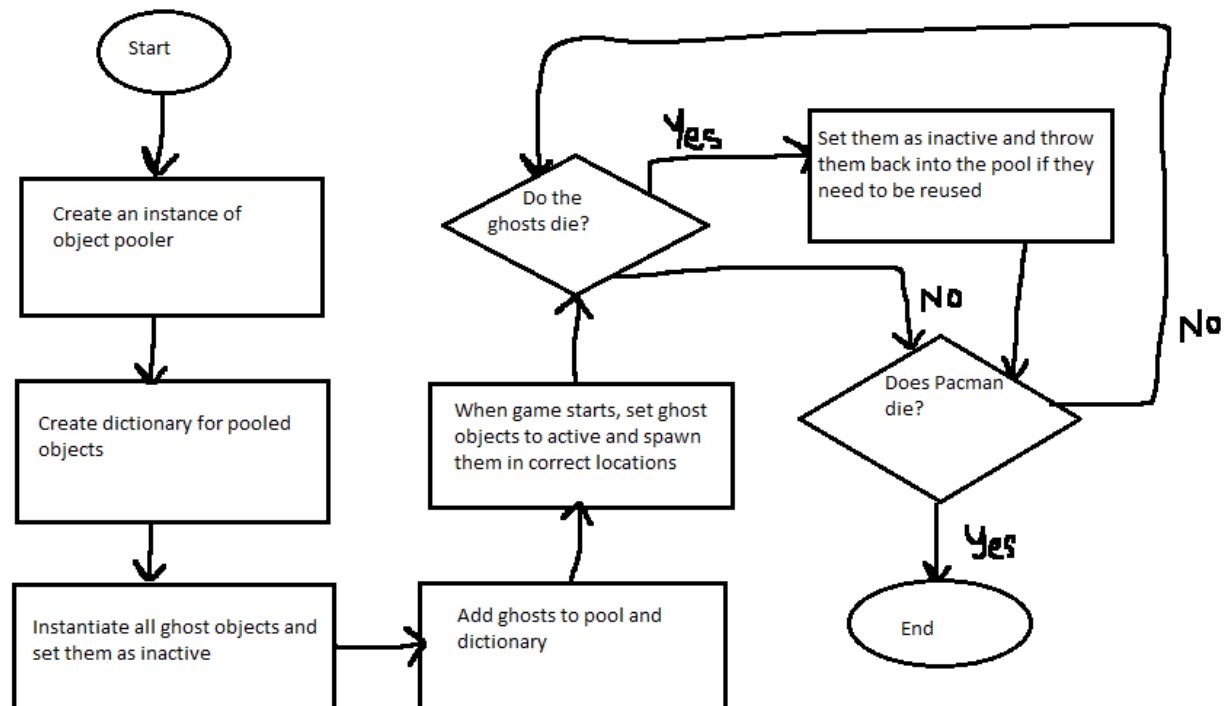


Explanations – Odd Number

Object Pooling



I didn't implement it, but here's an explanation for how it would work. The reason why object pooling works for Pacman is that you can reuse ghosts if you need to respawn them. If they are eaten by Pacman, you can set them as inactive and put them back into the object pool. Then if you ever need to reuse them, you can respawn them by setting them as active again and resetting their positions. The second added benefit of having object pooling is so you can instantiate everything at the beginning instead of in the middle of the game. If you have a lot more ghosts than 4 for whatever reason, and each of them are extremely complex in modelling, requiring a lot of resources to be used up upon instantiation, you can offset all the load time to the beginning instead of during the game. This prevents the possibility of lag spikes during the game.

Command Design Pattern

I didn't implement this nor explained it, but I'll mention why it's useful. Being able to undo the last 7 eaten pellets could be a game mechanic that adds challenge to Pacman by undoing progress. It could also be a game mechanic that helps players in case they realise they made a turn in the wrong direction and are going to be cornered by the ghosts, so they can backtrack and try again.

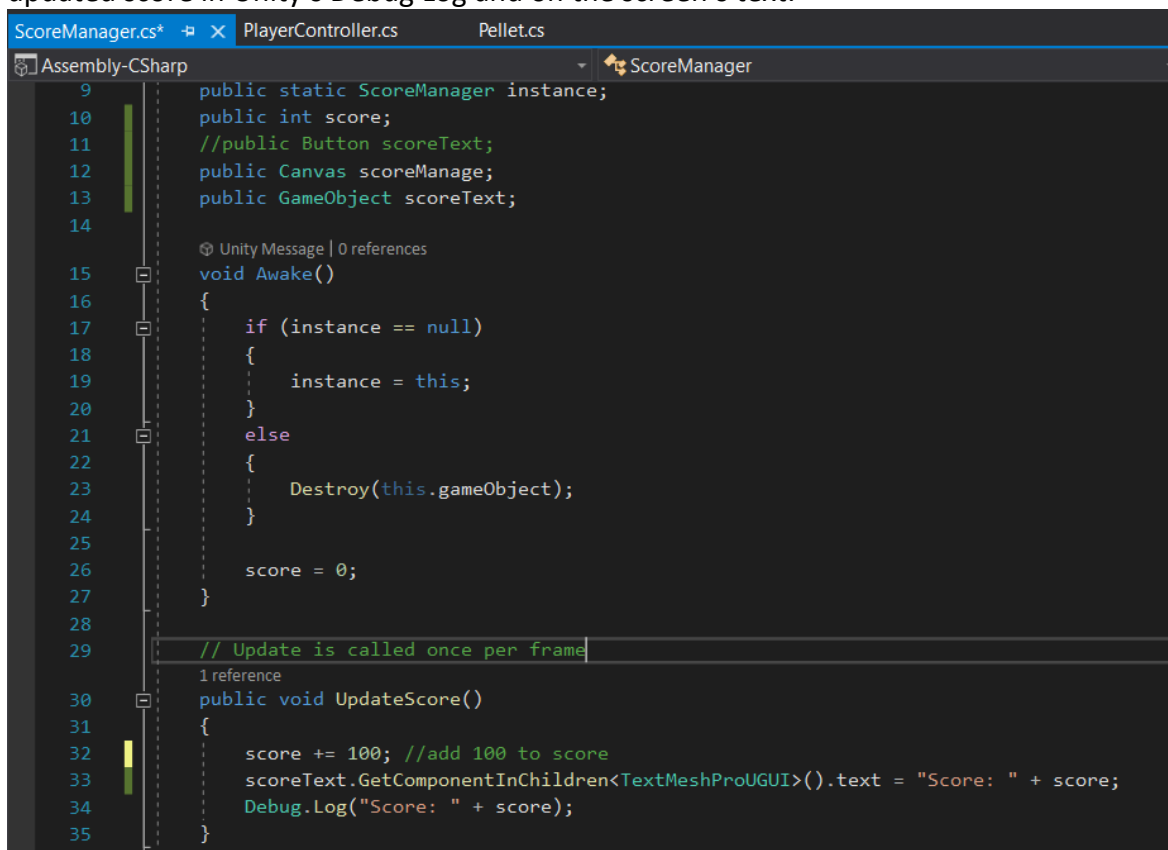
next page is management system

Question 7

Chose option 1 and created a score management system

I chose a score management system since the goal of Pacman is to get as many points as possible. It's good to have a score manager implemented as a singleton because it allows for centralised management, so if you have multiple items that grant different score values, you can assign different scripts and functions to have the score update differently. It's helpful on the programmer side by making the development process easier. If you updated scores individually for each object, you'd need to access the Canvas each time to update the number displayed and find a way to access the current score. Having a singleton management system cleans up the code and allows for easy access to global variables.

To implement this, I have a ScoreManager script that is created as an instance. It has an UpdateScore function that adds 100 to the player's score every time it is called. It displays the updated score in Unity's Debug Log and on the screen's text.



```
ScoreManager.cs* x PlayerController.cs Pellet.cs
Assembly-CSharp
ScoreManager
9 public static ScoreManager instance;
10 public int score;
11 //public Button scoreText;
12 public Canvas scoreManage;
13 public GameObject scoreText;
14
15 Unity Message | 0 references
16 void Awake()
17 {
18     if (instance == null)
19     {
20         instance = this;
21     }
22     else
23     {
24         Destroy(this.gameObject);
25     }
26     score = 0;
27 }
28
29 // Update is called once per frame
30 1 reference
31 public void UpdateScore()
32 {
33     score += 100; //add 100 to score
34     scoreText.GetComponentInChildren<TextMeshProUGUI>().text = "Score: " + score;
35     Debug.Log("Score: " + score);
36 }
```

Then, I attach a Pellet script to the pellets that Pacman must pick up. In the script, I check if Pacman collides with the pellet. If he does, you delete the object. You also access the instance of the Score Manager and call the UpdateScore function which adds to the score. In a future application, I could have a script for the ghosts, and if a collision occurs between them and Pacman when they're blue, I can call a different function in the score manager that will update with a different score value added than 100.

```
PlayerController.cs  Pellet.cs
CSharp
Unity Message | 0 references
private void OnCollisionEnter(Collision other)
{
    if (other.collider.tag == "Player")
    {
        Destroy(this.gameObject);
        ScoreManager.instance.UpdateScore();
    }
}
```