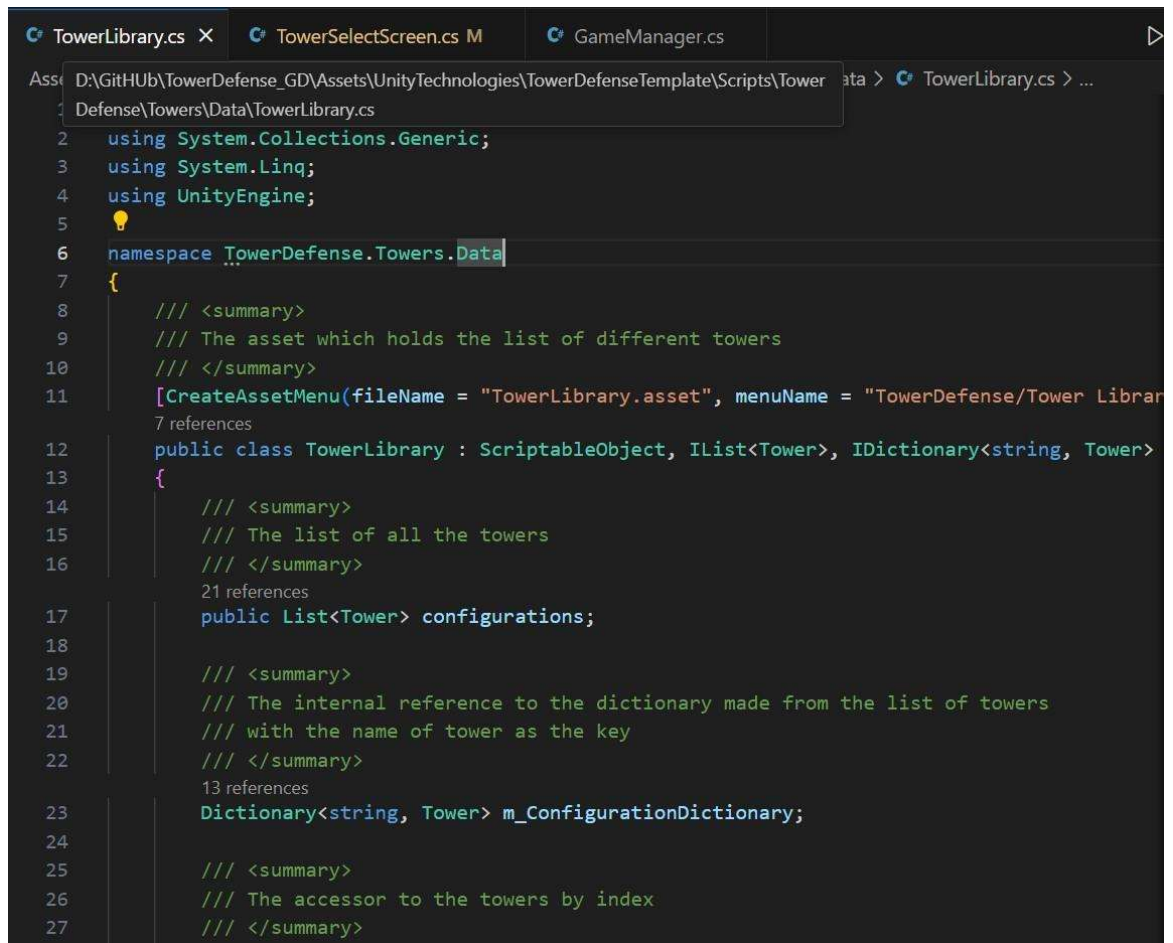


Chapter 3

Some Important Code Screenshots

3.1 TowerLibrary.cs

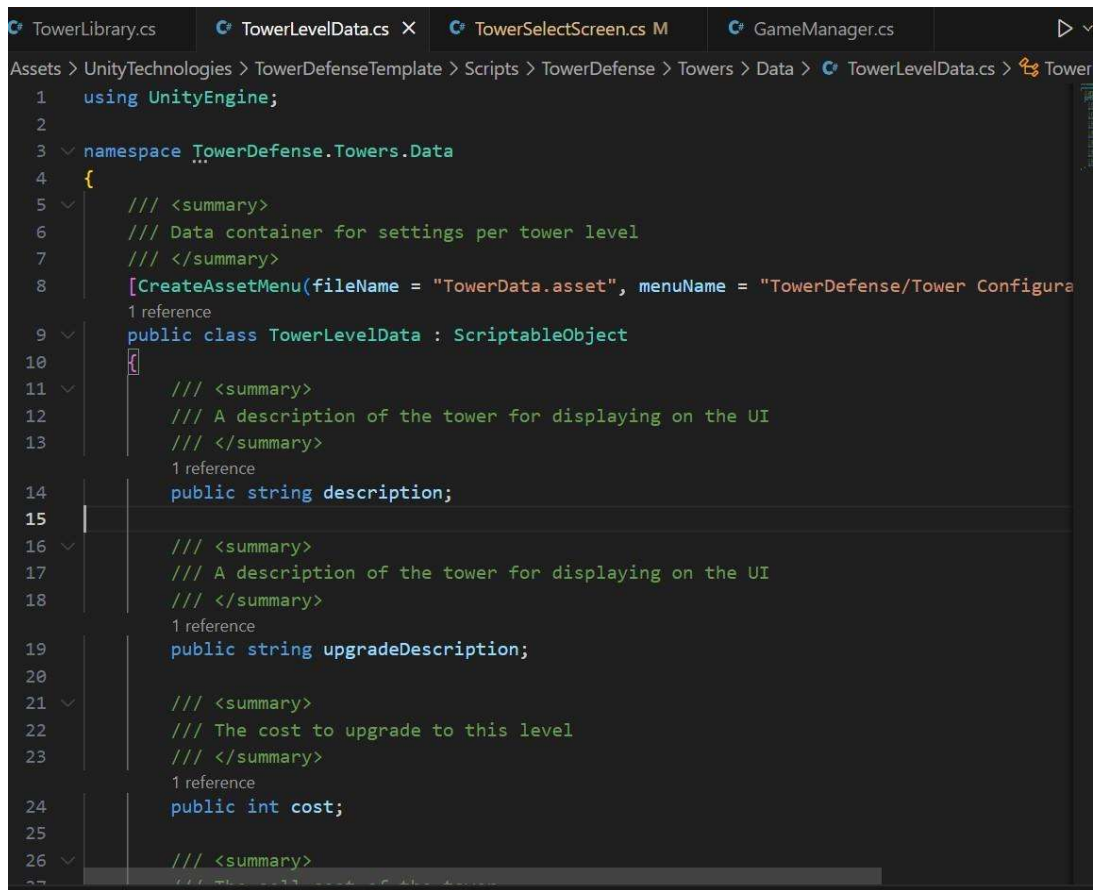
The screenshot shows a code editor with three tabs: TowerLibrary.cs, TowerSelectScreen.cs, and GameManager.cs. The TowerLibrary.cs tab is active, showing the following code:

```
1 using System.Collections.Generic;
2 using System.Linq;
3 using UnityEngine;
4
5
6 namespace TowerDefense.Towers.Data
7 {
8     /// <summary>
9     /// The asset which holds the list of different towers
10    /// </summary>
11    [CreateAssetMenu(fileName = "TowerLibrary.asset", menuName = "TowerDefense/Tower Library")]
12    public class TowerLibrary : ScriptableObject, IList<Tower>, IDictionary<string, Tower>
13    {
14        /// <summary>
15        /// The list of all the towers
16        /// </summary>
17        public List<Tower> configurations;
18
19        /// <summary>
20        /// The internal reference to the dictionary made from the list of towers
21        /// with the name of tower as the key
22        /// </summary>
23        Dictionary<string, Tower> m_ConfigurationDictionary;
24
25        /// <summary>
26        /// The accessor to the towers by index
27        /// </summary>
```

Figure 3.1: TowerLibrary Script

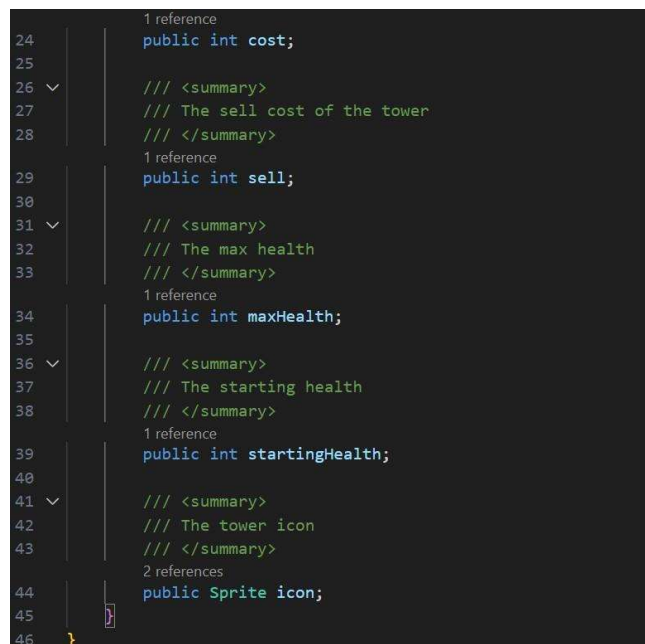
The Script in Figure 3.1 is a ScriptableObject that manages a collection of Tower objects, providing both list-based and dictionary-based access. It implements IList and IDictionary to enable retrieval by index and by tower name.

3.2 TowerLevelData.cs



```
1 using UnityEngine;
2
3 namespace TowerDefense.Towers.Data
4 {
5     /// <summary>
6     /// Data container for settings per tower level
7     /// </summary>
8     [CreateAssetMenu(fileName = "TowerData.asset", menuName = "TowerDefense/Tower Configura
9     1 reference
10    public class TowerLevelData : ScriptableObject
11    {
12        /// <summary>
13        /// A description of the tower for displaying on the UI
14        /// </summary>
15        1 reference
16        public string description;
17
18        /// <summary>
19        /// A description of the tower for displaying on the UI
20        /// </summary>
21        1 reference
22        public string upgradeDescription;
23
24        /// <summary>
25        /// The cost to upgrade to this level
26        /// </summary>
27        1 reference
28        public int cost;
29
30        /// <summary>
31        /// The sell cost of the tower
32        /// </summary>
33        1 reference
34        public int sell;
35
36        /// <summary>
37        /// The max health
38        /// </summary>
39        1 reference
40        public int maxHealth;
41
42        /// <summary>
43        /// The starting health
44        /// </summary>
45        1 reference
46        public int startingHealth;
47
48        /// <summary>
49        /// The tower icon
50        /// </summary>
51        2 references
52        public Sprite icon;
53    }
54 }
```

Figure 3.2: TowerLevelData Script Part 1



```
24    1 reference
25    public int cost;
26
27    /// <summary>
28    /// The sell cost of the tower
29    /// </summary>
30    1 reference
31    public int sell;
32
33    /// <summary>
34    /// The max health
35    /// </summary>
36    1 reference
37    public int maxHealth;
38
39    /// <summary>
40    /// The starting health
41    /// </summary>
42    1 reference
43    public int startingHealth;
44
45    /// <summary>
46    /// The tower icon
47    /// </summary>
48    2 references
49    public Sprite icon;
50
51    }
52 }
```

Figure 3.3: TowerLevelData Script Part 2

The script in figure [3.2,3.3] is a 'ScriptableObject' that stores settings for each tower level, including descriptions, cost, health, and an icon for UI display. It helps manage tower upgrades and attributes in a Tower Defense game.

3.3 GameManager.cs

```

15 public class GameManager : GameManagerBase<GameManager, GameDataStore>
27     protected override void Awake()
28     {
29         Screen.sleepTimeout = SleepTimeout.NeverSleep;
30         base.Awake();
31
32         int i;
33         for (i = 0; i < towerlist.Count; i++){
34             Debug.Log("is Unlocked "+ IsTowerUnlocked(i));
35         }
36
37         //Ensure first 4 towers are always unlocked
38         for (i = 0; i < 4; i++){
39             if (!IsTowerUnlocked(i)){
40                 UnlockTower(i);
41                 SelectTower(i);
42             }
43         }
44
45         if (LevelManager.instance){
46             LevelManager.instance.towerLibrary.Clear();
47
48             for (i = 0; i < towerlist.Count; i++){
49                 if (IsTowerUnlocked(i) && IsTowerSelected(i)){
50                     selectedTowers.Add(towerlist[i]);
51                     Debug.Log($"{i} Added");
52                 }
53             }
54             Debug.Log("Tower Updated");
55         }

```

Figure 3.4: GameManager Script Part 1

```

58     /// <summary>
59     /// Method used for completing the level
60     /// </summary>
61     /// <param name="levelId">The levelId to mark as complete</param>
62     /// <param name="starsEarned"></param>
63     1 reference
64     public void CompleteLevel(string levelId, int starsEarned)
65     {
66         if (!levelList.ContainsKey(levelId))
67         {
68             Debug.LogWarningFormat("[GAME] Cannot complete level with id = {0}. Not in
69             return;
70         }
71
72         m_DataStore.CompleteLevel(levelId, starsEarned);
73         SaveData();
74     }

```

Figure 3.5: GameManager Script Part 2

```

76  ✓    /// <summary>
77  //    /// Method used for unlocking the tower
78  //    /// </summary>
      2 references
79  ✓    public void UnlockTower(int ind)
80  {
81      m_DataStore.UnlockTower(ind);
82      SaveData();
83  }
84
85  ✓    /// <summary>
86  //    /// Method used for selecting the tower
87  //    /// </summary>
      3 references
88  ✓    public void SelectTower(int ind)
89  {
90      m_DataStore.SelectTower(ind);
91      SaveData();
92  }
93
      1 reference
94  ✓    public void DeSelectAllTowers(){
95      m_DataStore.DeSelectAllTowers();
96      SaveData();
97  }

```

Figure 3.6: GameManager Script Part 3

```

99      /// <summary>
100     /// Gets the id for the current level
101     /// </summary>
      4 references
102     public LevelItem GetLevelForCurrentScene()
103     {
104         string sceneName = SceneManager.GetActiveScene().name;
105
106         return levellist.GetLevelByScene(sceneName);
107     }
108
109     /// <summary>
110     /// Determines if a specific level is completed
111     /// </summary>
112     /// <param name="levelId">The level ID to check</param>
113     /// <returns>true if the level is completed</returns>
      0 references
114     public bool IsLevelCompleted(string levelId)
115     {
116         if (!levellist.ContainsKey(levelId))
117         {
118             Debug.LogWarningFormat("[GAME] Cannot check if level with id = {0} is comp
119             return false;
120         }
121
122         return m_DataStore.IsLevelCompleted(levelId);
123     }
124

```

Figure 3.7: GameManager Script Part 4

```

125 3 references
126 public bool IsTowerUnlocked(int ind){
127     return m_DataStore.IsTowerUnlocked(ind);
128 }

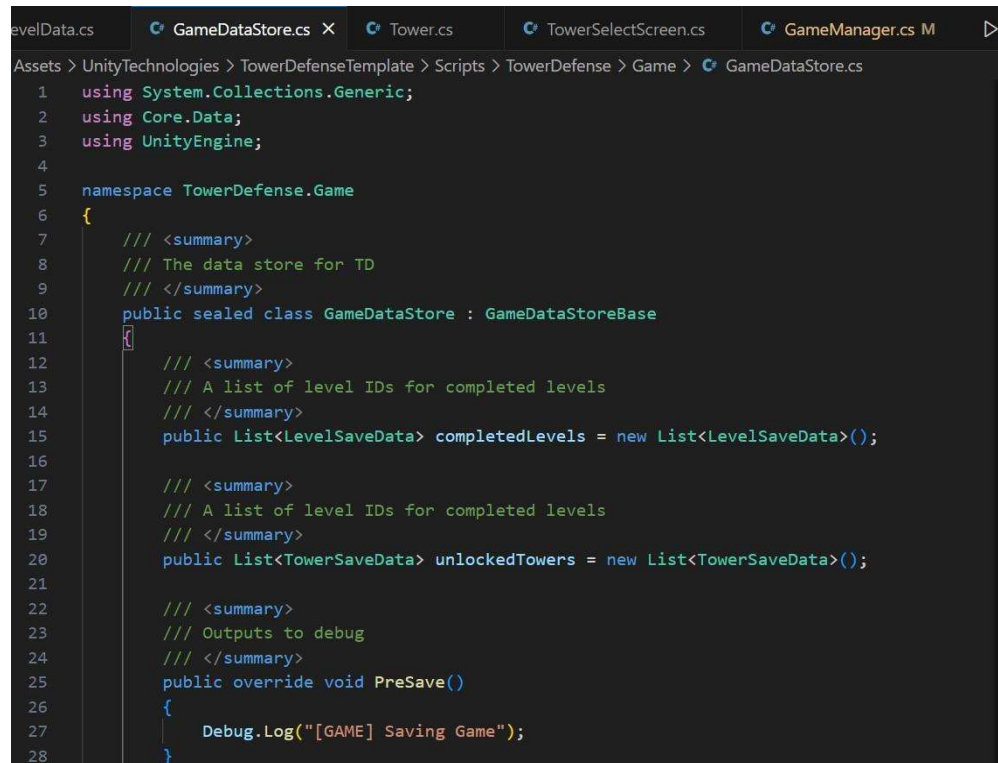
129 2 references
130 public bool IsTowerSelected(int ind){
131     return m_DataStore.IsTowerSelected(ind);
132 }
133 /// <summary>
134 /// Gets the stars earned on a given level
135 /// </summary>
136 /// <param name="levelId"></param>
137 /// <returns></returns>
138 2 references
139 public int GetStarsForLevel(string levelId)
140 {
141     if (!levelList.ContainsKey(levelId))
142     {
143         Debug.LogWarningFormat("[GAME] Cannot check if level with id = {0} is complete");
144         return 0;
145     }
146     return m_DataStore.GetNumberOfStarForLevel(levelId);
147 }
148 }

```

Figure 3.8: GameManager Script Part 5

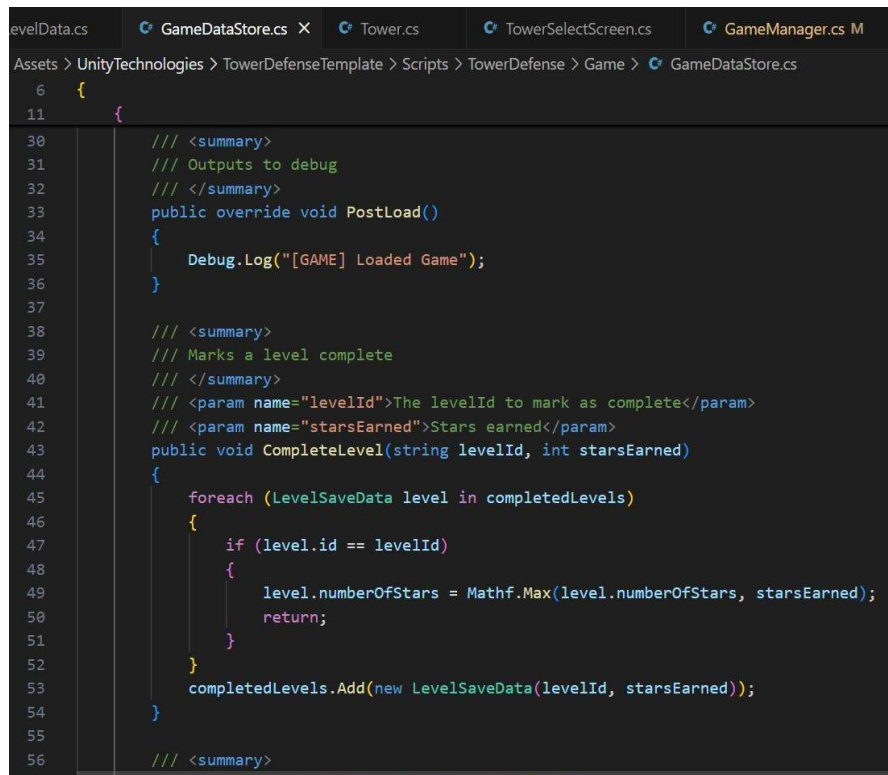
The script in figure [3.4,3.8] implements the GameManager for whole game and every level. This class contain code wrappers for implementing the core methods for towers and levels unlocking and selecting logic.

3.4 GameDataStore.cs



```
levelData.cs  GameDataStore.cs X  Tower.cs  TowerSelectScreen.cs  GameManager.cs M
Assets > UnityTechnologies > TowerDefenseTemplate > Scripts > TowerDefense > Game > GameDataStore.cs
1  using System.Collections.Generic;
2  using Core.Data;
3  using UnityEngine;
4
5  namespace TowerDefense.Game
6  {
7      /// <summary>
8      /// The data store for TD
9      /// </summary>
10     public sealed class GameDataStore : GameDataStoreBase
11     {
12         /// <summary>
13         /// A list of level IDs for completed levels
14         /// </summary>
15         public List<LevelSaveData> completedLevels = new List<LevelSaveData>();
16
17         /// <summary>
18         /// A list of level IDs for completed levels
19         /// </summary>
20         public List<TowerSaveData> unlockedTowers = new List<TowerSaveData>();
21
22         /// <summary>
23         /// Outputs to debug
24         /// </summary>
25         public override void PreSave()
26         {
27             Debug.Log("[GAME] Saving Game");
28         }
29     }
```

Figure 3.9: GameDataStore Script Part 1



```
levelData.cs  GameDataStore.cs X  Tower.cs  TowerSelectScreen.cs  GameManager.cs M
Assets > UnityTechnologies > TowerDefenseTemplate > Scripts > TowerDefense > Game > GameDataStore.cs
6  {
11 {
30     /// <summary>
31     /// Outputs to debug
32     /// </summary>
33     public override void PostLoad()
34     {
35         Debug.Log("[GAME] Loaded Game");
36     }
37
38     /// <summary>
39     /// Marks a level complete
40     /// </summary>
41     /// <param name="levelId">The levelId to mark as complete</param>
42     /// <param name="starsEarned">Stars earned</param>
43     public void CompleteLevel(string levelId, int starsEarned)
44     {
45         foreach (LevelSaveData level in completedLevels)
46         {
47             if (level.id == levelId)
48             {
49                 level.numberOfStars = Mathf.Max(level.numberOfStars, starsEarned);
50                 return;
51             }
52         }
53         completedLevels.Add(new LevelSaveData(levelId, starsEarned));
54     }
55
56     /// <summary>
57     /// Determines if a specific level is completed
58 }
```

Figure 3.10: GameDataStore Script Part 2


```

11      {
12
13          /// <summary>
14          /// Determines if a specific level is completed
15          /// </summary>
16          /// <param name="levelId">The level ID to check</param>
17          /// <returns>true if the level is completed</returns>
18          public bool IsLevelCompleted(string levelId)
19          {
20              foreach (LevelSaveData level in completedLevels)
21              {
22                  if (level.id == levelId)
23                  {
24                      return true;
25                  }
26              }
27              return false;
28          }
29
30          /// <summary>
31          /// Marks a tower unlock
32          /// </summary>
33          /// <param name="levelId">The levelId to mark as complete</param>
34          /// <param name="starsEarned">Stars earned</param>
35          public void UnlockTower(int ind)
36          {
37              foreach (TowerSaveData tower in unlockedTowers)
38              {
39                  if (tower.index == ind)
40                  {
41                      //level.number

```

Figure 3.11: GameDataStore Script Part 3

```

LevelData.cs  GameDataStore.cs M X  Tower.cs  TowerSelectScreen.cs  GameManager.cs
Assets > UnityTechnologies > TowerDefenseTemplate > Scripts > TowerDefense > Game > GameDataStore.cs
6      {
11      {
12
13          {
14
15              {
16
17                  {
18                      //level.number
19                      return;
20                  }
21              }
22          }
23          unlockedTowers.Add(new TowerSaveData(ind));
24      }
25
26      /// <summary>
27      /// Determines if a specific tower is unlocked
28      /// </summary>
29      /// <param name="levelId">The level ID to check</param>
30      /// <returns>true if the level is completed</returns>
31      public bool IsTowerUnlocked(int ind)
32      {
33          foreach (TowerSaveData tower in unlockedTowers)
34          {
35              if (tower.index == ind)
36              {
37                  return true;
38              }
39          }
40          return false;
41      }
42
43      }

```

Figure 3.12: GameDataStore Script Part 4

```

109     /// Select a tower
110     /// </summary>
111     public void SelectTower(int ind)
112     {
113         foreach (TowerSaveData tower in unlockedTowers)
114         {
115             if (tower.index == ind)
116             {
117                 tower.isSelected = true;
118                 return;
119             }
120         }
121     }
122
123     /// <summary>
124     /// Marks all towers as unselected
125     /// </summary>
126     public void DeSelectAllTowers()
127     {
128         foreach (TowerSaveData tower in unlockedTowers)
129         {
130             tower.isSelected = false;
131         }
132     }
133
134     /// <summary>
135     ///

```

Figure 3.13: GameDataStore Script Part 5

```

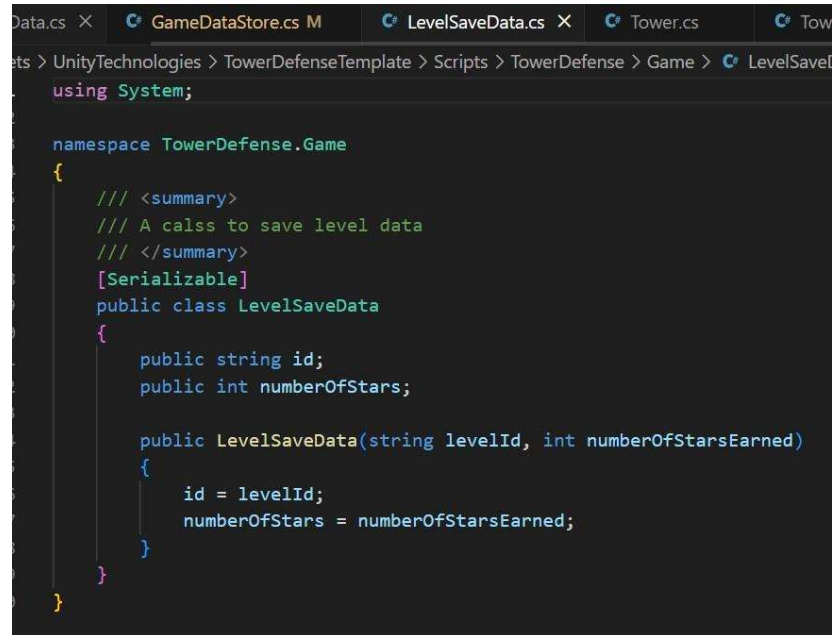
11     {
137     public bool isTowerSelected(int ind)
138     {
139         foreach (TowerSaveData tower in unlockedTowers)
140         {
141             if (tower.index == ind && tower.isSelected == true)
142             {
143                 return true;
144             }
145         }
146         return false;
147     }
148
149     /// <summary>
150     /// Retrieves the star count for a given level
151     /// </summary>
152     public int GetNumberOfStarForLevel(string levelId)
153     {
154         foreach (LevelSaveData level in completedLevels)
155         {
156             if (level.id == levelId)
157             {
158                 return level.numberOfStars;
159             }
160         }
161         return 0;
162     }
163 }
164

```

Figure 3.14: GameDataStore Script Part 6

The script in figure [3.9 - 3.14] is a Data Storage Container and contains the implementation of the levels and towers unlocking and selecting. The data is saved in a file in persistent path of the device.

3.5 LevelSaveData.cs



```
using System;

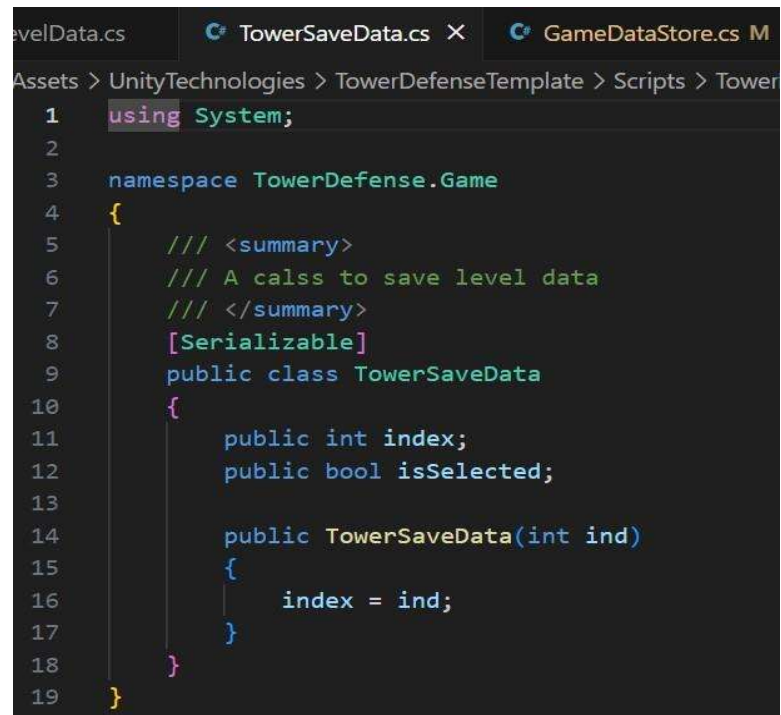
namespace TowerDefense.Game
{
    /// <summary>
    /// A class to save level data
    /// </summary>
    [Serializable]
    public class LevelSaveData
    {
        public string id;
        public int numberOfStars;

        public LevelSaveData(string levelId, int numberOfStarsEarned)
        {
            id = levelId;
            numberOfStars = numberOfStarsEarned;
        }
    }
}
```

Figure 3.15: LevelSaveData Script

The script in figure [3.15] is a serialized class for saving level data.

3.6 TowerSaveData.cs

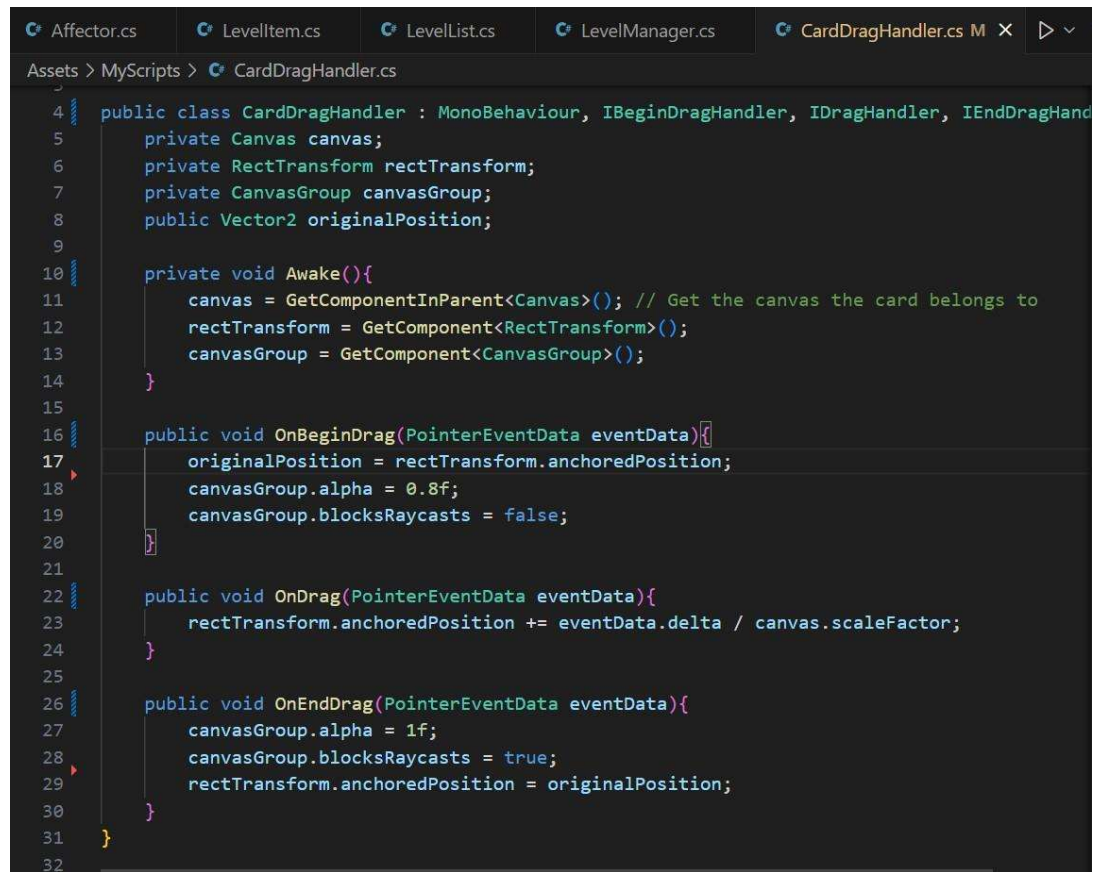


```
1  using System;
2
3  namespace TowerDefense.Game
4  {
5      /// <summary>
6      /// A class to save level data
7      /// </summary>
8      [Serializable]
9      public class TowerSaveData
10     {
11         public int index;
12         public bool isSelected;
13
14         public TowerSaveData(int ind)
15         {
16             index = ind;
17         }
18     }
19 }
```

Figure 3.16: TowerSaveData Script

The script in figure [3.16] is a serialized class for saving tower data.

3.7 CardDragHandler.cs



```
Assets > MyScripts > CardDragHandler.cs
4 public class CardDragHandler : MonoBehaviour, IBeginDragHandler, IDragHandler, IEndDragHandler
5 {
6     private Canvas canvas;
7     private RectTransform rectTransform;
8     private CanvasGroup canvasGroup;
9     public Vector2 originalPosition;
10
11     private void Awake()
12     {
13         canvas = GetComponentInParent<Canvas>(); // Get the canvas the card belongs to
14         rectTransform = GetComponent<RectTransform>();
15         canvasGroup = GetComponent<CanvasGroup>();
16     }
17
18     public void OnBeginDrag(PointerEventData eventData)
19     {
20         originalPosition = rectTransform.anchoredPosition;
21         canvasGroup.alpha = 0.8f;
22         canvasGroup.blocksRaycasts = false;
23     }
24
25     public void OnDrag(PointerEventData eventData)
26     {
27         rectTransform.anchoredPosition += eventData.delta / canvas.scaleFactor;
28     }
29
30     public void OnEndDrag(PointerEventData eventData)
31     {
32         canvasGroup.alpha = 1f;
33         canvasGroup.blocksRaycasts = true;
34         rectTransform.anchoredPosition = originalPosition;
35     }
36 }
```

Figure 3.18: CardDragHandler Script

The script in figure [3.18] contains the implementation of unity built-in methods for drag and drop.

3.8 DropZone.cs

```
public class DropZone : MonoBehaviour, IDropHandler
{
    [SerializeField] private int maxCards = 1; // Set the maximum number of cards allowed (

    public void OnDrop(PointerEventData eventData){
        GameObject droppedCard = eventData.pointerDrag;

        if (droppedCard != null){
            // Check if DropZone has reached its limit
            if (transform.childCount >= maxCards){
                // Replace the first card if the limit is reached
                Transform firstCard = transform.GetChild(0); // Get the first card
                TowerSelectScreen towerSelectScreen = transform.GetComponentInParent<TowerS
                firstCard.SetParent(towerSelectScreen.unSelectedParent);
                Debug.Log("Card replaced in DropZone.");
            }

            droppedCard.transform.SetParent(transform);

            RectTransform rectTransform = droppedCard.GetComponent<RectTransform>();
            if (rectTransform != null){
                rectTransform.anchoredPosition = Vector2.zero;
            }
            Debug.Log($"Card added to DropZone. Current count: {transform.childCount}/{maxC
        }
    }
}
```

Figure 3.19: DropZone Script

The script in figure [3.19] creates a drop zone for the draggable object.

3.9 TowerSelector.cs

```
12 public class TowerSelector : MonoBehaviour{
13
14     [Header("Buttons")]
15     public Button nextButton;
16     public Button prevButton;
17     public Button buyButton;
18
19     [Header("Texts")]
20     public TMP_Text nameText, maxHealthText, searchRateText, fireRateText, radiusText;
21     public TMP_Text IdleWaitTimeText, priceText, descText;
22
23     public TowerLibrary towerLib; // Reference to the library containing towers
24     private int index = 0; // Current tower index
25
26     public List<TowerItem> towers; // List of tower GameObjects in the scene
27     public float rotationSpeed = 1f;
28     // Start is called before the first frame update
29     void Start(){
30         nextButton.onClick.AddListener(NextTower);
31         prevButton.onClick.AddListener(PreviousTower);
32         buyButton.onClick.AddListener(Purchase);
33
34         if (towerLib == null || towerLib.Count == 0){
35             Debug.LogError("Tower Library is empty or not assigned.");
36             return;
37         }
38         DisplayTower(0);
39     }
40
41
42
43 }
```

Figure 3.20: TowerSelector Script Part 1

```
12 public class TowerSelector : MonoBehaviour{
13
14     // Update is called once per frame
15     void Update()
16     {
17         transform.Rotate(new Vector3(0f, 1f*Time.deltaTime*rotationSpeed, 0f));
18         if (Input.GetKeyDown(KeyCode.LeftArrow))
19         {
20             PreviousTower();
21         }
22         else if (Input.GetKeyDown(KeyCode.RightArrow))
23         {
24             NextTower();
25         }
26
27         if (Input.GetKeyDown(KeyCode.Return))
28         {
29             //SelectTower();
30         }
31     }
32
33     // Enable the tower at the given index and disable others
34     public void DisplayTower(int newIndex)
35     {
36         for (int i = 0; i < towers.Count; i++){
37             towers[i].towerPrefab.SetActive(i == newIndex);
38         }
39         UpdateUI();
40     }
41 }
```

Figure 3.21: TowerSelector Script Part 2


```

72 | public void NextTower(){
73 |     index = (index + 1) % towers.Count;
74 |     DisplayTower(index);
75 |
76 | }
77 |
78 | public void PreviousTower(){
79 |     index = (index - 1 + towers.Count) % towers.Count;
80 |     DisplayTower(index);
81 |
82 | }
83 |
84 | public void Purchase(){
85 |     int currentCurrency;
86 |     GameManager.instance.GetCurrency(out currentCurrency);
87 |
88 |     if (currentCurrency >= towers[index].price){
89 |         GameManager.instance.UnlockTower(index);
90 |         GameManager.instance.SetCurrency(currentCurrency - towers[index].price, true);
91 |         Debug.Log("Purchased");
92 |     }
93 |
94 |     else{
95 |         Debug.Log("NOT ENOUGH MONEY");
96 |     }
97 |     UpdateUI();
98 | }

```

Figure 3.22: TowerSelector Script Part 3

```

100 | void UpdateUI(){
101 |
102 |     AttackAffector affector = towerLib.configurations[index].levels[0].GetComponentInCh
103 |     Targetter targetter = towerLib.configurations[index].levels[0].GetComponentInChildr
104 |
105 |     nameText.text = towerLib[index].towerName;
106 |     descText.text = towerLib[index].levels[0].description;
107 |     priceText.text = "Price:" + towers[index].price.ToString();
108 |     maxHealthText.text = towerLib[index].levels[0].maxHealth.ToString();
109 |
110 |     radiusText.text = targetter != null ? targetter.effectRadius.ToString() : "N/A";
111 |     searchRateText.text = affector != null ? affector.fireRate.ToString() : "N/A";
112 |     fireRateText.text = affector != null ? affector.fireRate.ToString() : "N/A";
113 |     IdleWaitTimeText.text = targetter != null ? targetter.idleWaitTime.ToString() : "N/
114 |
115 |     bool isUnlocked = GameManager.instance.IsTowerUnlocked(index);
116 |     if (isUnlocked){
117 |         lockImage.SetActive(false);
118 |         buyButton.gameObject.SetActive(false);
119 |         priceText.gameObject.SetActive(false);
120 |     }
121 |     else{
122 |         lockImage.SetActive(true);
123 |         buyButton.gameObject.SetActive(true);
124 |         priceText.gameObject.SetActive(true);
125 |     }
126 |
127 | }

```

Figure 3.23: TowerSelector Script Part 4

The script in figure [3.20 – 3.23] implements the tower shop. It contain methods for selecting and buying different towers.