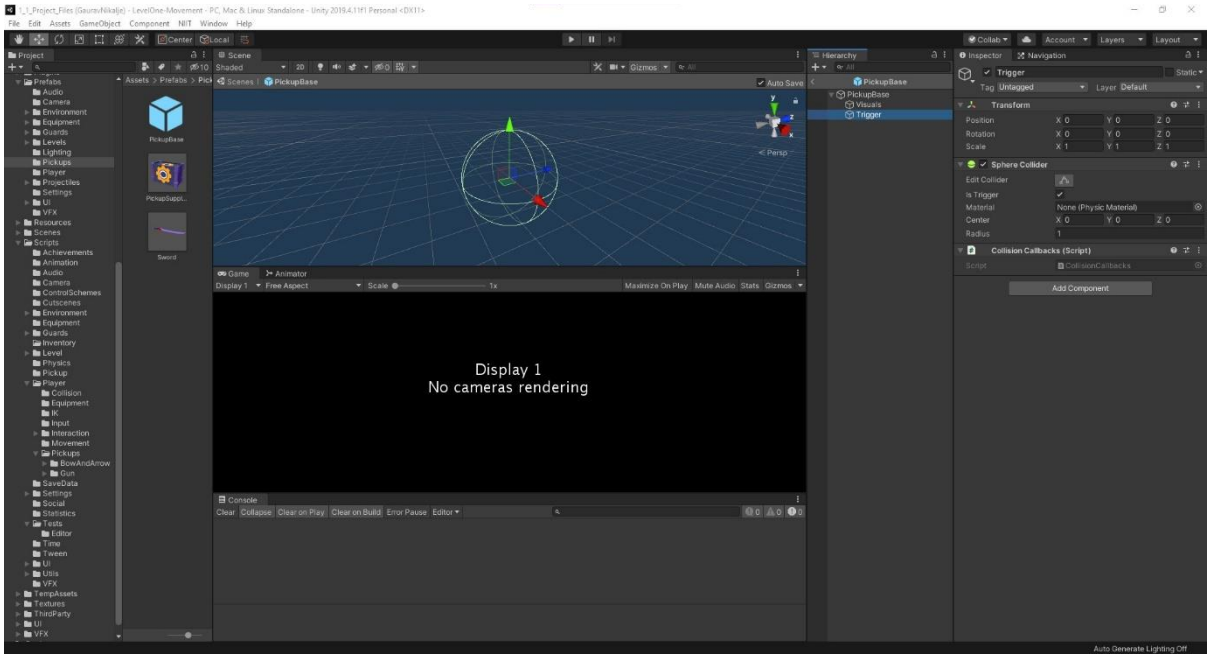
Sprint 01

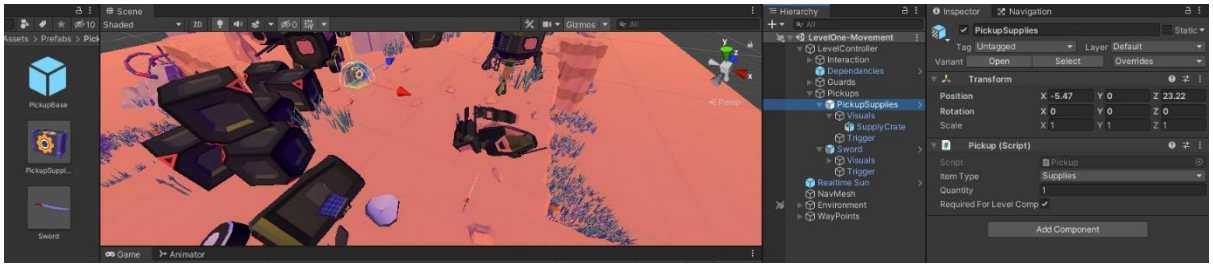
Assignment 04 : Inventory Part I

Step 01: Setup

- First, we need to create a controller to control the Inventory Items.
- So, we create a script called "*InventoryController*" in the "Scripts > Inventory" folder.
- Then create a Prefab called "*PickupBase*" and add two Empty GameObjects to it: Visuals & Trigger.
- Under Trigger, add a "*SphereCollider*" and "*CollisionCallbacks*" script.



- Then create a prefab variant of the "*PickupBase*" and rename it as preference.
- Then under the Visuals GameObject, add the 3D Model of the Pickup Item.
- Do the same for other Pickup Objects.



Step 02: Script & Workflow

- The Scripts workflow is like this:
  - InventoryController > ItemType & ItemData (classes) > Pickup > PickupController > PickupEvents > LevelController > PlayerController > PlayerEquipmentController.

- Where, "*Pickup*" is a MonoBehavior class which is attached to Pickup Items Prefab Variants.
- "*PickupEvents*" acts as the main parent object between "*PickupController*" & its associated classes and "*LevelController*" & its associated classes.
- The InventoryController consists of two classes "ItemType" and "*ItemData*".
- "*PickupEvents*" simply processes the Picked Up Items from one script to another.
- Just like in the "*Scene Loading*" Assignment, where we passed the value of "*level*".

❖ InventoryController.

- It consists of two sub classes "*ItemType*" and "*ItemData*".
- InventoryController is basically a class to Add, Remove, select Primary and Secondary items and etc.
- The Contents Dictionary is of a "*get*" type used to get anything that is in the Inventory into Player's hand.
- Also, there is a "*Copy()*" function in the "*ItemData*" class which simply copies the "***values***" to the "***key***" of the Dictionary.

```
C# InventoryController.cs U ×   C# Pickup.cs U   C# PickupController.cs U   C# PickupEvents.cs U   C# LevelController.cs M   C# PlayerController.cs M   C# PlayerEquipmentController.cs U
Assets > Scripts > Inventory > C# InventoryController.cs > InventoryController
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

9 references
public class InventoryController
{
    2 references
    public const int MAX_NUMBER_PER_CATEGORY = 99;
    7 references
    public Action<ItemType, int> OnItemCountUpdated = delegate { };  //This function is used to keep track of items (Type as well as its Count).

    //A Dictionary is a collection of Values that can be accessed by One or More Keys.
    //Dictionary<Key, Value>
    18 references
    private Dictionary<ItemType, ItemData> heldItems;
    3 references
    private ItemData primary;
    3 references
    private ItemData secondary;

    2 references
    public InventoryController()
    {
        heldItems = new Dictionary<ItemType, ItemData>();
        foreach(ItemType pickupItem in Enum.GetValues(typeof(ItemType)))
        {
            heldItems[pickupItem] = new ItemData();  //We do this bcoz we need to add New ItemData values(Count & canDrop) to each ItemType(Melee, Gun, etc...)
        }
    }

```

```
4 references
public bool CanAdd(ItemType type)
{
    return heldItems[type].count < MAX_NUMBER_PER_CATEGORY;
}

    //The below function simply Adds items to the Inventory and Notifies using the delegate function.
    6 references
    public void Add(ItemType type)
    {
        if(CanAdd(type))
        {
            heldItems[type].count++;
            OnItemCountUpdated(type, heldItems[type].count);
        }
    }

//The below function helps to only pickup items based on Inventory's Capacity.
//i.e. suppose if there are 20 ammos stack on the gorund, and the Inventory can only add 10 more ammos stack,
//then the Player will take only 10 ammos stack into the inventory and leave the rest 10 on the ground.
1 reference
public void Add(ItemType type, int amount)
{
    for (int i = 0; i < amount; i++)
    {
        if(CanAdd(type))
        {
            heldItems[type].count++;
            OnItemCountUpdated(type, heldItems[type].count);
        }
    }
}

2 references
public bool CanRemove(ItemType type)
{
    return heldItems[type].count > 0 && heldItems[type].canDrop;    //canDrop = true. (Already set to True in the "ItemData" Constructor)
}
1 reference
public void Remove(ItemType type)
{
    if(CanRemove(type))
    {
        heldItems[type].count--;
        OnItemCountUpdated(type, heldItems[type].count);
    }
}
```

```
    public void RemoveAll(ItemType type)
    {
        heldItems[type].count = 0;
        OnItemCountUpdated(type, heldItems[type].count);
    }

    1 reference
    public void SetAsUndroppable(ItemType type)
    {
        heldItems[type].canDrop = false;
    }

    0 references
    public int GetCount(ItemType type)
    {
        return heldItems[type].count;
    }

    0 references
    public void EquipPrimary(ItemType type)
    {
        primary = heldItems[type];
    }
    0 references
    public void UnEquipPrimary(ItemType type)
    {
        primary = null;
    }

    0 references
    public void EquipSecondary(ItemType type)
    {
        secondary = heldItems[type];
    }

    0 references
    public void UnEquipSecondary(ItemType type)
    {
        secondary = null;
    }
```

```
    //The below function is used get Everything thats in our Inventory but not in our Hands.
    //To access all the Contents available in our Inventory, but are not available in the Primary or Secondary slots.
    0 references
    public Dictionary<ItemType, ItemData> Contents
    {
        get     //Used to get values. In this case gets toReturn.
        {
            Dictionary<ItemType, ItemData> toReturn = new Dictionary<ItemType, ItemData>();

            foreach(KeyValuePair<ItemType, ItemData> item in heldItems)
            {
                if(item.Value != primary &&
                    item.Value != secondary &&
                    item.Value.count > 0)
                {
                    toReturn[item.Key] = item.Value.Copy(); //We simply copy the values and store it. (i.e., create a Clone or Duplicate)
                }
            }

            return toReturn;
        }
    }


}
```

```
public enum ItemType
{
    0 references
    Melee,
    1 reference
    Gun,
    12 references
    DamageAmmo,
    0 references
    ExplosiveAmmo,
    0 references
    GuardAmmo,
    0 references
    Supplies
}

11 references
public class ItemData   //Class
{
    15 references
    public int count;
    5 references
    public bool canDrop;

    2 references
    public ItemData()   //Constructor
    {
        count = 0;
        canDrop = true;
    }
    1 reference
    public ItemData Copy()
    {
        return new ItemData()
        {
            count = count,
            canDrop = canDrop
        };
    }
}
```

❖ Pickup.

```
InventoryController.cs U   Pickup.cs U ×   PickupController.cs U
Assets > Scripts > Player > Pickups > Pickup.cs > ...
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

10 references
public class Pickup : MonoBehaviour
{
    2 references
    public ItemType itemType;
    1 reference
    public int quantity;
    2 references
    public bool requiredForLevelCompletion;

}
```

❖ PickupController.

- It controls all the Picked-Up Items.

```
InventoryController.cs U   PickupController.cs U ×   Pickup.cs U   PickupEvents.cs U   LevelController.cs M   PlayerContr
Assets > Scripts > Player > Pickups > PickupController.cs > PickupController > SetupPickup(Pickup pickup)
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

2 references
public class PickupController
{
    1 reference
    private Transform levelObjects; //levelObjects = "LevelController" GameObject's Transform in each Level.
    2 references
    private PlayerController playerController;
    4 references
    private List<Pickup> requiredPickups;
    3 references
    private List<Pickup> activePickups;
    3 references
    private  PickupEvents events;

    1 reference
    public PickupController(Transform levelObjects, PlayerController playerController, PickupEvents pickupEvents)
    {
        this.levelObjects = levelObjects;
        this.playerController = playerController;
        this.events = pickupEvents;

        FindPickupsInLevel(levelObjects);
```

- It looks for Active & Required Items.
- It looks every "*Pickup Item*" under the "*Pickups*" group and assigns "*Pickup*" script to each.
- It also invokes Pickup Event.

```
private void FindPickupsInLevel(Transform levelObjects)
{
    requiredPickups = new List<Pickup>();
    activePickups = new List<Pickup>();

    Transform pickupRoot = levelObjects.Find("Pickups");    //Pickups = is a Empty GameObject under LevelController.

    if(pickupRoot == null)
    {
        Debug.LogError("PickupController tried to find 'Pickups' GameObject, but failed!");
        return;
    }

    //pickupRoot = Pickup GameObjects container. (Contains all the pickupTrans)
    //pickupTrans = Pickups GameObjetcs child under "pickupRoot"
    foreach (Transform pickupTrans in pickupRoot)
    {
        Pickup pickup = pickupTrans.GetComponent<Pickup>();

        if(pickup != null)
        {
            SetupPickup(pickup);
            events.OnPickupEventCollected(pickup);
        }
    }
}
```

- The "*SetupPickup()*" function is responsible for Adding items only once.
- It triggers a Pickup Event using the Collision & Collider Trigger event from the "*CollisionsCallBack*" script.

```
    private void SetupPickup(Pickup pickup)
    {
        activePickups.Add(pickup);  //Bcoz it is our Current Pickup Object.

        CollisionCallbacks collisionCallbacks = pickup.GetComponentInChildren<CollisionCallbacks>();
        Action<Collider> OnPickupCollision = null;  //We create this separately, as we need to Subscribe & UnSubscribe from the Pickup Events.

        //Workflow: OnTriggerEntered > OnPickupCollision > Lambda Function.
        //Bcoz Unity looks where is "OnPickupCollision" is called or subscribed and then executes it and then enters the Lambda Expression.
        //No matter where you place the "OnPickupCollision" Subscriber before the Lamba Expression or after, it would look around to the
        //Subscriber and execute its commands respectively

        OnPickupCollision = (collider) =>
        {
            //The below code executes such as if we Pickup the Item once, we wont be able to Pick it up again.
            //Despite if we do not Set the object Active to false or Unhide it from the Inspector.

            if(playerController.OwnsCollider(collider))     //Checks for Player's seperate Own Collsions
            {
                collisionCallbacks.OnTriggerEntered -= OnPickupCollision;   //Garbage Code clean up.

                //We remove our Current & Required Pickups in Order to perform the Drop Command
                activePickups.Remove(pickup);
                if(requiredPickups.Contains(pickup))
                {
                    requiredPickups.Remove(pickup);
                }

                events.OnPickupEventCollected(pickup);
            }
        };

        collisionCallbacks.OnTriggerEntered += OnPickupCollision;

        if(pickup.requiredForLevelCompletion)
        {
            requiredPickups.Add(pickup);    //Adds Current Pickups under Required for LevelCompletion context.
        }
    }

}
```

❖ **CollisionCallBacks.**

- It Contains all the Collision event triggered by respective delegates so that these events can be accessed in any script at any time.

C# InventoryController.cs U | C# CollisionCallbacks.cs × | C# PickupController.cs U

Assets > Scripts > Physics > C# CollisionCallbacks.cs > ...

```
using System;
using UnityEngine;

3 references
public class CollisionCallbacks : MonoBehaviour
{
    3 references
    public Action<Collider> OnTriggerEntered = delegate { };
    1 reference
    public Action<Collision> OnCollisionEntered = delegate { };
    2 references
    public Action<Collider> OnTriggerStayed = delegate { };
    2 references
    public Action<Collider> OnTriggerExited = delegate { };

    0 references
    private void OnTriggerEnter(Collider other)
    {
        OnTriggerEntered(other);
    }

    0 references
    private void OnCollisionEnter(Collision collision)
    {
        OnCollisionEntered(collision);
    }

    0 references
    private void OnTriggerStay(Collider other)
    {
        OnTriggerStayed(other);
    }

    0 references
    private void OnTriggerExit(Collider other)
    {
        OnTriggerExited(other);
    }
```

❖ **PickupEvents.**

- It simply consists of an Action delegate which is used to passed the Pickup value.

```
C# InventoryController.cs U   C# PickupEvents.cs U ×   C# PickupController.cs U   C#
Assets > Scripts > Player > Pickups > C# PickupEvents.cs > ...
1   using System;
2   using UnityEngine;
3
    4 references
4   public class PickupEvents
5   {
        3 references
6       public Action<Pickup> OnPickupEventCollected = delegate { };
7   }
8
```

❖ **LevelController.**

- It simply creates instances required classes.
- And pass the Picked-Up Item value form this script to another through delegates & functions.

C# InventoryController.cs U | C# LevelController.cs M × | C# PickupEvents.cs U | C# PickupController.cs U | C# Pickup.cs U | C# PlayerController.cs M

Assets > Scripts > Level > C# LevelController.cs > LevelController > Start()

```
3 references
private GuardManager guardManager;  //Assignment - 03
3 references
private TimeController timecontroller;  //Assignment - 01
4 references
private LevelStatsController levelStatsController;
//private PlayerEquipmentController equipmentController;  //Assignment 04 - Part I
1 reference
private PickupController pickupController;  //Assignment 04 - Part I
3 references
private InventoryController inventory;  //Assignment 04 - Part I
//private PickupEvents pickupEvents;    //Assignment 04 - Part I

0 references
public void Start()
{
    LevelDependancies dependancies = GetComponentInChildren<LevelDependancies>();
    if(dependancies == null)
    {
        Debug.LogError("Unable to find LevelDependancies. Cannot play level.");
    }

    GameObject playerObj = CreatePlayerObject(dependancies.player);

    PickupEvents pickupEvents = new PickupEvents();     //Assignment - 04 Part I

    player = CreatePlayer(playerObj);       //Assignment - 04 Part I

    pickupController =  new PickupController(transform, player.Controller, pickupEvents);   //Assignment 04 - Part I

    pickupEvents.OnPickupEventCollected += (pickup) =>  //Assignment 04 - Part I
    {
        player.Controller.OnPickupCollected(pickup);    //Inside Player Controller
    };

    player.Controller.OnDeathSequenceCompleted += () =>
    {
        FailLevel();
    };

```

```
1 reference
private Player CreatePlayer(GameObject playerObject)
{
    NavMeshAgent navMeshAgent = playerObject.GetComponent<NavMeshAgent>();

    PlayerObjectData objectData = playerObject.GetComponent<PlayerObjectData>();

    PlayerCollision collision = new PlayerCollision(playerObject.transform);

    PlayerInteractionController interaction = new PlayerInteractionController(transform, playerObject.transform, collision, objectData);

    PlayerInputBroadcaster broadcaster = new PlayerInputBroadcaster();

    inventory = new InventoryController();  //Assignment 04 - Part I

    PlayerEquipmentController equipmentController = new PlayerEquipmentController(inventory, objectData);   //Assignment 04 - Part I

    PlayerController controller = new PlayerController(playerObject.transform, navMeshAgent,
        interaction, collision, broadcaster, inventory, equipmentController);   //Assignment 04 - Part I

    broadcaster.Callbacks.OnPlayerStartUseFired += () => controller.StartUse();

    return new Player(controller, broadcaster,
        objectData, interaction);
}

```

❖ PlayerController.

- This script triggers the "*OnPlayerPickedUp*" function of the "*PlayerEquipmentController*" class and is the second last stage of passing value.

```
C# InventoryController.cs U   C# PlayerController.cs M ×   C# LevelController.cs M   C# PickupEvents.cs U   C# PickupController.cs U   C# Picku
Assets > Scripts > Player > C# PlayerController.cs > PlayerController > PlayerController(Transform transform, NavMeshAgent navMeshAgent, PlayerInteractionC
        1 reference
28      private float playerMaxHP = 100f;
        0 references
29      private PlayerInputCallbacks callbacks;
        2 references
30      private PlayerEquipmentController equipmentController;  //Assignment 04 - Part I
        1 reference
31      private InventoryController inventory;  //Assignment 04 - Part
32
        1 reference
33      public PlayerController(Transform transform,
34          NavMeshAgent navMeshAgent, PlayerInteractionController interactionController,
35          PlayerCollision collision, PlayerInputBroadcaster inputBroadcaster,
36           InventoryController inventory, PlayerEquipmentController equipmentController)  //Assignment 04 - Part I
37
38          this.transform = transform;
39          this.navMeshAgent = navMeshAgent;
40          this.interactionController = interactionController;
41          this.collision = collision;
42          this.inputBroadcaster = inputBroadcaster;
43          this.equipmentController = equipmentController;     //Assignment 04 - Part I
44          this.inventory = inventory; //Assignment 04 - Part I
45
46          playerHealth = new PlayerHealth(playerMaxHP);
47
48          playerHealth.OnDamageTaken += (currentHealth) =>
49          {
50              OnPlayerDamageTaken(currentHealth);
51          };
```

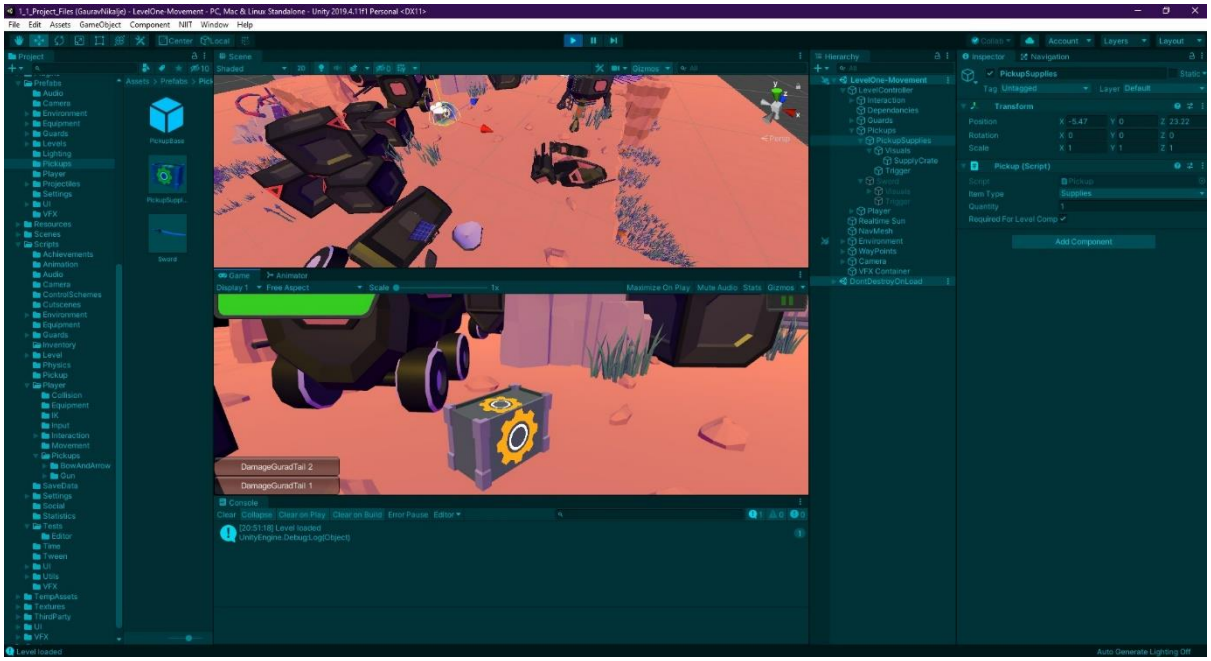
```
73
        1 reference
74      public void OnPickupCollected(Pickup pickup)    //Assignment - 04 Part I
75      {
76          equipmentController.OnPlayerPickedUp(pickup);
77      }
78
        0 references
79      public void TakeDamage(float damageAmount, Vector3 damageLocation)
80      {
81          lastDamageLocation = damageLocation;
82          playerHealth.TakeDamage(damageAmount);
83      }
84
```

❖ PlayerEquipmentController.

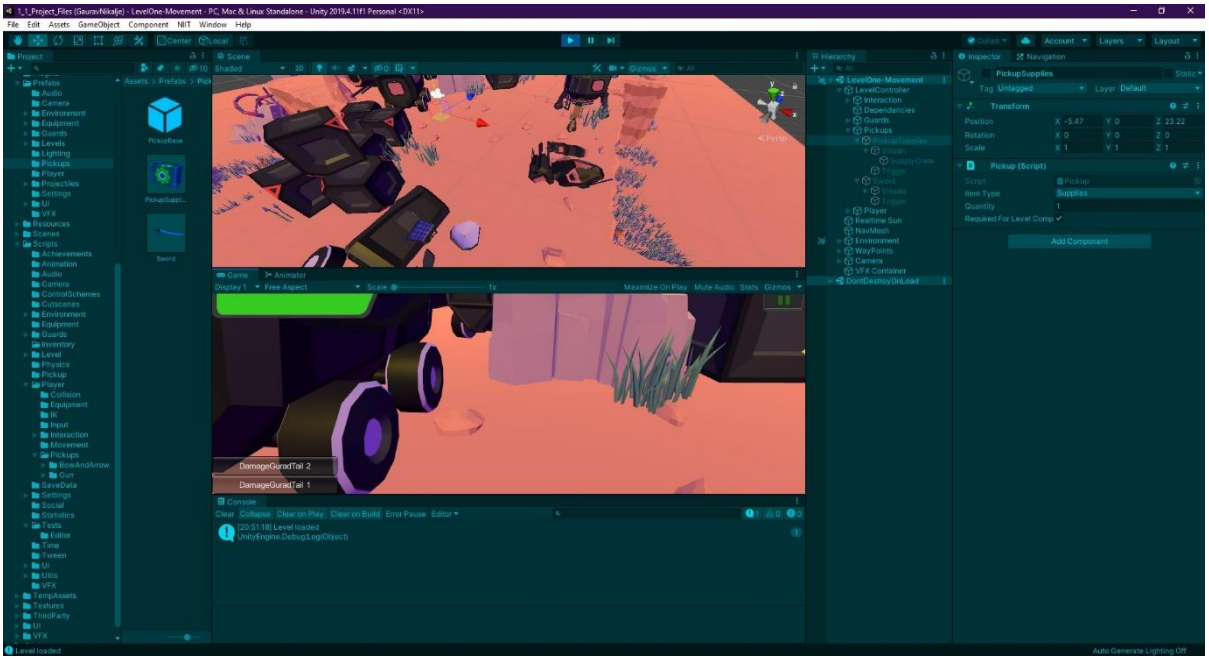
- This is the last stage where the processed value is added to the Inventory and the item picked up from the game disappears as soon as it is picked by Player.

```
C# InventoryController.cs U   C# PlayerEquipmentController.cs U ×   C# PlayerController.cs M   C# LevelController.cs M   C# PickupEvents.cs U
Assets > Scripts > Player > Equipment > C# PlayerEquipmentController.cs > PlayerEquipmentController > OnPlayerPickedUp(Pickup pickup)
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

4 references
public class PlayerEquipmentController
{
    3 references
    private InventoryController inventory;
    1 reference
    private PlayerObjectData playerObjectData;

    1 reference
    public PlayerEquipmentController(InventoryController inventoryController, PlayerObjectData playerObjectData)
    {
        this.inventory = inventoryController;
        this.playerObjectData = playerObjectData;
    }

    1 reference
    public void OnPlayerPickedUp(Pickup pickup)
    {
        inventory.Add(pickup.itemType, pickup.quantity);    //We access the 2nd Add function which takes 2 arguments.

        if(pickup.requiredForLevelCompletion)
        {
            inventory.SetAsUndroppable(pickup.itemType);
        }
        pickup.gameObject.SetActive(false);
    }
}
```

❖ Final Output.



Before



After (Picking Up)

❖ **InventoryTest.**

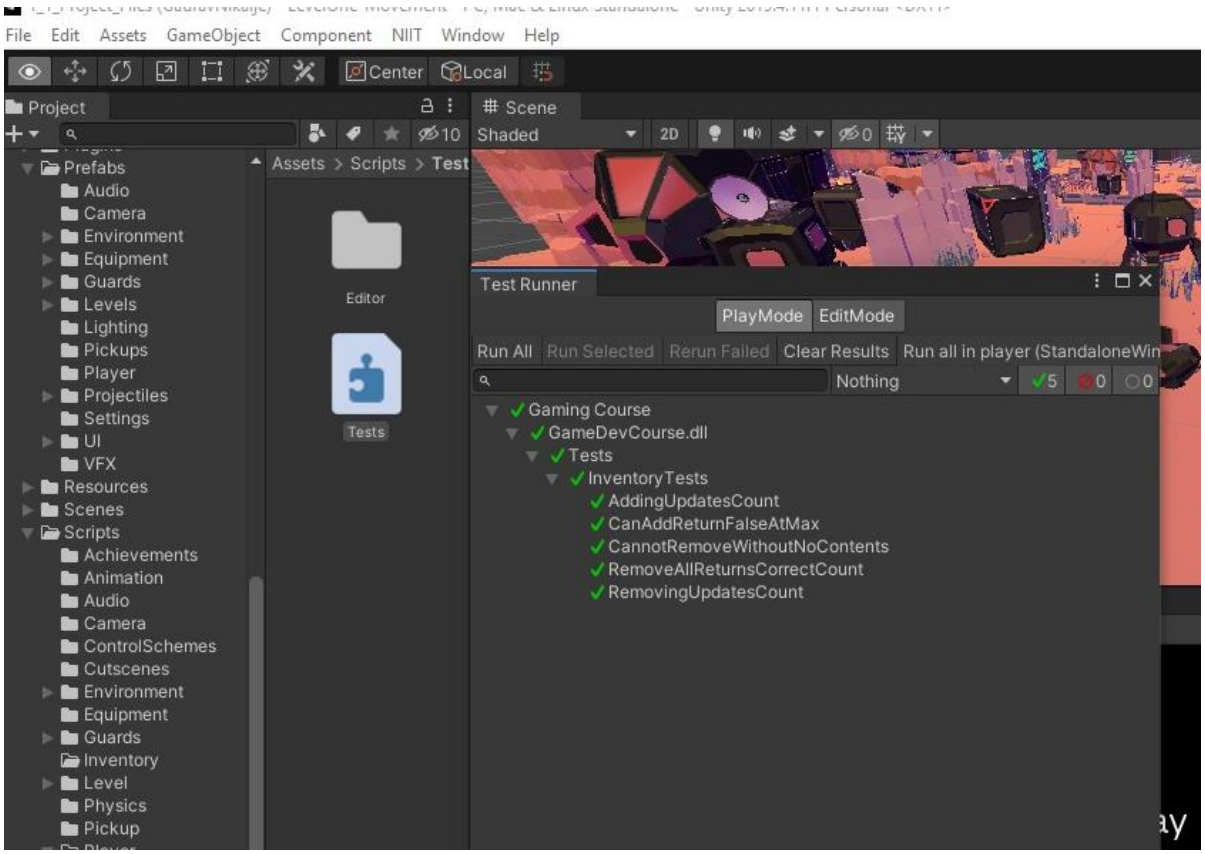
C# InventoryController.cs U | C# InventoryTests.cs U ✕ | C# PlayerEquipmentController.cs U

Assets > Scripts > Inventory > C# InventoryTests.cs > {} Tests > Tests.InventoryTests

```
using System.Collections;
using System.Collections.Generic;
using NUnit.Framework;
using UnityEngine;
using UnityEngine.TestTools;

namespace Tests
{
    0 references | Run All Tests | Debug All Tests
    public class InventoryTests
    {
        15 references
        private InventoryController inventory;
        [SetUp]
        0 references
        public void Setup()
        {
            inventory = new InventoryController();
        }
        // A Test behaves as an ordinary method
        [Test]
        0 references | Run Test | Debug Test
        public void AddingUpdatesCount()
        {
            bool updated = false;
            inventory.OnItemCountUpdated += (pickupType, count) =>
            {
                if(pickupType == ItemType.DamageAmmo)
                {
                    Assert.AreEqual(1, count);
                    updated = true;
                }
            };
            inventory.Add(ItemType.DamageAmmo);
            Assert.IsTrue(updated);
        }

```

```
34      [Test]
        0 references | Run Test | Debug Test
35      public void CanAddReturnFalseAtMax()
36      {
37          for(int i=0; i<InventoryController. MAX_NUMBER_PER_CATEGORY; i++
38          {
39              if(inventory.CanAdd(ItemType.DamageAmmo) == false)
40              {
41                  Assert.Fail();
42              }
43
44              inventory.Add(ItemType.DamageAmmo);
45          }
46
47          Assert.IsFalse(inventory.CanAdd(ItemType.DamageAmmo));
48      }
49      [Test]
        0 references | Run Test | Debug Test
50      public void RemovingUpdatesCount()
51      {
52          bool updated = false;
53          inventory.Add(ItemType.DamageAmmo);
54
55          inventory.OnItemCountUpdated += (pickupType, count) =>
56          {
57              Assert.AreEqual(0, count);
58              updated = true;
59          };
60
61          inventory.Remove(ItemType.DamageAmmo);
62          Assert.IsTrue(updated);
63      }
64      [Test]
        0 references | Run Test | Debug Test
65      public void CannotRemoveWithoutNoContents()
66      {
67          Assert.IsFalse(inventory.CanRemove(ItemType.Gun));
68      }
```

```
69          [Test]
            0 references | Run Test | Debug Test
70          public void RemoveAllReturnsCorrectCount()
71          {
72              bool updated = false;
73              inventory.Add(ItemType.DamageAmmo);
74              inventory.Add(ItemType.DamageAmmo);
75              inventory.Add(ItemType.DamageAmmo);
76
77              inventory.OnItemCountUpdated += (type, count) =>
78              {
79                  if(type == ItemType.DamageAmmo)
80                  {
81                      Assert.AreEqual(0, count);
82                      updated = true;
83                  }
84              };
85
86              inventory.RemoveAll(ItemType.DamageAmmo);
87              Assert.IsTrue(updated);
88          }
89      }
90  }
```



Step 03: What have I learnt

- **What is a design pattern?**

  - Design patterns are general repeatable solutions to commonly occurring problems in software design.
  - Patterns are about reusable designs and interactions of objects.
  - They have a defined Structure, which can be implemented in order to make a code easier to read, understand and make changes or modifications.
  - It is like having a Flowchart; where you setup the outline structure and then add things to it.
  - Some *Examples* are Observer Patter, Command Patter, Singleton Pattern, etc.

- **What is the observer pattern? Explain how it can be used in Unity. Give one non-inventory scenario where it could be useful**

  - Observer pattern is a behavioural design pattern that allows some objects to notify other objects about changes in their state.
  - The Observer pattern provides a way to *subscribe* and *unsubscribe* to and from these events for any object that implements.
  - The Main Script fires its event and the other respective scripts can access its contents, but the Main Script is unaware of it and doesn't really care.
  - With this, we can listen to as many events as we want and this gives flexibility to work among various scripts together.

- It can be used in Unity to reward Player with the Achievements.
- It can also be used to interact between environment, small interactions like: Opening & Closing of a Door and a Drawer, etc.
- Here a common Environment Event Trigger function can be used where open & close functionality can be accessed by both the Door & Drawer events.

- **What is the singleton pattern? Explain how you would create a singleton in Unity. What are the potential drawbacks of this pattern?**

  - The singleton pattern is a software design pattern that restricts the instantiation of a class to one "*single*" instance.
  - The singleton is denoted as "*static*" as prefix before any variable.
  - Example: public static instance;
  - Instance = this; - In Start() or Awake() function.
  - Singleton sets the variable as a Global Variable which can be accessed in other scripts by directly calling the Script name, then its instance reference and access the variables or functions in it.
  - Although, singleton pattern is not helpful everywhere, but some cases it is useful are, classes that have a unique instance in project, such as graphics pipeline, input libraries and more.
  - *And I am not sure, but can it also be used for Saving Game's Progress in Save Slots? As it can create multiple instances of specific Save Games (each different Save State in different Save Slots).*

- ***Disadvantages:***

  - Unit testing is more difficult (because it introduces a global state into an application).
  - They deviate from the Single Responsibility Principle.
  - Singletons can hide dependencies.
  - This pattern reduces the potential for parallelism within a program, because to access the singleton in a multi-threaded system, an object must be serialized (by locking).

The **Single Responsibility Principle** is one of the principles defined as part of the SOLID design pattern. It implies that **a unit, either a class, a function, or a microservice**, should have **one and only one** responsibility. At no point in time, one microservice should have more than one responsibility.

- **Research another design pattern from the following website, explain it and discuss an area of the game where it could potentially be implemented.**

  - States or Finite State Machines.
  - A finite state machine or state machine is a model used to represent and control execution flow.
  - Each state has a set of transitions, each associated with an input and pointing to a state.
  - When an input comes in, if it matches a transition for the current state, the machine changes to the state that transition points to.

- For example, pressing down while standing transitions to the ducking state.
- It is useful to avoid various bugs or glitches between two states or actions, like Jumping & Ducking at the same time, if we use the traditional "if-else" statements.
- We have used the finite state machines in the AI Project where we have created a Finite State Machine for our Guard, where there are transition set from walk to run, and vice versa.
- Also, some additional functionalities like hiding, searching can be added.
- The reason using FSM here, is that they do not transition or swtich to two or more states at one time, they simply transition from one state to another.
- Like this it is easire to add functionalities and avoid bugs & errors.

- **What are the benefits of creating a test suite? In what circumstances might this be more useful than others?**

  - The Unity Test Framework package (formerly the "*Unity Test Runner*") is a tool that allows you to test your code in both Edit mode and Play mode, and also on target platforms such as Standalone, Android, or iOS.
  - You should design a unit test to validate that a small, logical, snippet of code performs exactly as you expect it to in a specific scenario.

- ***Advantages:***

  - Provides confidence that a method behaves as expected.
  - Serves as documentation for new people learning the code base (unit tests make for great teaching).
  - Forces you to write code in a testable way.
  - Helps you isolate bugs faster and fix them quicker.
  - Prevents future updates from adding new bugs to old working code (known as regression bugs).

- ***Disadvantages:***

  - Writing tests can take longer than writing the code itself.
  - Bad or inaccurate tests create false confidence.
  - Requires more knowledge to implement correctly.
  - Important parts of the code base might not be easily testable.
  - Some frameworks don't easily allow private method testing, which can make unit testing harder.
  - If tests are too fragile (fail too easily for the wrong reasons), maintenance can take a lot of time.
  - Unit tests don't catch integration errors.
  - UI is hard to test.
  - Inexperienced developers might waste time testing the wrong things.
  - Sometimes, testing things with external or runtime dependencies can be very hard.

- A Unit Test will be helpful in cases where we have created a structure of execution and we want to check whether it functions properly as expected.
- Like Inventory Functionality, Score Functionality, Achievement Functionality, Skill Tree Functionality, and so on.

----------THE END----------