One of many projects i've worked on:

Unstoppable Swords

Overview:

Unstoppable Swords is a 2D side scrolling video game that requires the player to maneuver their character across platforms, simultaneously confronting enemies and obstacles along the way. Basic maneuvering will include running, jumping, and attacking. Players can collect diamonds/gems along the way in order to upgrade their weapon and acquire a boost of skills at a local in-game shop. The main focus of this video game is to get the player the required key in order to access the castle on the far right and complete the game.

Design Goals:

There will be several design goals that will be discussed in this section. We'll start off with the player design, enemy design, in-game shopping design, dialogue system design, game manager design and lastly, the user interface design.

- **Player Design**- The main gameobject at the top of the hierarchy will contain all the physics components, along with the colliders and scripts(**called Player**) that will give the player functionality, while having a child game(**called Sprite**) object attached to the main gameobject with just the sprite and animator. Having these two isolated will not only ensure a decluttered gameobject, but it will also be easy to target specific sections of the player.

  Having the player attack enemies is a crucial part of our game, and here is the design behind the scenes. Because we created a separate child gameobject just for the sprite and animator component for the player, we can take advantage of that. As a reminder, our player is being animated with sprite sheets so we took advantage of that. We created a **hitbox** child gameobject of the sprite gameobject, attached a box collider(trigger is checked) and now as our attack animation is being played frame by frame, we can edit the collider attached to the hitbox and incorporate it within the attack animation by the click of the record button in the Animation component. This will ensure an accurate swing everytime we attack an enemy to deal damage.

  For the actual logic of the player attack system, we used something called an **interface**. This interface with the proper set up will allow us to attack and damage anything in our environment without having to uniquely target a specific enemy/object. The Interface is

basically like a contract that gets attached to an object that must use that specific function, in our case the Damage(); function. Now if the player swings his sword onto an object that has this interface implemented, that Damage(); function will be called. All we have to do is create an Attack script, attach it to the player that checks for an OnTriggerEnter2D that contains the IDamagable interface -that we created and it will call that Damage(); function.

- **Enemy Design**- The use of **inheritance and abstract classes** in our enemy design will greatly organize our logic and will avoid repeated code in our enemies. We will have an abstract class(**called Enemy**) that will contain common attributes such as speed,health,gems, Attack(), etc… We can then inherit those attributes and functionalities onto our enemies such as the spider enemy, moss enemy and skeleton enemy. This design will also allow for each enemy to have unique functionalities. Each enemy will make use of that interface stated above for the player to deal damage.

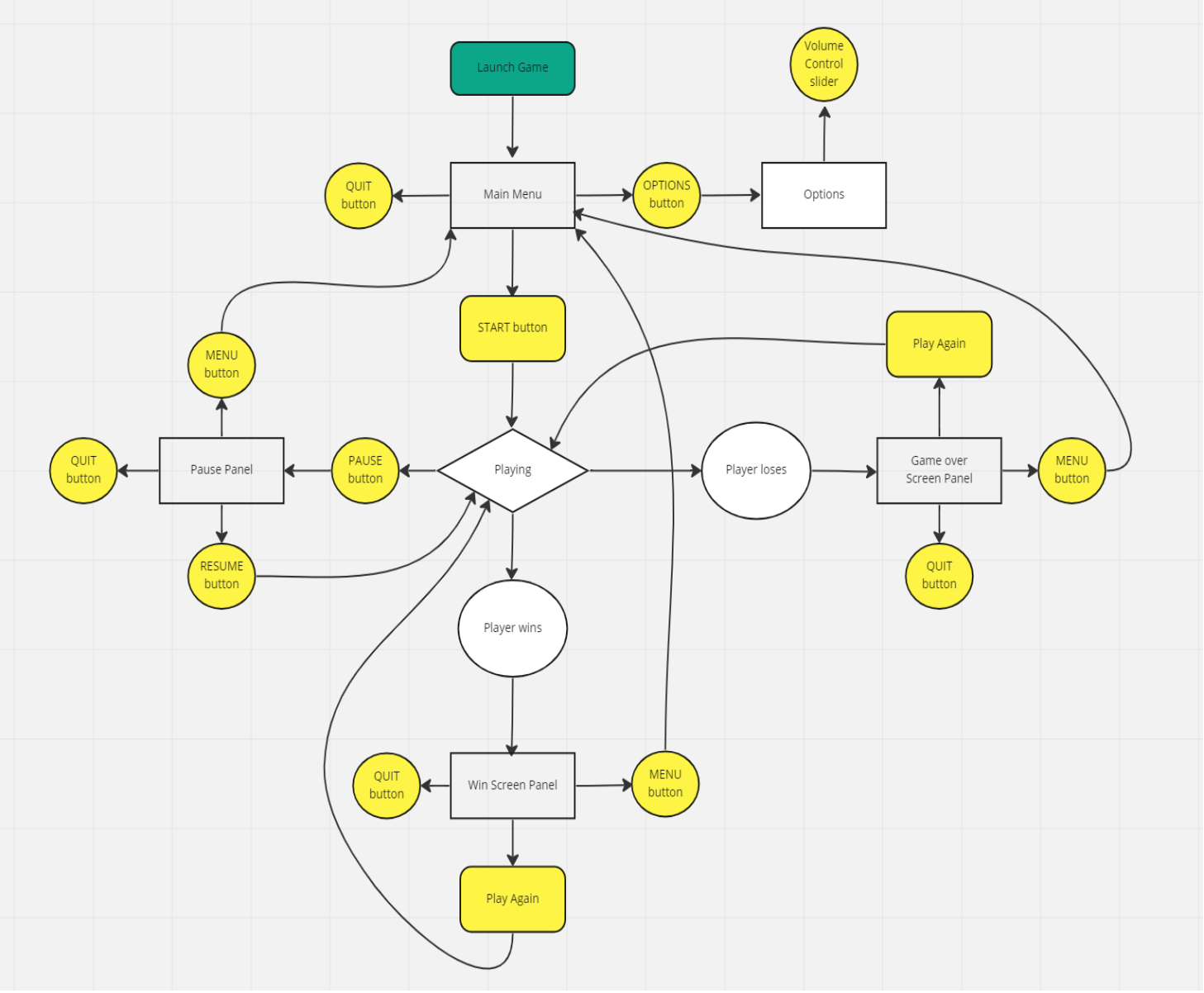
- **In-game Shopping Design-** The shopping system in our game is very straight forward. The shop itself will contain a script(**called Shop**) that will contain an OnTriggerStay2D that triggers the panel consisting of the shop UI to enable. We then have a script(**called UIManager**) that will update the gemCount, as well as the selection in terms of y-axis. The Shop script contains a SelectItem() function that will allow us to scroll through different items to purchase. Each item is initiated with an integer which is how we're keeping track of which item is what. The BuyItem() function handles the purchasing of the item which is done with a getter and setter that returns a bool.

- **Dialogue-System Design-** With some research, we found that the best way to create a dialogue system is with the use of Inky Editor. Inky is a powerful narrative scripting language that allows us to start writing, testing and integrating ink stories in Unity. We created the dialogue script within the inky editor and exported the file onto unity and incorporated it in our game. Much like the In-game shopping system, this is triggered with Unity's integrated function called OnTriggerEnter2D.

- **Game Manager Design-** The game manager is made onto a **singleton** which can be accessed by other game objects. Within this game manager, the main components are the getters and setters for the items that we purchase from the in-game shop. We have the key, boots and flame sword. We need to know if we've bought these items and returned true in order to incorporate functionality. The main point of our game is to reach the end

of the castle with the key. Having this system set up makes it easy to check for the win condition.

- **User interface-** The user interface is composed of 3 different canvas'. First one is within the main menu. Within the canvas, we have the start button, options and quit button. The options menu has its own panel which will introduce a volume slider that can be altered. Within the game scene, we have 2 'canvases'. One is for the shop panel,dialogue panel and heads up display, and the other one is for the pause menu, game over menu, and win screen menu. The script that handles the functionality is quite simple. Functions within that handle the enabling of each panel and contains the actual logic of each button which is being called through unitys On Click function within the buttons.



Project Experience

Having clean, reusable code structure and efficiency is key. A handful of interesting topics that I've come across…

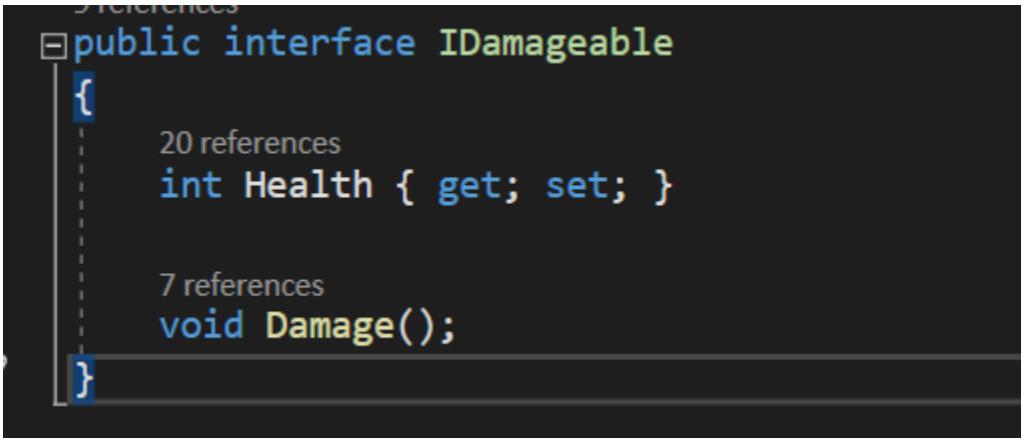
Object pooling: I've created multiple games that require instantiation of game objects but rather than reusing the game object, I would destroy them and re-instantiate a new one. This would cause major efficiency issues. Here's a snippet of code of how I structure my object pooling system.

```
1 reference
public void CreateBulletPool(int weaponType, Bullet bullet)
{
    GameObject newBullet = null;

    if(weaponType == 0)
    {
        for(int i = 0; i < initialBulletCount; i++)
        {
            newBullet = Instantiate(bullet.gameObject);
            newBullet.SetActive(false);
            newBullet.transform.SetParent(bulletHolder);

            weapon_1_BulletPool.Add(newBullet.GetComponent<Bullet>());
        }
    }
```

```
public void AddBulletToPool(int weaponType, Bullet bullet)
{
    if (weaponType == 0)
        weapon_1_BulletPool.Add(bullet);

    if (weaponType == 2)
        weapon_3_BulletPool.Add(bullet);

    if (weaponType == 3)
        weapon_4_BulletPool.Add(bullet);

    bullet.transform.SetParent(bulletHolder);
}

1 reference
public Bullet GetBullet(int weaponType)
```

```
1 reference
public void ShootBullet(float facingLeftSide)
{
    currentBullet = bulletPool.GetBullet(weaponType);

    if(currentBullet != null)
    {
        currentBullet.gameObject.transform.position = bulletSpawnPos[weaponType].position;
        currentBullet.gameObject.SetActive(true);
    }
    else
    {
        currentBullet = Instantiate(bullets[weaponType]);
        currentBullet.gameObject.transform.position = bulletSpawnPos[weaponType].position;

        bulletPool.AddBulletToPool(weaponType, currentBullet);
    }
```

Interfaces: The use of interfaces is extremely important and it completely changed the way i implement specific features such as applying damage to enemies/gameobjects in the scene. Interfaces is like a programming construct that allows you to define a contract/blueprint for a class. You can specify a set of methods, properties and events that a class implementing the interface must define and implement. Here is an example of how I used it in my game to be able to cause damage to anything in the scene with ease.



Project Experience

```
public class MossGiant : Enemy, IDamageable
{
    4 references
    public int Health { get; set; }

    private AudioManager audioManager;
    5 references
    public override void Init()...
    Unity Message | 0 references
    private void Awake()...

    3 references
    public void Damage()
    {
        isHit = true;
        Health--;
        anim.SetBool("InCombat", true);
        anim.SetTrigger("Hit");
        if (Health < 1)
```

```
public class Attack : MonoBehaviour
{
    private bool canDamage = true;



    Unity Message | 0 references
    private void OnTriggerEnter2D(Collider2D other)
    {
        IDamageable hitInfo = other.GetComponent<IDamageable>();

        if (hitInfo != null)
        {
            if (canDamage == true)
```

Project Experience

Events: Events is a feature that allows objects to communicate with each other through a publish-subscribe mechanism. They provide a way for one object(publisher) to notify other objects(subscriber) when a specific action/state change occurs. The best part is that both the subscriber and publisher are totally decoupled from each other. There are many different use cases for events. In my case I used it to update the active building type in my game. Initially I had that function running in the Update function which was very wasteful. Instead of checking 60 frames per second whether the user clicked on the UI button, I can fire an event which triggers the function in that instance.

```
private void Start() {
    BuildingManager.Instance.OnActiveBuildingTypeChanged += BuildingManager_OnActiveBuildingTypeChanged;
    UpdateActiveBuildingTypeButton();
}

1 reference
private void BuildingManager_OnActiveBuildingTypeChanged(object sender, BuildingManager.OnActiveBuildingTypeChang
    UpdateActiveBuildingTypeButton();
}

/* private void Update()
{
    UpdateActiveBuildingTypeButton();
}*/

2 references
private void UpdateActiveBuildingTypeButton() {
    arrowBtn.Find("selected").gameObject.SetActive(false);
    foreach (BuildingTypeSO buildingType in btnTransformDictionary.Keys) {
        Transform btnTransform = btnTransformDictionary[buildingType];
        btnTransform.Find("selected").gameObject.SetActive(false);
    }

    BuildingTypeSO activeBuildingType = BuildingManager.Instance.GetActiveBuildingType();
    if (activeBuildingType == null) {
```