Data Structures Final Project Reflection Report

By Stephen Zoccoli.

Tier 1:

- ShopManager.cs used to use a List<> of Artifacts for the shop. The list was fixed to size 6, or the maximum number of undiscovered artifacts remaining (whichever is less). Since the size is fixed and pre-determined, I switched to a fixed size Array.
- The old algorithm shuffled all undiscovered artifacts, and took the first 6 of the shuffle (Fisher Yates). I changed to do a partial Fisher Yates shuffle, only up to the first 6 items.
- (Insert algorithm into PDF. Insert Array change).

```
+     public A_Base[] GetRandomArtifacts(int maxItems)  //List<A_Base> GetRandomArtifacts(int maxItems)
      {
+         int itemCount = Mathf.Min(maxItems, undiscoveredArtifacts.Count);
+         // 0 to itemCount - 1
+         int[] itemNums = new int[itemCount];
+         for (int i = 0; i < itemCount; i++) { itemNums[i] = i; }
+
+         // We use an Array instead of a list since the size is fixed
+         A_Base[] selectedItems = new A_Base[itemCount];
+         // Partial Fisher-Yates shuffle for random selection
+         for (int i = 0; i < itemCount; i++)
+         {
+             int randomIndex = Random.Range(i, itemCount);
+             int temp = itemNums[i];
+             itemNums[i] = itemNums[randomIndex];
+             itemNums[randomIndex] = temp;
+
+             selectedItems[i] = undiscoveredArtifacts[itemNums[i]];
+
+         }
```

- The change in ShopManager.cs required a small change in ArtifactManager.cs, now getting an Array from the respective function instead of a List.

Tier 2:

- PlayerStats.cs had several separate variables for the “# enemies defeated”, “artifacts discovered”, “damage dealt”, “health remaining”, etc. Instead, I used a hash table (dictionary) called “playerStatDict” for all of these.

```
9   -       public int enemiesDefeated;
10  -       public int artifactsDiscovered;
11  -       public int artifactsTriggered;
12  -       public int damageDealt;
13  -       public int damageTaken;
14  -       public int healthHealed;
15  -       public int coinsCollected;
        10  +
        11  +       public Dictionary<string, int> playerStatsDic = new Dictionary<string, int>();
```

- All variables were initialized to 0. Instead of doing it manually for each variable, now with a dictionary, I can for-loop “foreach key” and set all to 0. (include code).

```
17  20      public void ResetStats()
18  21      {
19  22          win = false;
20  23          stage = 0;
21  24          time = 0;
22      -       enemiesDefeated = 0;
23      -       artifactsDiscovered = 0;
24      -       artifactsTriggered = 0;
25      -       damageDealt = 0;
26      -       damageTaken = 0;
27      -       healthHealed = 0;
28      -       coinsCollected = 0;
    25  +
    26  +       string[] keys = { "enemiesDefeated", "artifactsDiscovered", "artifactsTriggered", "damageDealt", "damageTaken", "healthHealed", "coinsCollected" };
    27  +       foreach (string key in keys)
    28  +       {
    29  +           if (playerStatsDic.ContainsKey(key))
    30  +           {
    31  +               playerStatsDic[key] = 0;
    32  +           }
    33  +           else
    34  +           {
    35  +               playerStatsDic.Add(key, 0);
    36  +           }
    37  +
    38  +       }
```

- This change propagated to several other files that used these variables. That took some time to find everything.

Tier 3:

DungeonManager.cs randomly generated the dungeon as a tree. This is done layer by layer up to a max tree depth. I pre-defined a tree structure, then made the dungeon follow the pre-defined tree.

```
51  +           Tree tree = new Tree();
52  +
53              // Generate first room
    -           CreateRoom(stage, 0, null);
54  +           CreateRoom(stage, 0, null, tree.root);
55              // Iteratively generate layers
    -           GenerateLayer(stage, 1);
56  +           GenerateLayer(stage, 1, tree);
```

To get this to work with the current tree building algorithm, I needed to write a getLayer function for the tree, to integrate into the current dungeon build.

```
    void GenerateLayer(int stage, int depth)
    void GenerateLayer(int stage, int depth, Tree tree)
    {
        // Stop when reached max depth or no doors to fill
        if (depth > gen.maxDepth || doorsToFill.Count <= 0) return;
@@ -108,17 +111,19 @@ void GenerateLayer(int stage, int depth)
            _doorsToFill.Add(doorsToFill[i]);
        doorsToFill.Clear();

        List<Node> _nodes = tree.GetLayer(depth);

        // Create new room for all empty doors
        for (int i = 0; i < _doorsToFill.Count; i++)
        {
            Room _parentRoom = _doorsToFill[i].GetComponent<EC_Entity>().room;
            Room _newRoom = CreateRoom(stage, depth, _parentRoom);
            Room _newRoom = CreateRoom(stage, depth, _parentRoom, _nodes[i]);
            _doorsToFill[i].destination = _newRoom;
            _parentRoom.children.Add(_newRoom);
        }

        // Generate next layer
        GenerateLayer(stage, depth + 1);
        GenerateLayer(stage, depth + 1, tree);
    }
```

```
+     public List<Node> GetLayer(int depth)
+     {
+         List<Node> layer = new List<Node>();
+         if (depth == 0)
+         {
+             layer.Add(root);
+             return layer;
+         }
+
+         if (depth == 1)
+         {
+             if (root.left != null)
+             {
+                 layer.Add(root.left);
+             }
+             if (root.right != null)
+             {
+                 layer.Add(root.right);
+             }
+             return layer;
+         }
+
+         // depth > 1
+         if (root.left != null)
+         {
+             layer.AddRange(GetLayer(depth, root.left));
+         }
+         if (root.right != null)
+         {
+             layer.AddRange(GetLayer(depth, root.right));
+         }
+
+
+         return layer;
+     }
```

In all cases, the end result is the same, but the code should work faster, i.e., optimized.