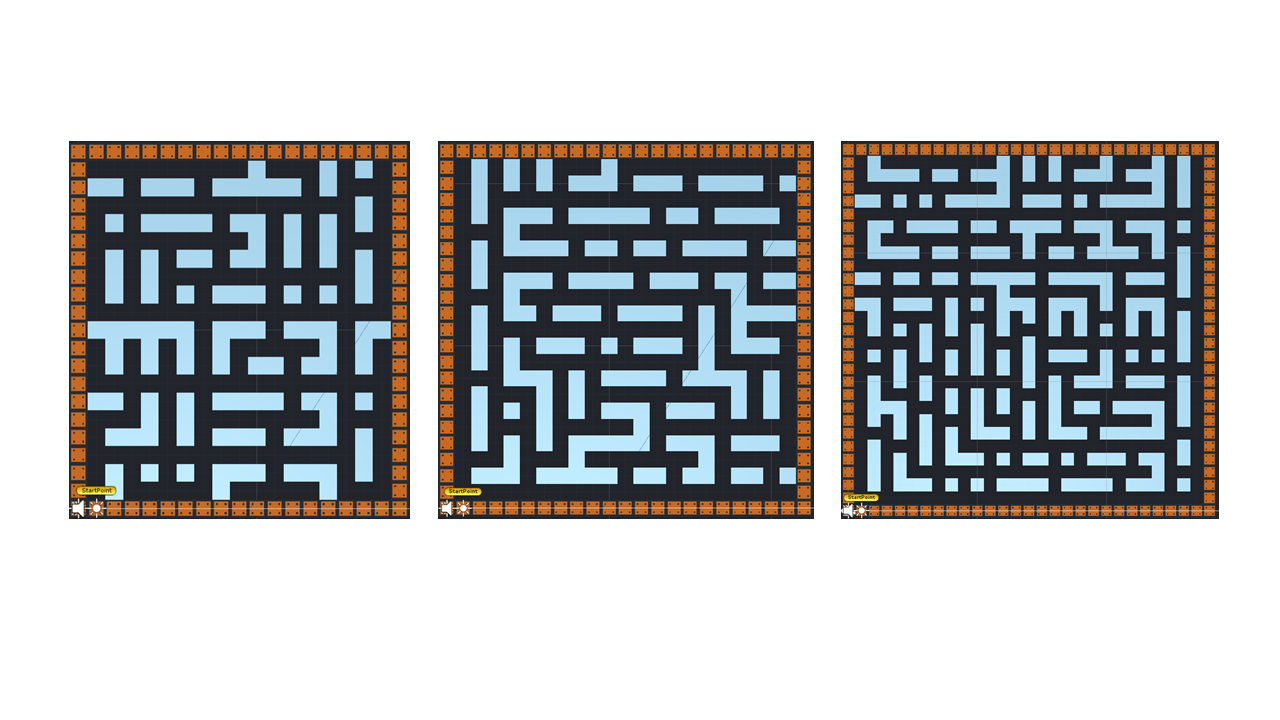
1. **Realisation**

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.

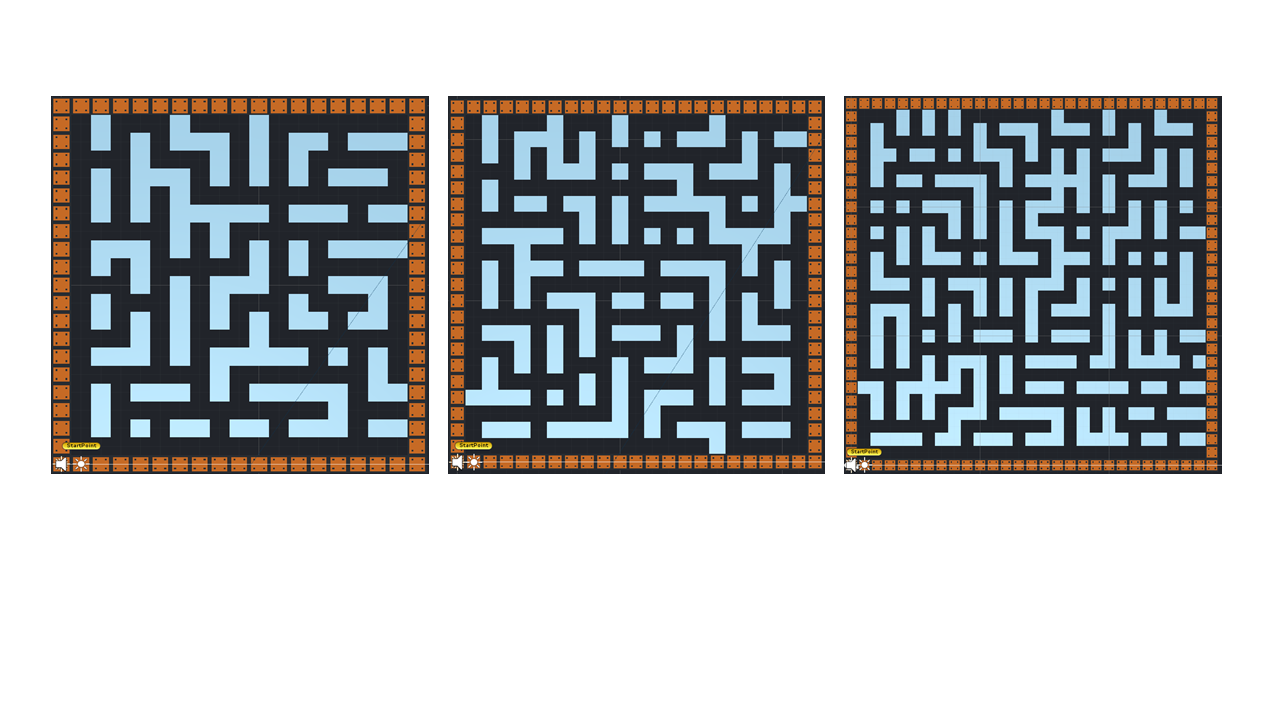
* 1. **Maze Generation**

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.



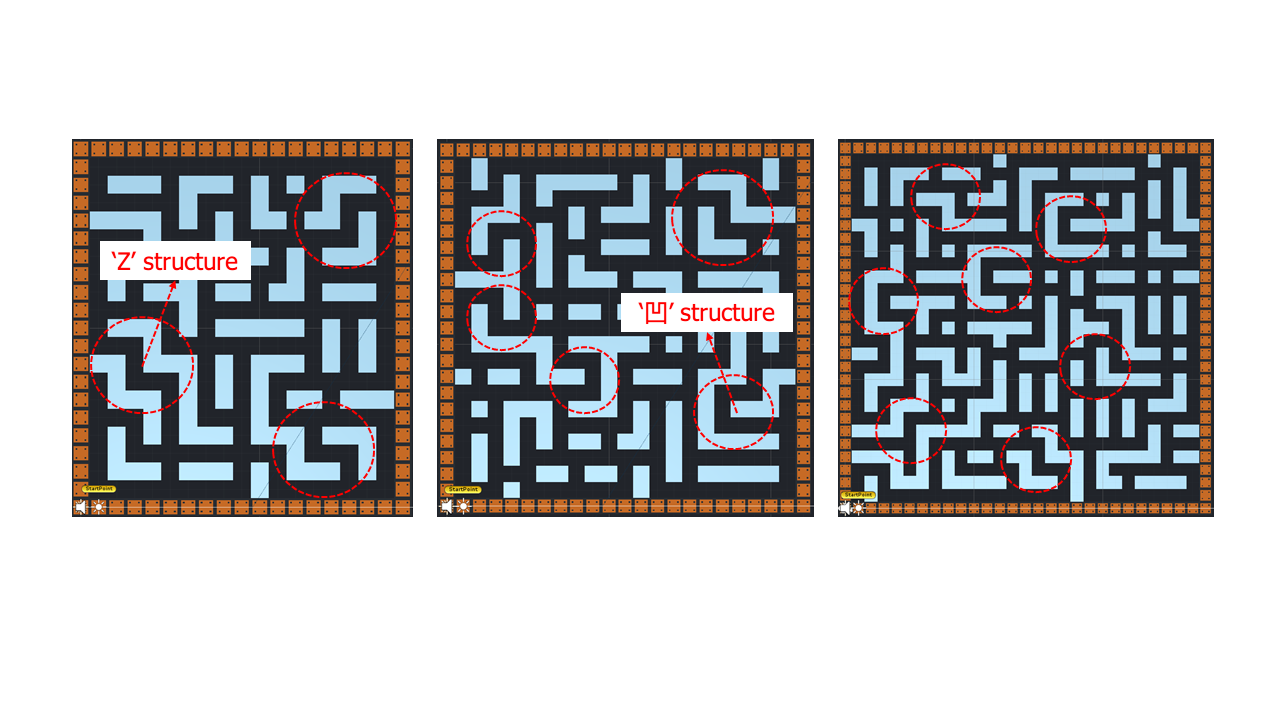
*Figure 6.1.1 Mazes generated by recursive division*

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.

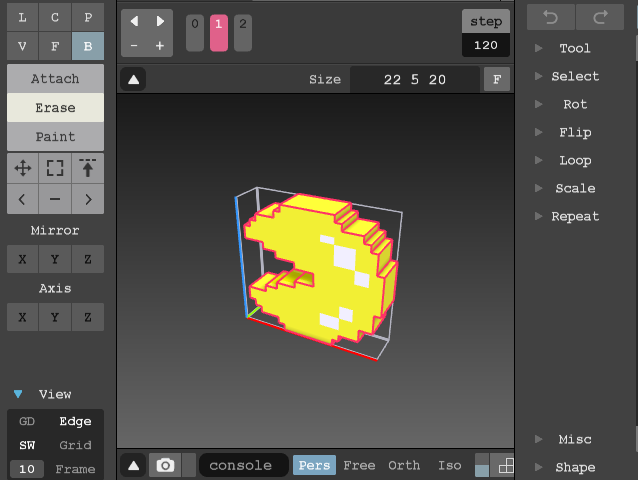


*Figure 6.1.3 Mazes generated by recursive backtracker algorithm*

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.

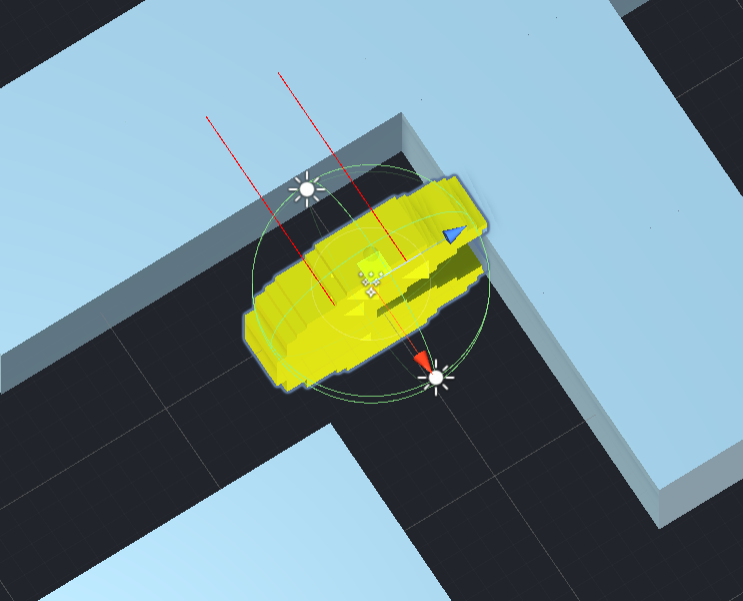
* 1. **Player**

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*

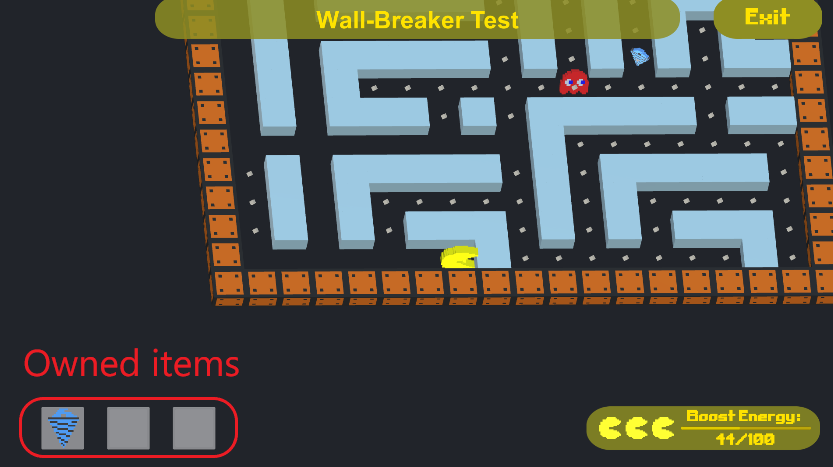
Figure 6.2.1 shows the making of PacMan’s 3D model in MagicaVoxel. MagicaVoxel is a modelling tool that provides modelling, rendering, painting and animation functions. All models used in the game are created using MagicaVoxel.



*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.

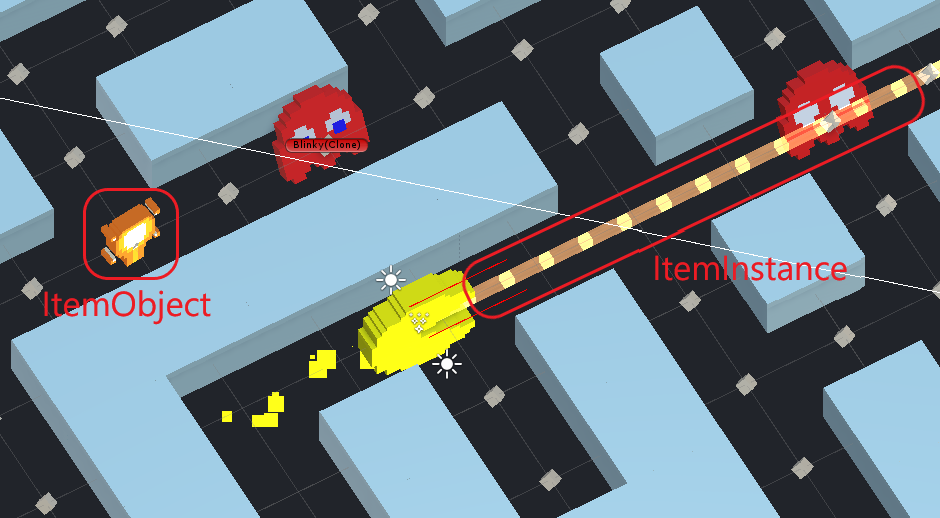


*Figure 6.2.4 Boost and boost energy*

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.



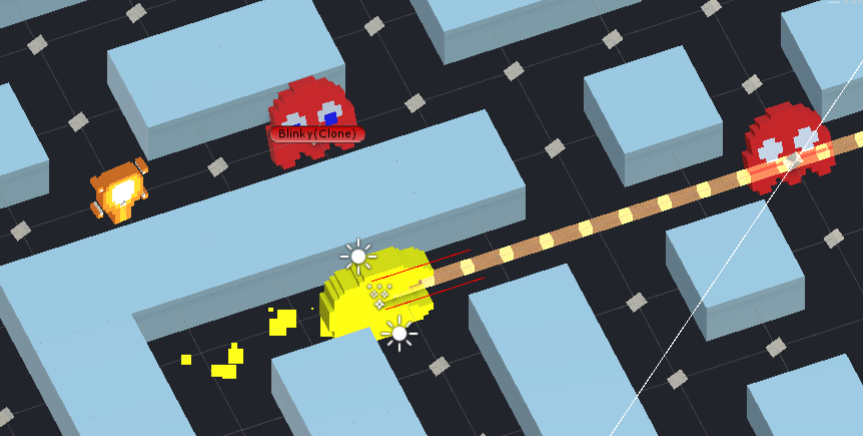
*Figure 6.3.1 ItemObject and ItemInstance*

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.



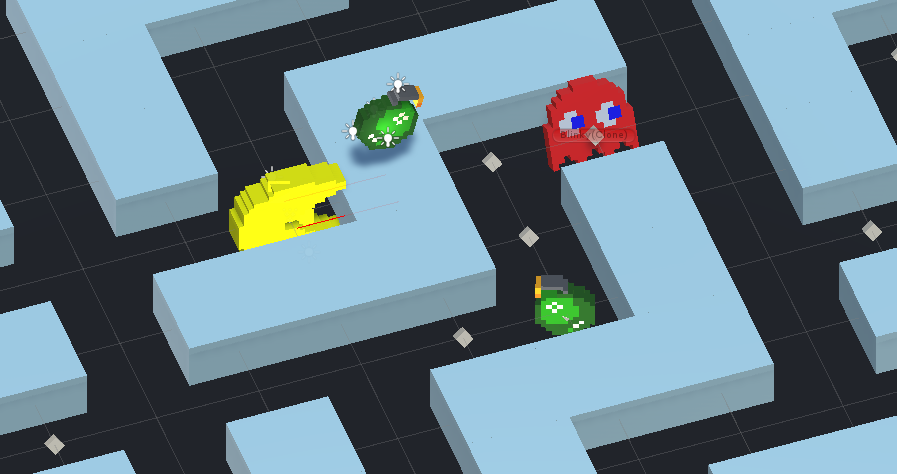
*Figure 6.3.2 Laser*

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.



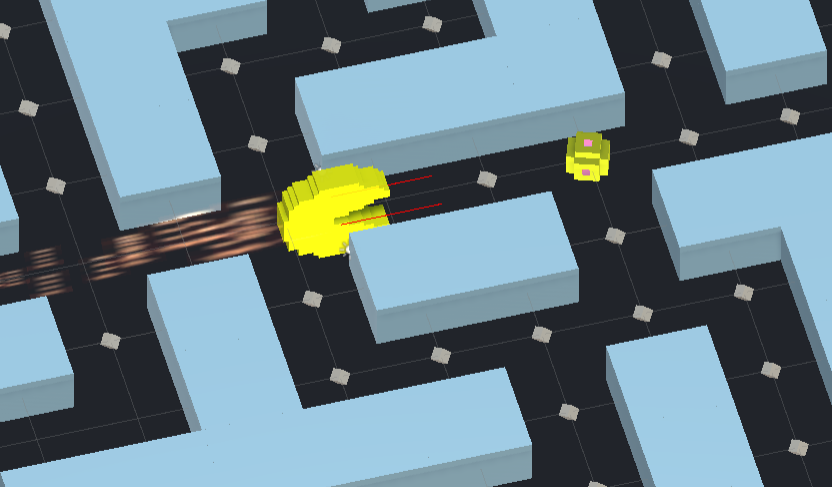
*Figure 6.3.3 Wall-Breaker*

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.

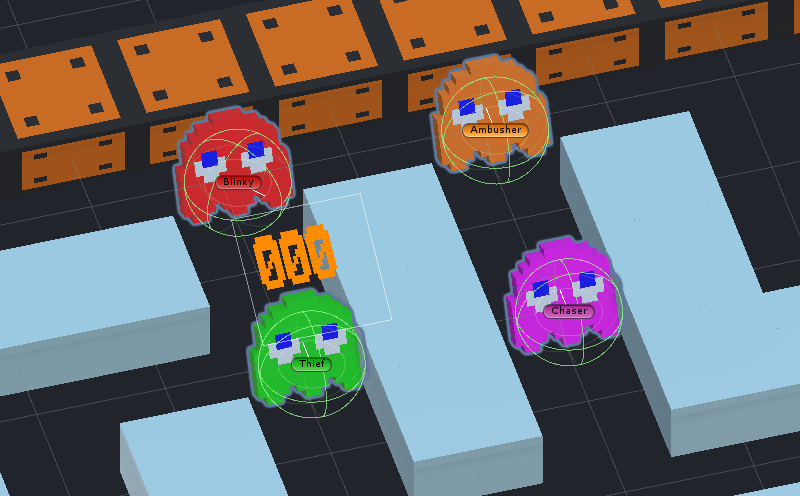


*Figure 6.3.5 Energy Pellet*

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.

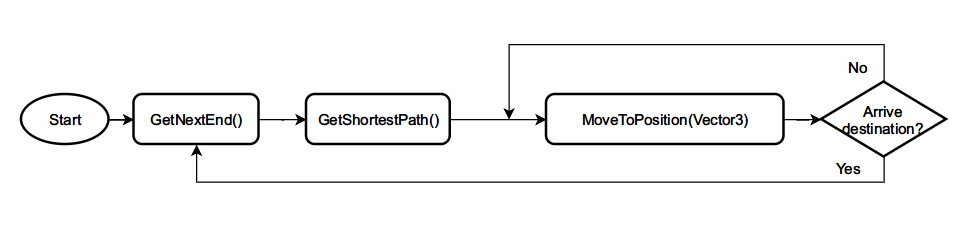
* **Interactions between Ghost and other GameObject**

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.

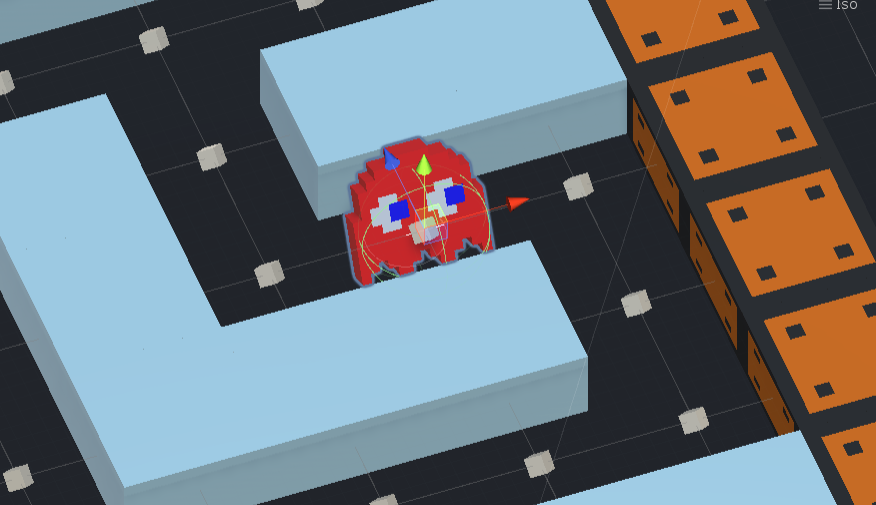
* **Movement of Ghost**



*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.

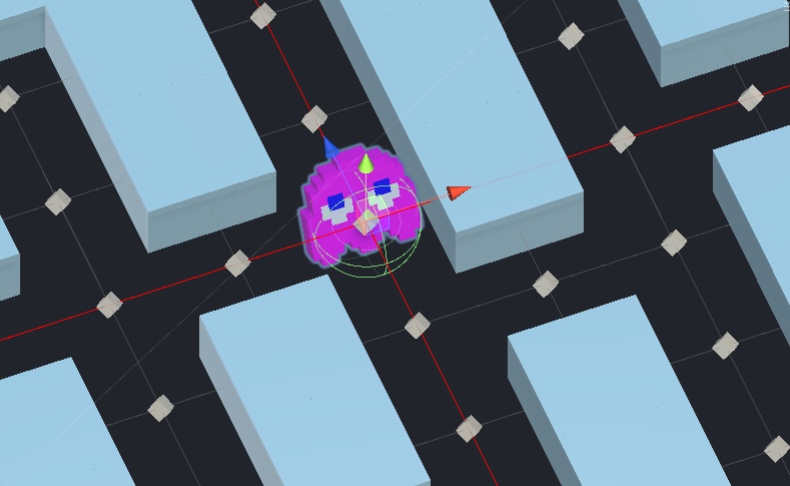
* **Implement of Blinky**



*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**

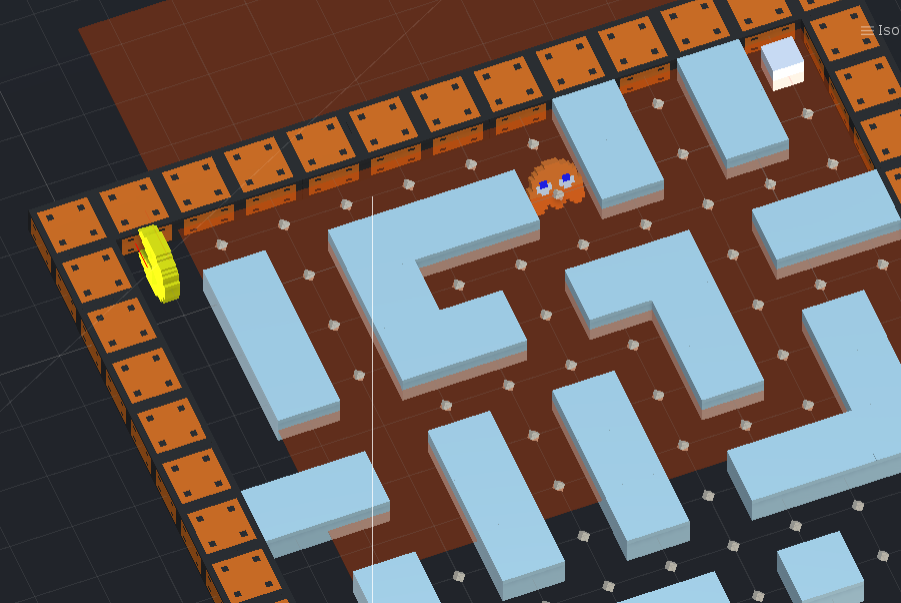


*Figure 6.4.4 Chaser*

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.

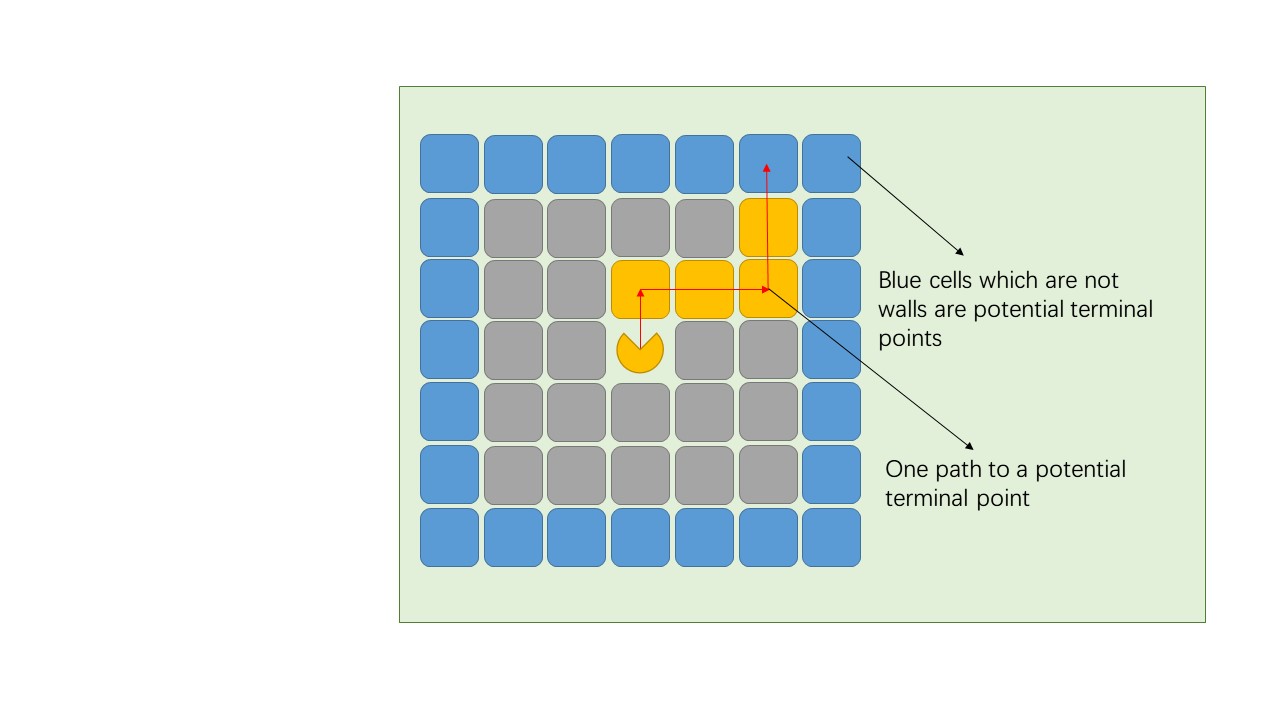
* **Implement of Ambusher**



*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**



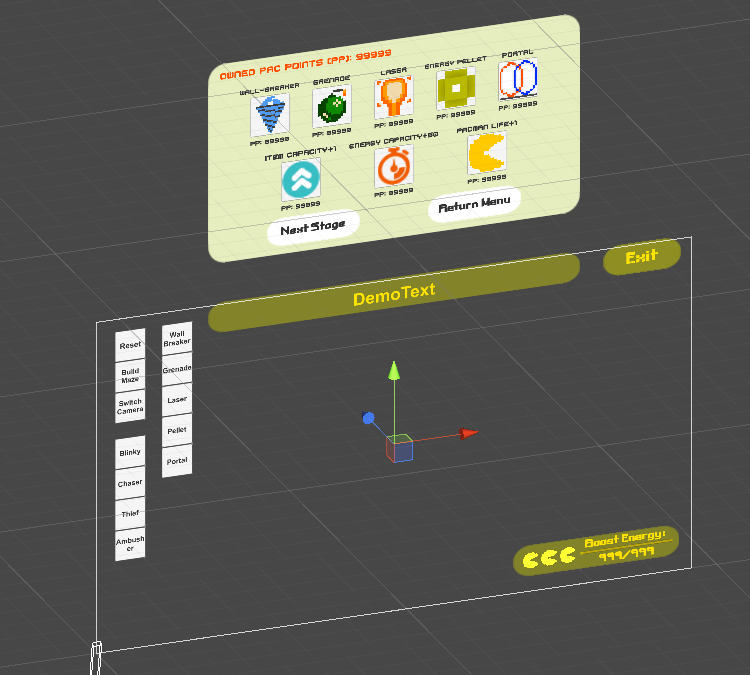
*Figure 6.4.7 Thief*

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**

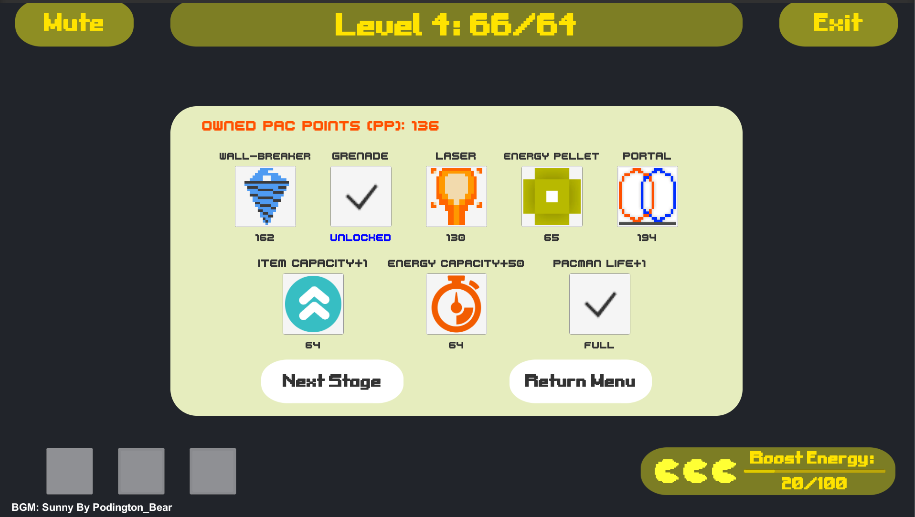
In the implement stage, user interface is created using Unity canvas system. Firstly, a canvas GameObject will be created as a container of UI elements. Then UI components such as buttons will be added to the canvas. Figure 6.5.1 shows an example of layout of canvas and UI components. In this example, 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*

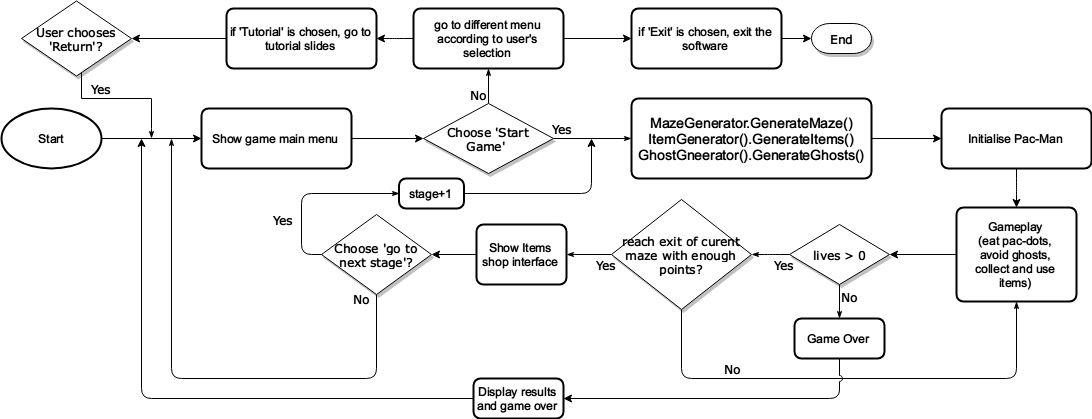
* 1. **ItemGenerator and GhostGenerator**

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

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