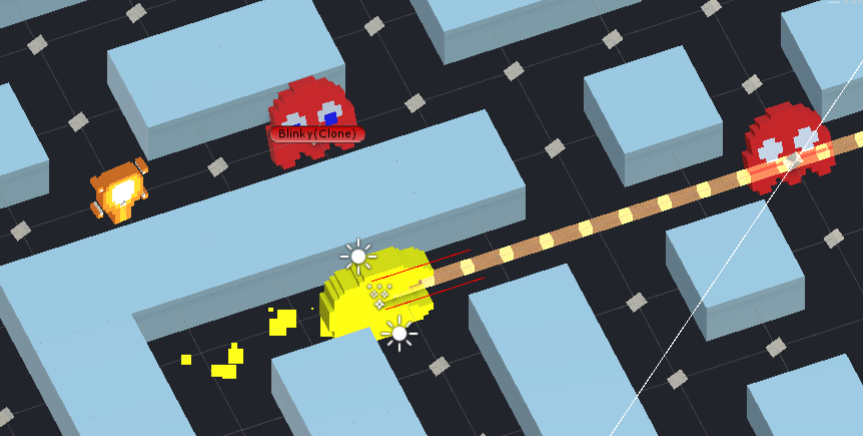
*Figure 6.3.1 ItemObject and ItemInstance*

The following table shows the structures for ItemObject and ItemInstance class. There are also 5 subclass of ItemObject: LaserObject, WallBreakerObject, PortalObject, PelletIObject and GrenadeObject. All of them inherit ItemObject and have their unique 3D model. Because EnergyPellet does not need an ItemInstance, there are 4 subclass of ItemInstance: LaserInstance, WallBreakerInstance, PortalInstance and GrenadeInstance.

|  |  |  |
| --- | --- | --- |
| **ItemObject** |  | **ItemInstance** |
| Name: String  Model: 3DModel  position: Vector3  trigger: Collider3D  unlocked: Bool | Name: String  Model: 3DModel  Position: Vector3  Collider: Collider3D |
| setPosition(): void  OnTriggerEnter(Collider): void  GenerateItemInstance(): void | OnColliderEnter(Collision): void |

Firstly, ItemObjects are placed in the maze. Then, if PacMan collides with the ItemObject and there is an empty item box, PacMan will pick it up and store it in an item box. When player presses corresponding number key, *GenerateItemInstance()* will be called to generate the ItemInstance. Finally, this ItemObject will be deleted from item box and ItemInstance in the game scene starts to interact with other GameObject.

Functions of items are not changed. ItemInstance is now responsible for functions of items and pseudo codes for each item has been discussed in ‘Design-Items’ section. The implement of these items follows their pseudo codes. Therefore, only ItemObject, ItemInstance and key methods used in the items will be discussed in this section.



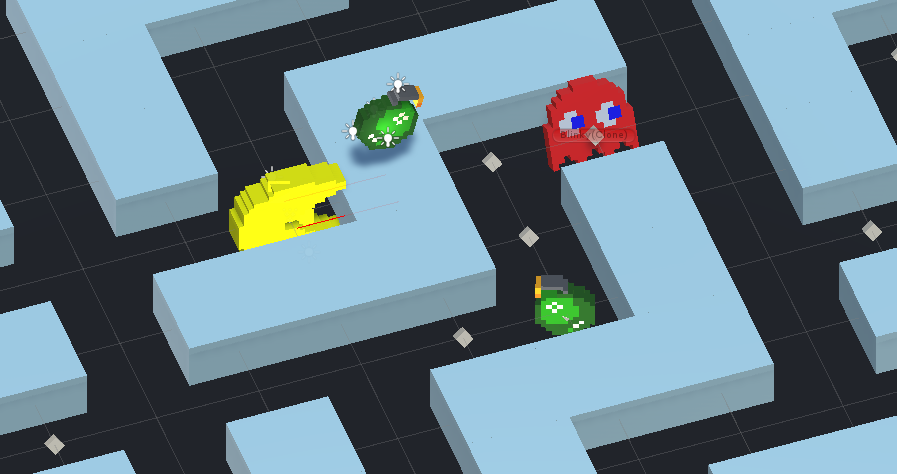
*Figure 6.3.2 Laser*

Figure 6.3.2 shows Laser item in the game, including LaserObject and LaserInstance. When *LaserObject.GenerateItemInstance()* is called, LaserInstance object will be generated in front of the PacMan and move forward in a high speed. When LaserInstance collides with any other collider, *OnTriggerEnter(Collider)* and *OnColliderEnter(Collision)* will be called. If LaserInstance collides with a wall, the part of laser that collides with wall will be destroyed. If LaserInstance collides with a Ghost, the Ghost will be disabled for 15 seconds. If LaserInstance collides with other GameObjects, nothing will happen. This item gives PacMan an ability to disable many ghosts with one item. However, ghosts are not destroyed so PacMan leave these ghosts before they awake.



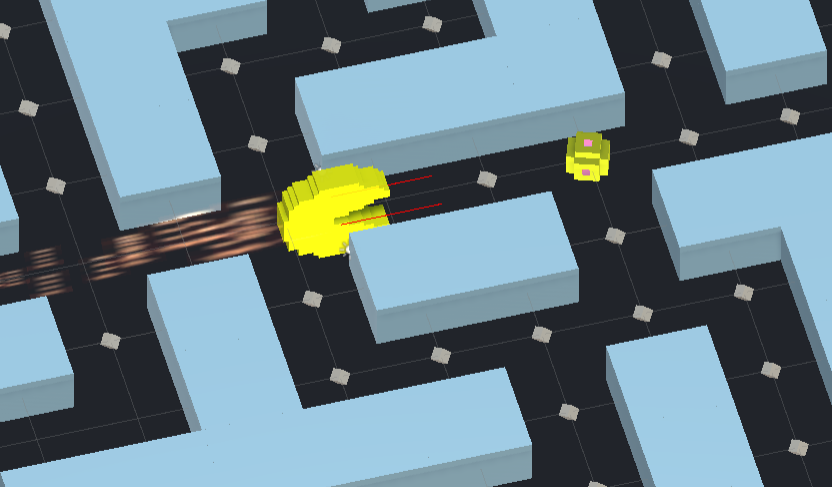
*Figure 6.3.3 Wall-Breaker*

Wall-Breaker item, including WallBreakerObject and WallBreakerInstance are shown in figure 6.3.3. When *WallBreakerObject.GenerateItemInstance()* is called, WallBreakerInstance object will be generated in front of the PacMan and move with PacMan. When WallBreakerInstance collides with any other collider, *OnColliderEnter(Collision)* will be called. If WallBreakerInstance collides with a wall, the wall and the WallBreakerInstance will be destroyed. If WallBreakerInstance collides with a Ghost, the Ghost and the WallBreakerInstance will be disabled for 15 seconds. If LaserInstance collides with other GameObjects, nothing will happen. Wall-Breaker is a powerful weapon and it has two different usages: destroy wall or ghost. Player should consider carefully before using it.



*Figure 6.3.4 Grenade*

Figure 6.3.4 shows GrenadeObject and GrenadeInstance in the game scene. When a Grenade is used, a GrenadeInstance object will be generated in the position of PacMan. The GrenadeInstance will fly over the wall in front of PacMan and then move forward. When the GrenadeInstance is moving, if it collides with a Ghost, both Grenade and the Ghost will be destroyed. If GrenadeInstance collides with a wall, the GrenadeInstance will be destroyed and a new GrenadeObject will be created in the empty cell of the collision. PacMan needs to move to the collision point to get grenade back. This item can help PacMan to destroy ghosts from a safe position but it may be difficult to aim the target.

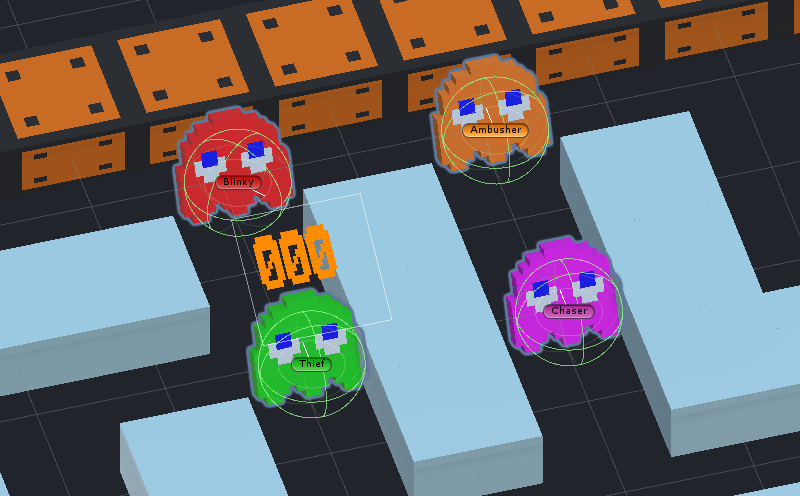


*Figure 6.3.5 Energy Pellet*

Figure 6.3.5 shows the ItemObject of energy pellet. Energy pellet has no ItemInstance because energy pellet will provide PacMan 100 boost energy and it will always be used when PacMan collides with it. In addition, the energy provided by the pellet will ignore the energy capacity of PacMan so player can make use of the pellet without considering PacMan’s current energy.

* 1. **Ghosts**

In the implement stage, 4 different kinds of ghosts: Blinky, Chaser, Ambusher and Thief are realised. These 4 kinds of GameObject inherit the Ghost class. Blinky, Chaser and Ambusher GameObject override the *GetNextEnd()* function and Thief GameObject overrides the *OnColliderEnter(Collision)* function. This section will discuss the implement of Ghost class first and then 4 kinds of ghost will be described.



*Figure 6.4.1 Ghosts in Unity game engine*

3D model of Ghost is also made by MagicaVoxel. 4 kinds of Ghost shares one 3D model but they have different material and colour to show their identity. The implement of Ghost class follows the design in the ‘Design-Ghost’ section.

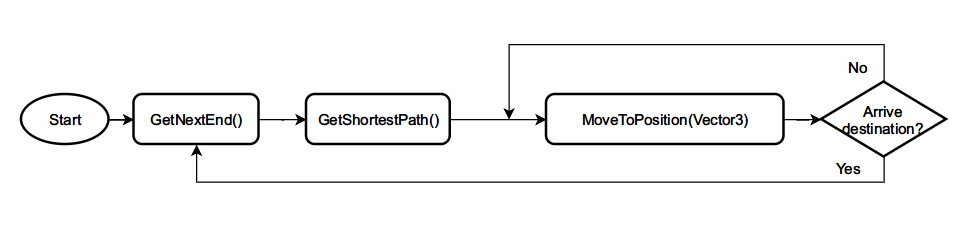
* **Interactions between Ghost and other GameObject**

The interaction between Ghosts and other GameObjects is implemented using collider and *OnColliderEnter(Collision)* function. In figure 6.4.1, the green lines around Ghosts represent the collider. When the collider collides with other GameObjects’ collider, the *OnColliderEnter(Collision)* in the Ghosts’ script will be called and different ghosts will take different actions according to the information of the collision.

For Blinky, Chaser and Ambusher, if they collide with PacMan, PacMan will be destroyed and lose one life. For a Thief, it overrides the *OnColliderEnter(Collision)* function so if it collides with a PacMan, the Thief will be destroyed and PacDots stolen by the Thief will be given back to PacMan.

When a Ghost collides with other GameObject (except PacMan), the other GameObject will handle this collision so codes in Ghost class does not need to react to this collision. For example, when a Ghost collides with a Wall-Breaker, the codes in Wall-Breaker will handle this collision and destroy the Ghost.

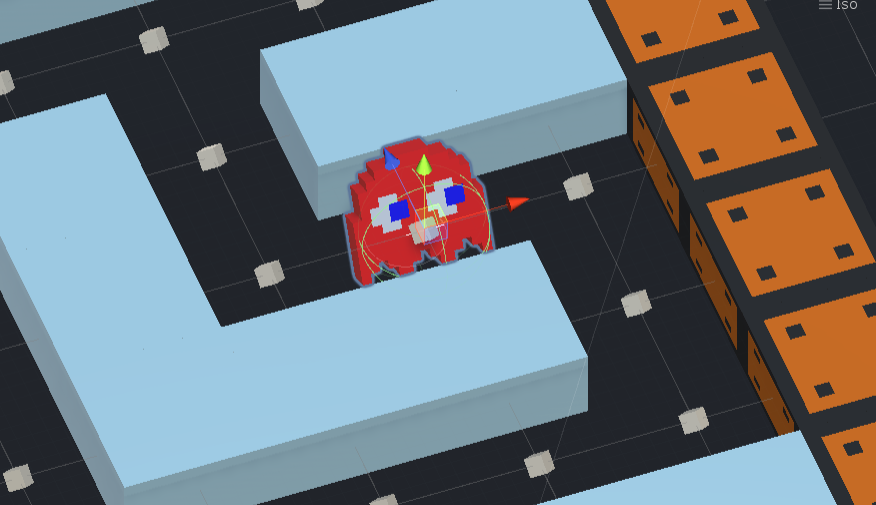
* **Movement of Ghost**

****

*Figure 6.4.2 movement of ghost*

Figure 6.4.2 shows the logic of movement of ghost. *GetNextEnd()* is an abstract function in Ghost class and every ghost should implement this function. Each ghost implements its unique logic to find its next destination, which will be discussed later. Then, a breadth-first search will be applied to find the shortest path from the position of ghost to its destination. The result of path will be stored in a *List<Vector3>*. After getting the path, *Transform.Translate(Vector3)* is called to move the ghost along the path and reach the destination. Finally, the next destination will be calculated again and ghost will start to move again. By running this loop, ghosts can run in the maze and realise their designed action logic.

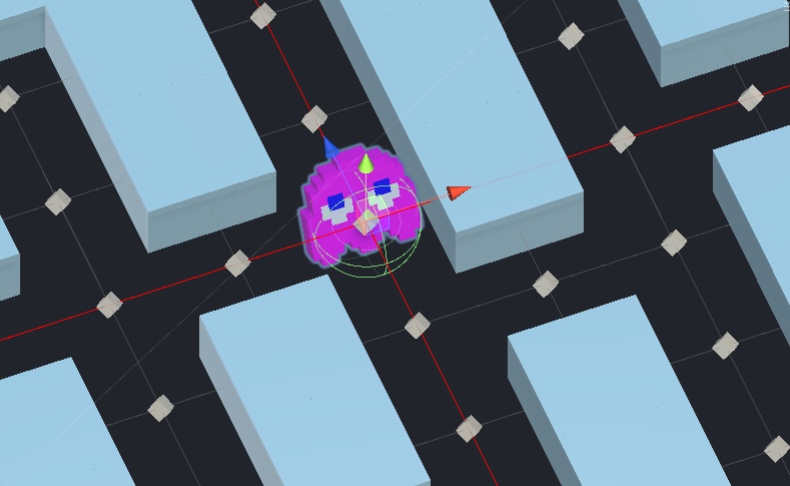
* **Implement of Blinky**



*Figure 6.4.3 Blinky*

The Blinky is the simplest ghost in the game. It will choose a destination in the maze randomly and if this position is not occupied by a wall, the position will be Blinky’s next destination. A *Random.Range(Int, Int)* function is used to generate the random position.

* **Implement of Chaser**

****

*Figure 6.4.4 Chaser*

Chaser is a kind of ghosts that will chase the PacMan when PacMan is in its sight. To implement this, *Physics.Linecast(Vector3, Vector3, HitInfo, LayerMask)* function will be used to cast 4 detection lines to detect the position of PacMan (red lines in figure 6.4.4) [3]. Two Vector3 variables in the parameter list indicate the start position and end position of the line. They will be determined according to the position of Ghost and the detection distance in the game. HitInfo provides a container for the information about any possible hit. LayerMask indicates what layers will interact with the line. For a Chaser, detection line will be able to interact with PacMan and wall layer. In addition, one line can only hit one GameObject at the same time so Chaser cannot detect PacMan behind walls.

The Chaser will keep casting lines to detect PacMan. When PacMan is found, the position of the PacMan will be set as the next destination of the Chaser and Chaser will start to move. The speed of Chaser is 1.25 times of PacMan’s speed so player should learn to use walls to avoid Chaser.

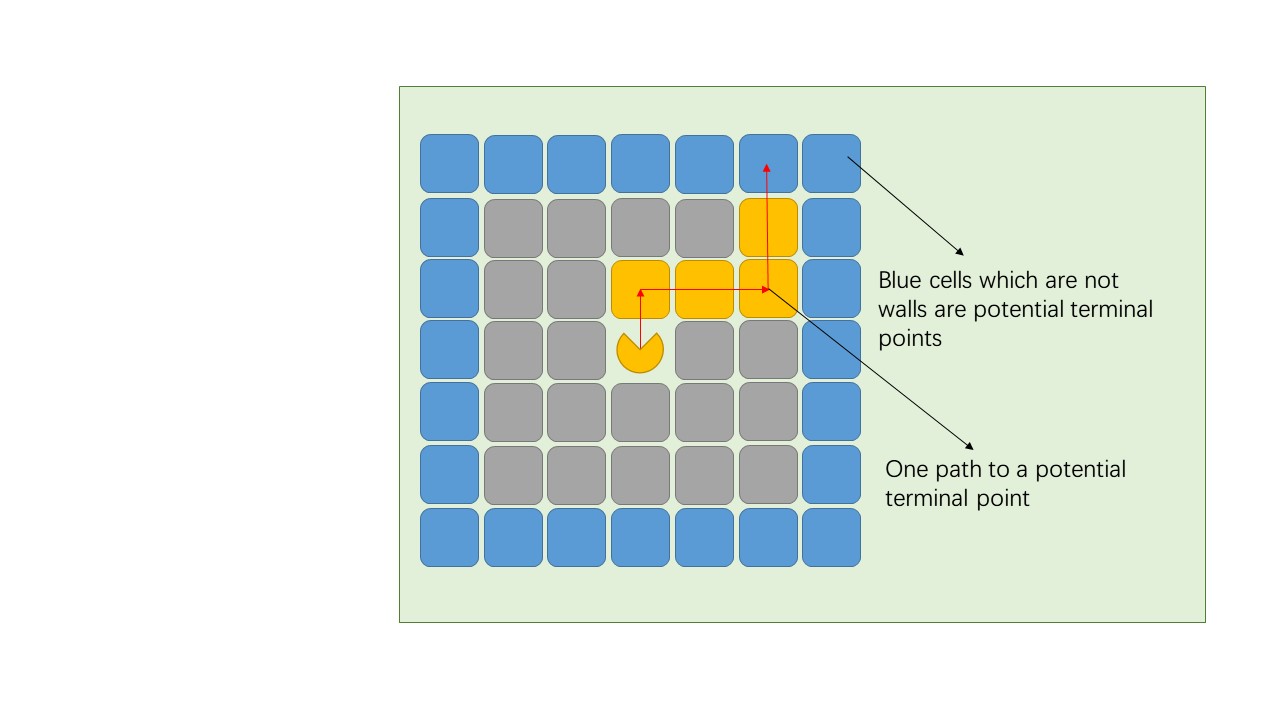
* **Implement of Ambusher**

****

*Figure 6.4.5 Ambusher*

An ambusher set its next destination by predict the movement of PacMan. In this game, the objective of PacMan is to collect enough PacDots and leave the maze. Therefore, the predication of PacMan’s movement is based on the assumption that PacMan tends to go to those positions where they can collect more PacDots.

The predication of PacMan’s movement is divided into 3 steps: find potential terminal points, find all paths and find the most valuable path.

**

*Figure 6.4.6 Find potential terminal points*

Pseudo codes for 3 steps have been discussed in the ‘Design-Ghost-Ambusher’ section. Figure 6.4.6 shows how to find potential terminal points. Then, paths from the position of PacMan to all those terminal points will be calculated. Each path has a “value” and the value equals to the number of PacDots on the path. Finally, a path will be chosen randomly from paths which have most “value” and the end of this path will be Ambusher’s next destination.

* **Implement of Thief**

**

*Figure 6.4.7 Thief*

Different from other Ghosts, the objective of Thief is not to hunt PacMan and Thief will always try to leave PacMan. Thief will set its next destination according to PacMan’s position and its current position. Pseudo codes for how Thief finds it next destination have been given in ‘Design-Ghost-Thief’ section.

In addition, an area for text is also attached to the Thief GameObject. The text will show how many PacDots are eaten by this thief. A small piece of codes is also attached to text so the text area will also face to the Camera.

* 1. **User Interface**

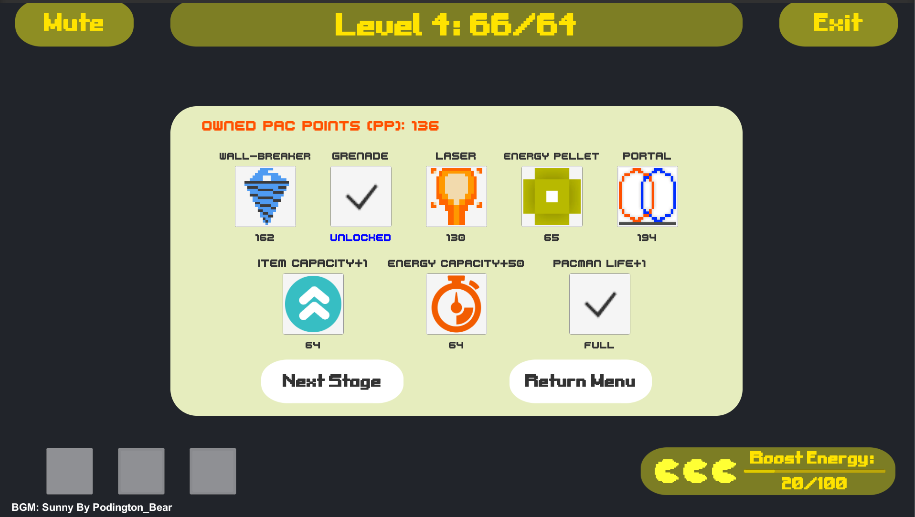
In the implement stage, user interface is created using Unity canvas system. Firstly, a canvas GameObject will be created as a container of UI elements. Then UI components such as buttons will be added to the canvas. Figure 6.5.1 shows an example of layout of canvas and UI components. In this example, because the shop interface will move from the outside of the screen, it is placed outside the canvas and it will be moved inside when needed.

**

*Figure 6.5.1 Canvas and UI components*

Finally, some interactable components such as buttons will be linked with some C# script functions. For example, Exit button will be linked with *LoadScene(“MainMenu”)* function so that player can go back to main menu by clicking this button.

The following figures show some examples of user interface in the final product.

*Figure 6.5.2 Examples of final UI*

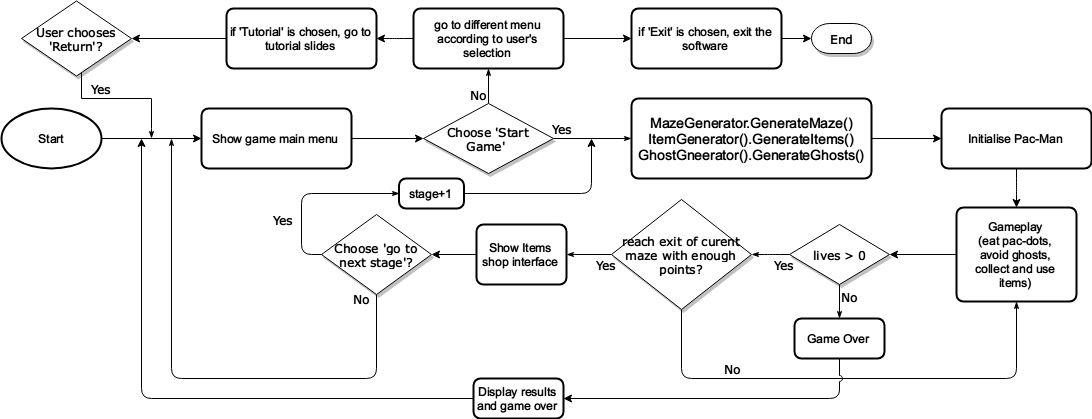
* 1. **ItemGenerator and GhostGenerator**

After all Items and Ghosts are ready for the game, ItemGenerator and GhostGenerator are created to generate items and ghosts in the maze. The basic logic for these two generator is to divide a maze into several 5\*5 areas. In each square, one item and one ghost will be placed in an empty maze cell. In addition, for a maze, there will be at most one Portal item and one Thief ghost.

The key technology used in these two generator is the *Instantiate(GameObject, Vector3)* function. This function is a Unity build-in function which is used to generate a pre-defined GameObject in the game scene. Codes about these two generators can be found in the appendices.

* 1. **Game Loop**

After all other game components are created properly, GameManager will be used to manage all these components and control the game loop.



*Figure 6.7.1 Game Loop*

The game loop in the final product follows the flowchart (figure 6.7.1) which has been discussed in ‘Design-GameSceneManager’ section. The following is part of codes in the GameSceneManager class. These codes are used to clear last stage and start next stage. The complete codes will be attached with the appendices.

1. **public** **void** PacManArriveEndPoint() { //PacMan reach end point
2. **if** (pacDotsEatenByPlayer>=pacDotsNeeded) {  //PacMan has enough points
3. soundManager.PlayStageClearAudio();
4. soundManager.DisableBoostAudio();
5. soundManager.DisableBreakerAudio(); //stop all sounds
6. ClearLastStage();   //clear last stage
7. itemAndShopGM.SetShopMenuState(pacDotsEatenByPlayer);
8. uiGmScript.StageMenuEnter();    //show shop UI
9. } **else** {    //PacMan reach end point without enough points
10. levelText.GetComponent<Animator>().SetTrigger("playWarningAnim");
11. }
12. }
14. **public** **void** ClearLastStage() {
15. Debug.Log("Start Clear Last Stage");
16. ClearAllGhost();
17. ClearLastMaze();
18. ClearAllPacDots();
19. ClearPlayerAndItems();
20. }
21. **public** IEnumerator AfterStageClearMenuReturn(**float** time) {
22. ClearPlayerAndItems();              //Shop UI disappear. Start next stage
23. Destroy(planeClone);    //Destroy old background
24. yield **return** **new** WaitForSeconds(time);
25. BuildMaze();        //generate a maze
26. GeneratePacDot();   //generate PacDots
27. GeneratePlane();    //generate new background
28. endPoint=Instantiate(endPointPrefab,    //generate new end point
29. **new** Vector3(mazeWidth-2, 0, mazeHeight-2),
30. **new** Quaternion());
31. pacMan=RespawnPacMan(**new** Vector3(1, 0, 1)); //generate new PacMan
32. virtualCamera1.Follow=pacMan.transform;     //adjust camera
33. itemObjectsList= itemGenerator.GenerateItemObejcts();// generate items
34. **if** (level>3) {
35. ghostsList=ghostGenerator.GenerateGhosts(); //generate ghosts
36. }
37. }

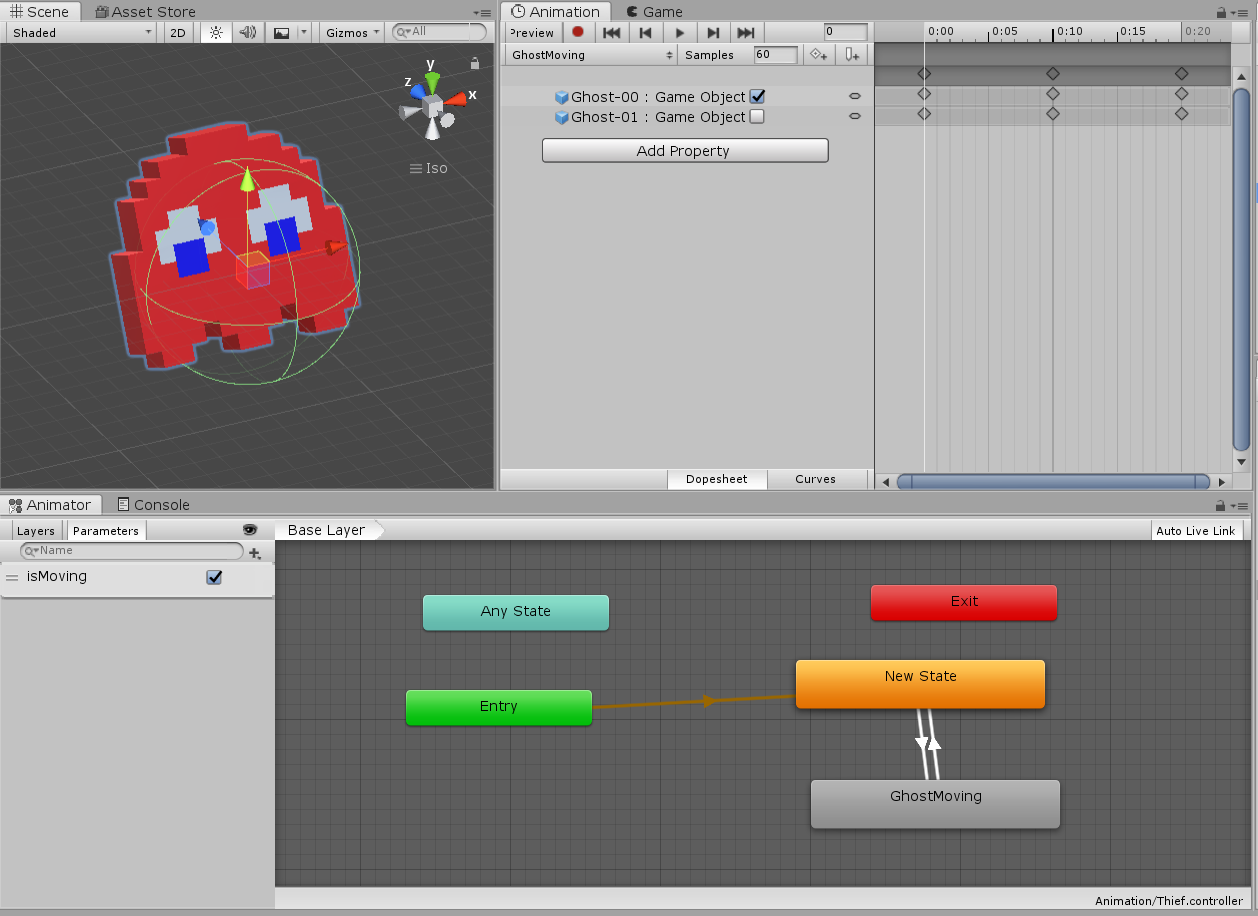
After game loop is formed, all necessary components for this game have been completed. Players can press the ‘Start’ button to start and play this game.

* 1. **Extra features**

After all other components are basically completed, several extra features are added to the game in order to provide better gameplay experience.

* **Animation system**

Animation systems are added to PacMan, Ghost and the user interface. The animation is implemented using Unity animation system. For user interface, the animation is played when the interface slips into the screen. For PacMan and Ghosts, an animator component is attached and each animator has a trigger called “isMoving”. When they are moving, the trigger will be set to “true” and a moving animation clip will be played. Figure 6.8.1 shows the animation system in Unity.



*Figure 6.8.1 Animation system*

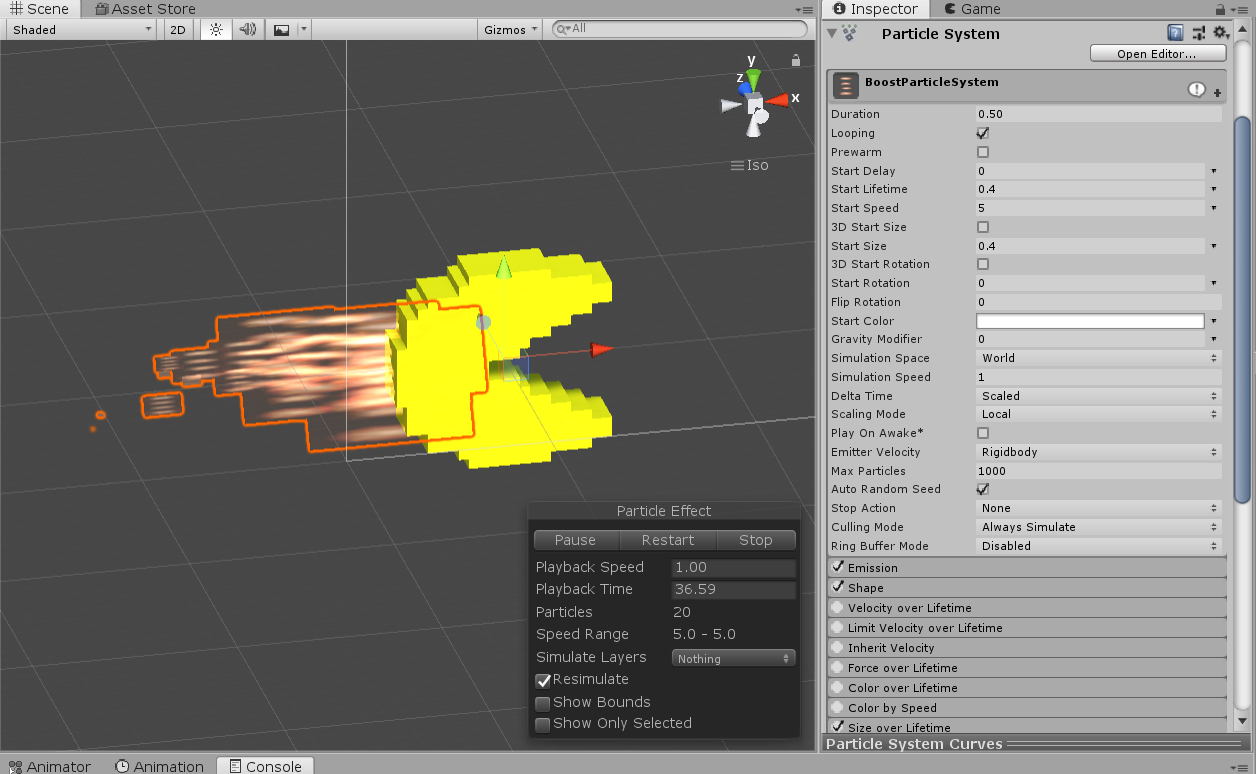
* **Sound system**

Sound system is also added to the game. All sounds are divided into background music and sound effects. A new object called SoundManager is created in the game scene. It contains a C# script that manages all sound effect in the game. When a game is started, a random music will be chosen from the background music list and played. During the game, if needed, objects in the game can ask the manager to play a certain sound effect. For example, when a player is using a laser, a laser sound effect will be played. the following table shows the structure of SoundManager.

|  |
| --- |
| **SoundManager** |
| BgmList: List<AudioClip> |
| PlayBGM(): void  PlaySoundEffect(String): void |

* **Particle System**

In the final product, particle system is used to improve the visual effects of the game. Unity has a built-in particle system. Developers can build their own particle effects by providing resources for the particle and adjusting attributes of the particle system. Particle system is used to improve the visual effect of boosting and death of game objects. It is also used in the loading scene. Figure 6.8.2 shows the particle system in Unity.



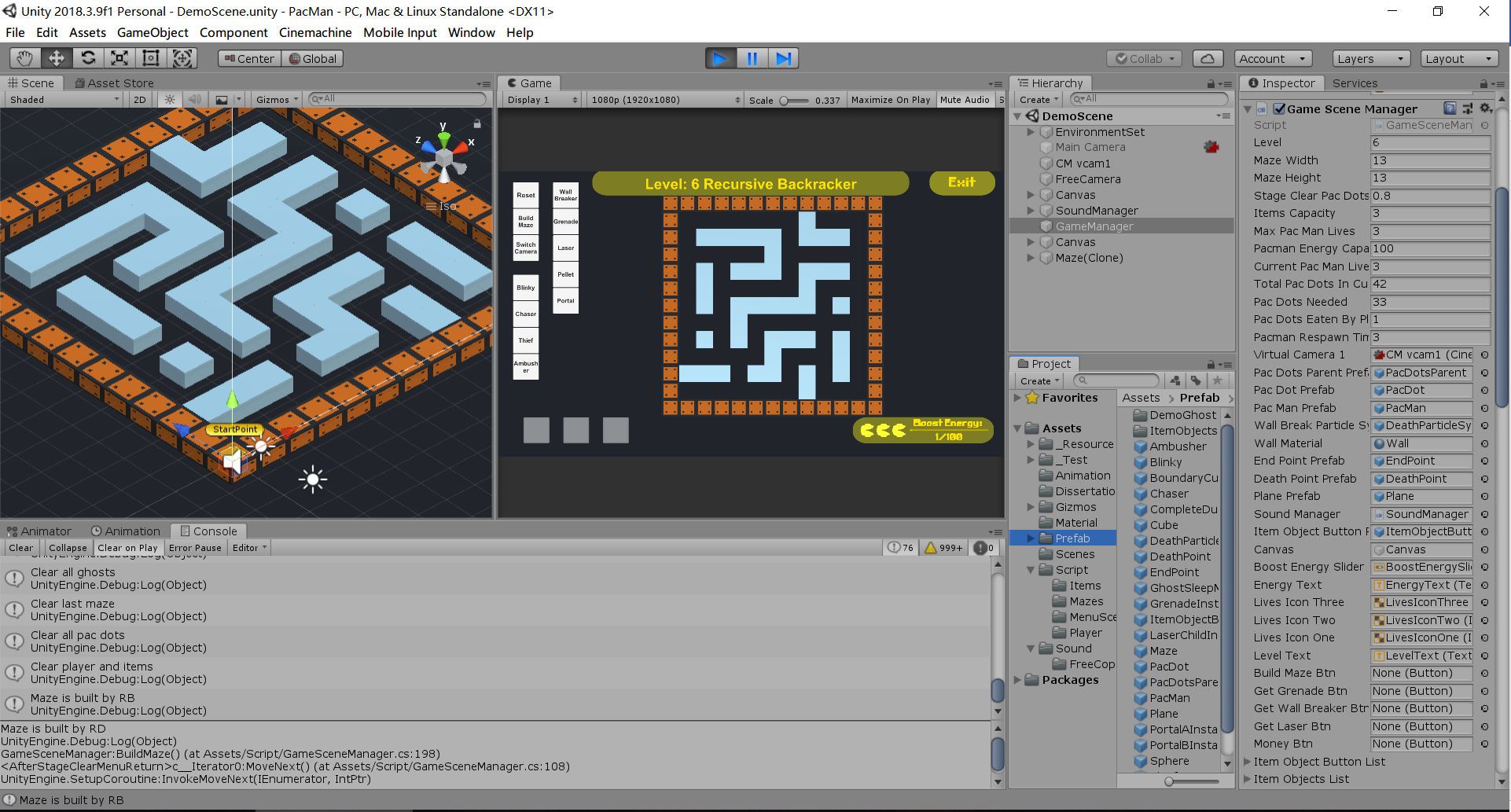
*Figure 6.8.2 Particle system*

1. **Evaluation**

In evaluation stage will be divided into three parts: test for each component, performance tests and player’s feedback. Test for each component will be completed inside Unity Game Engine. Unity performance testing will be done by recording the performance state of the final product. Players’ feedback will be collected from players of this game.

* 1. **Test for each component**

A special game scene called test scene will be created inside the Unity Game Engine. This scene will create a special game environment for each component of the game. All game objects in the test scene use the same application interface as the real game so they will have the same behaviour. Each game component including maze generation, items and ghosts will be tested in the scene separately. Figure 5.7.1 shows the interface of test scene.



*Figure 7.1.1 Test Scene*

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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test ID** | **Description of test** | **Test data** | **Expected Result** | **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 | Pass |
| 2 | Load of “Tutorial” menu: when loadMenu(“Tutorial”) is called, the “Tutorial” menu should appear |  | Tutorial interface should appear on the screen | Pass |
| 4 | A stage of game is ready to start: when loadScene(“game”) is called, load the game environment and ready for a game |  | All necessary game components for the game is ready for play | Pass |
| 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 | Pass |
| 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. | Pass |
| 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 | Pass |
| 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 | Pass |
| 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 | Pass |
| 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 | Pass |
| 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 | Pass |
| 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 | Pass |
| 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. | Pass |
| 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 | Pass |
| 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 | Pass |
| 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 | Pass |
| 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 | Pass |
| 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. | Pass |
| 19 | Test that a grenade can be used to throw over a wall and destroy 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 destroyed. | Pass |
| 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. | Pass |
| 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 | Pass |
| 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. | Pass |
| 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. | Pass |
| 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. | Pass |
| 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. | Pass |
| 26 | Test that if the “Exit” button works. | Player uses “Exit” button | The software is terminated. | Pass |

* 1. **Performance test**

Performance test will be completed by recording the average, maximum and minimum frame rate during the running of this game.

* A real-time video benchmarking software “Fraps” will be used to record the framerate of the final product. Fraps can perform benchmarks and record the frame rate.  The statistics can also be saved and used for reviews and applications [15].
* To pass this test, in a computer with 2.6 GHz CPU frequency, 8G 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.

5 performance tests have been done and the following table shows the result for the performance testing. The performance test file is also attached in the appendices.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 2019-04-16 21:49:13 - PacMan | | | | |
| Total Frames: 2990 | Time: 50031ms | Avg: 59.763 | Min: 55 | Max: 62 |
| 2019-04-16 21:51:59 - PacMan | | | | |
| Total Frames: 4081 | Time: 69172ms | Avg: 58.998 | Min: 35 | Max: 61 |
| 2019-04-16 21:53:50 - PacMan | | | | |
| Total Frames: 2091 | Time: 34844ms | Avg: 60.010 | Min: 59 | Max: 61 |
| 2019-04-16 21:55:13 - PacMan | | | | |
| Total Frames: 7961 | Time: 140344ms | Avg: 56.725 | Min: 36 | Max: 61 |
| 2019-04-16 21:57:54 - PacMan | | | | |
| Total Frames: 8733 | Time: 150015ms | Avg: 58.214 | Min: 34 | Max: 62 |

For these 5 tests, all average framerate results are more than 30 and the minimum is 35. Therefore, the final product passes the performance test.

* 1. **Players’ feedback**

A questionnaire will be designed to collect players’ opinions 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 least favourite?

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

1. Do you have any extra comments and suggestions?

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

1. **Learning Points**

In this project, a Pac-Man game is designed, realised and evaluated. During this process, several professional skills such as object-oriented design and Unity game engine have been learned. This section will discuss those skills that have been learned and practised.

* **Object-oriented design**

Object-oriented design is the discipline that solve a certain problem by defining different objects and interactions between them [8]. Because game is a software that includes many game objects and interactions, the OO design is used as the main design methodology in this project.

Object-oriented design methodology is learned and practised in this project. To use object-oriented design in this game project, firstly all game components and objects such as ‘GameManager’ and ‘Ghost’ are listed. Then, relations between each class are defined. Finally, attributes and interfaces are added to each class. A class diagram that indicates the structure of this game is now can be created. With the completed class diagram, details for each class can be completed one by one.

The advantage of object-oriented design is that it divides a large system into several linked components. The whole system can be completed by defining each components and interactions, which makes the design process easy and clear.

* **Unity game engine**

Another important learning point is the using of unity game engine. Unity is the most important tool in the project so the learning of Unity engine starts from the beginning of this project. Firstly, some Unity official tutorials have been practised. These tutorials provide basic knowledge for users to use the game engine. For example, using colliders to detect interactions between objects and using C# script to implement the logic for different objects. Some tutorials are practised before the design stage so that I can avoid some problems from design stage and provide a more all-round design.

Unity skills are also learned in implement stage. The Unity Scripting API and the discussion board in Unity community are powerful tools when any problem about Unity is met. In addition, some powerful functions such as Coroutine [16] and animation system are learned during the realisation stage to provide better gameplay experience.

* **Project management**

In terms of the project management, some software engineering skills have been learned in this project. Firstly, a good design document is important for a project. The design document often records key information about the project and key methods that might be useful in the final product. Without an explicit design document, developers may become confusing about what they should do and which kind of product they should achieve. If there is a good design and the development can follow the requirements of design, the final software can be built faster and easier.

Another learning point about project management is the using of GitHub. The development of such project needs many iterations, which is difficult for backup and code management. GitHub provides a powerful platform for developers to solve this problem. Each state of project can be recorded and fetched conveniently using Git. In addition, GitHub provides online repository, which is also good for safety of codes. In summary, using of GitHub can help developers to save much time in management and backup of their software.

1. **Professional Issues**

In this project, several rules and professional standards are satisfied to meet the requirement of British Computer Society (known as BCS). This section will discuss how this project is related to the codes of practice and conduct issued by the BCS.

* **Project management**

The BCS encourages developers to follow some basic rules in each stage of a project. This part will discuss how this project follows the “Project Management” section of BSC’s “Code of Good Practice”.

In the design stage of this project, a design document that covers all aspects of this project is provided. This design document ensures that the scope, deliverables, responsibilities and time scales are agreed in advance. This design also helps me to assure that resources, cost and timescale is enough for completing of the project.

During the implement of the project, each component in the project is realised according to the design. The time cost of each part of the project is monitored. No assumption that any overruns can be recovered later in the project are made so no later activities are cut back in this project.

When closing a project, several designed evaluations are carried out to judge this project. Three different kinds of evaluation: components’ test, performance test and users’ feedback are completed to make sure that the final product achieve the ideal objectives satisfies the requirements. In addition, strengths and weakness of the final product is also summarized, which will be useful for next project.

* **Public Interest**

This project also follows the BCS’s rules and professional standards about public interest.

In terms of health, this game is a pixel Pac-Man game. This game provides lovely visual style and game environment. There is no any displeased element such as blood and extreme violence in this game. Therefore, this game is expected to help players to relax and enjoy.

In terms of privacy, this project does not need any human data in the design and implement stage. In the final product, this game software will not collect and store any information about the players. In addition, in the evaluation stage, all evaluators are volunteered participants and they are informed that they are involved in this project. All feedback submitted by them are anonymous and they are only asked to provide necessary information.

In addition, this game does not contain any discrimination on the grounds such as sex and nationality and all people are welcomed to play this game.

* **Duty to the Relevant Authority**

Many resources are used in this project. These resources include self-made resources and others’ resources. Therefore, the duty to those relevant authority is paid much attention in this project.

In this project, most of resources, including models, scripts, materials and textures are made by myself. Part of resources are gotten form the Internet and the following table shows source of these resources. All of those resources come from a legal way and situation that may give rise to a conflict of interest is avoided.

|  |  |
| --- | --- |
| ***Resource*** | ***Source of the resource*** |
| Sound effects of all items | Unity official standard assert package |
| Part of textures and materials of particle system | Unity official standard assert package |
| Font of user interface text | Unity official standard assert package |
| Part of materials used on Portal item | Free “Gem” unity assert store package |
| Images of “Item Capacity” and “Energy Capacity” in shop interface | Unity official standard assert package |

In addition, this game also contains several background music. All of those music come from Jamendo Music [7]. This website provides royalty free music so these music can be used freely in this non-commercial project. Titles of music and the name of artist will also be shown on the bottom-left of the screen. Players can click the link on the main menu to check details about copyright of these music.

In summary, rules and professional standards set by British Computer Society direct the behaviour of developers and all of these rules will be observed in future projects.

1. **Bibliography**

[1] MagicaVoxel. (2018, 1 Nov). *MagicaVoxel*. Available: [https://ephtracy.github.io/index.html?page=mv\_main#](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] Jamendo. (2019, 20 April). *Jamendo Music*. Available: <https://www.jamendo.com/start>

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

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

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

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

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

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

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

[15] Beppa. (2019, 25 April). *Fraps Real-time video capture & benchmarking*. Available: <http://www.fraps.com/>

[16] Unity. (2019, 1st May). *Coroutines*. Available: <https://docs.unity3d.com/Manual/Coroutines.html>