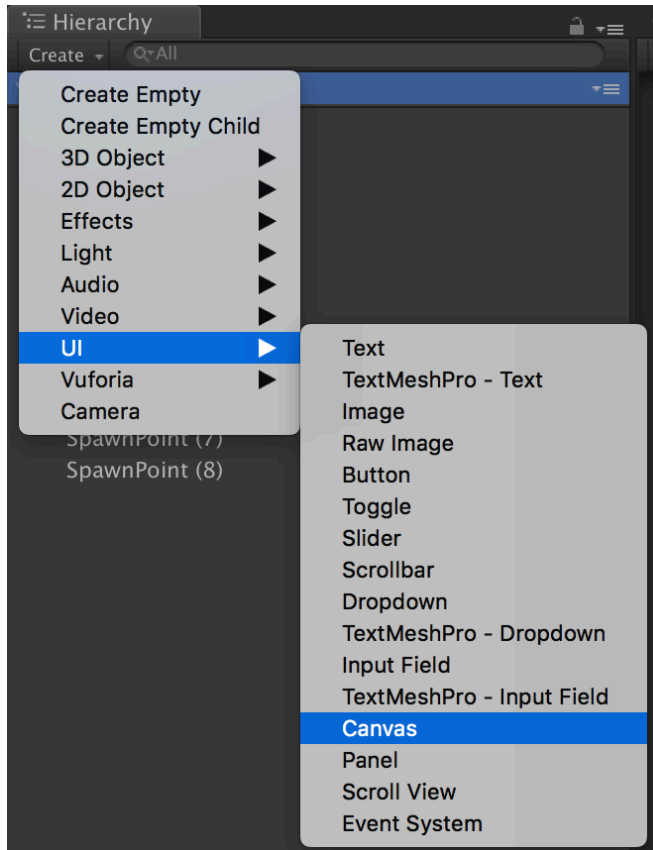


Quest 6 - Steps

1) Complete all of Part 1 (Q5)

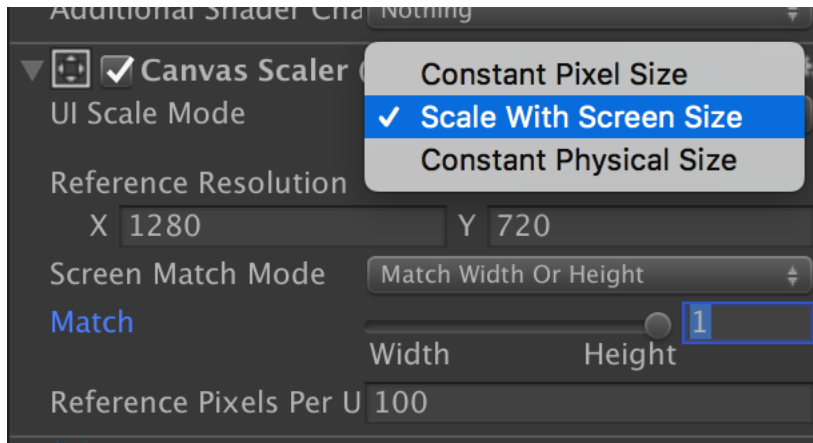
2) Create and Setup UI Canvas



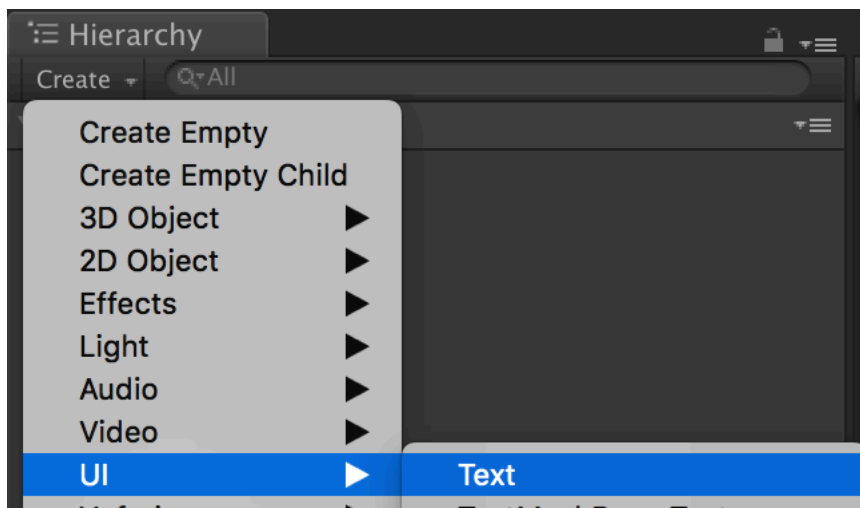
Creating a Canvas also spawns an EventSystem.



Use these settings so your Canvas is adaptive to widescreen and has a reference HD resolution.



3) Create Score Text



Text objects should automatically be created within the Canvas.



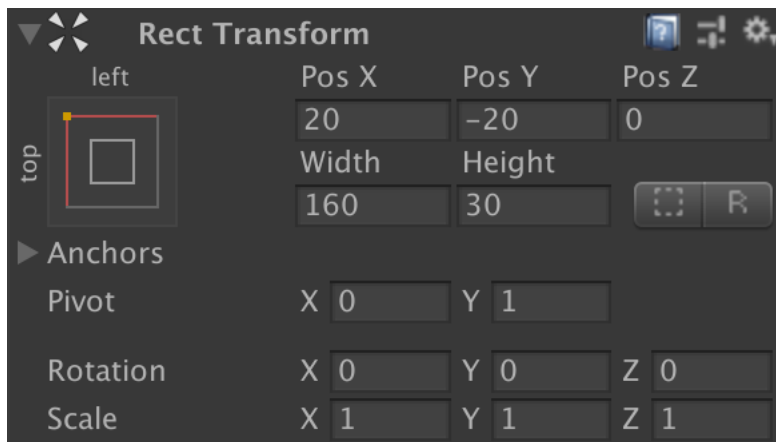
We set these Overflow settings so our Text still appears at large sizes. We don't want wrapping or truncating for this text.



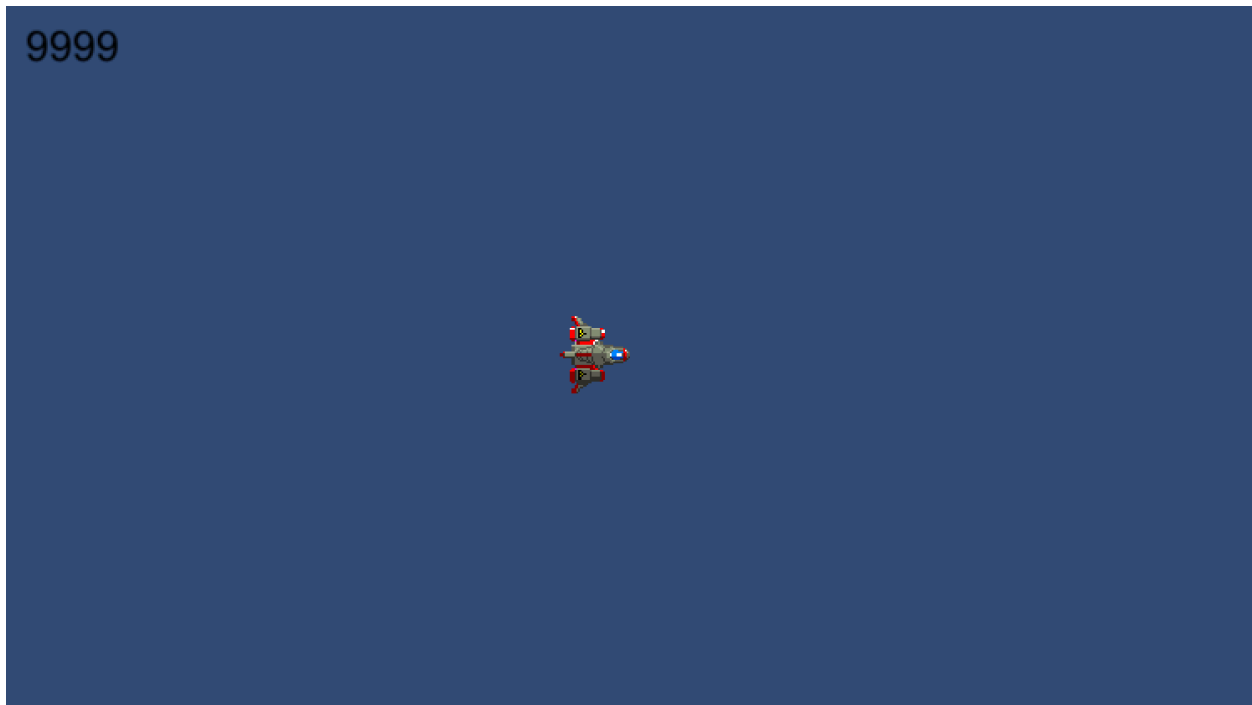
UI objects have a RectTransform instead of a Transform, which has extra settings for anchoring and alignment.



Position your Score Text in the Upper-Left by first setting the Anchor, then the Pivot, then the rest of the numbers.

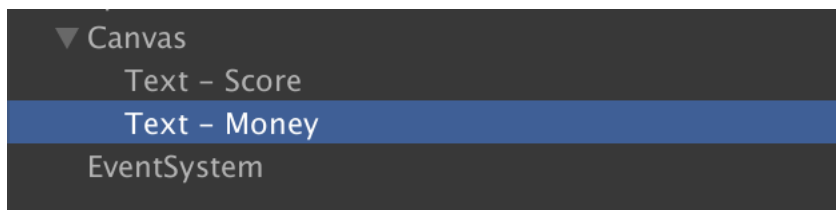


Your Score Text should be anchored Upper-Left.

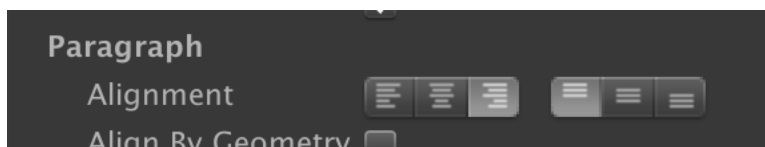


4) Create Currency Text

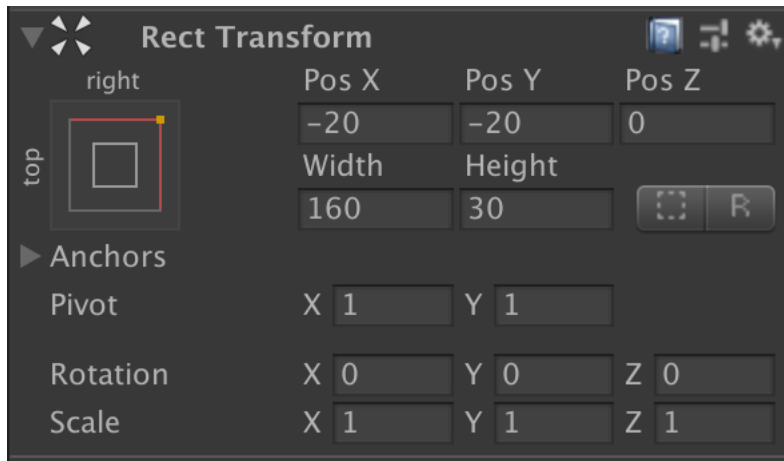
We can Duplicate and Rename our “Text - Score” as “Text - Money” to save some time with Text settings.



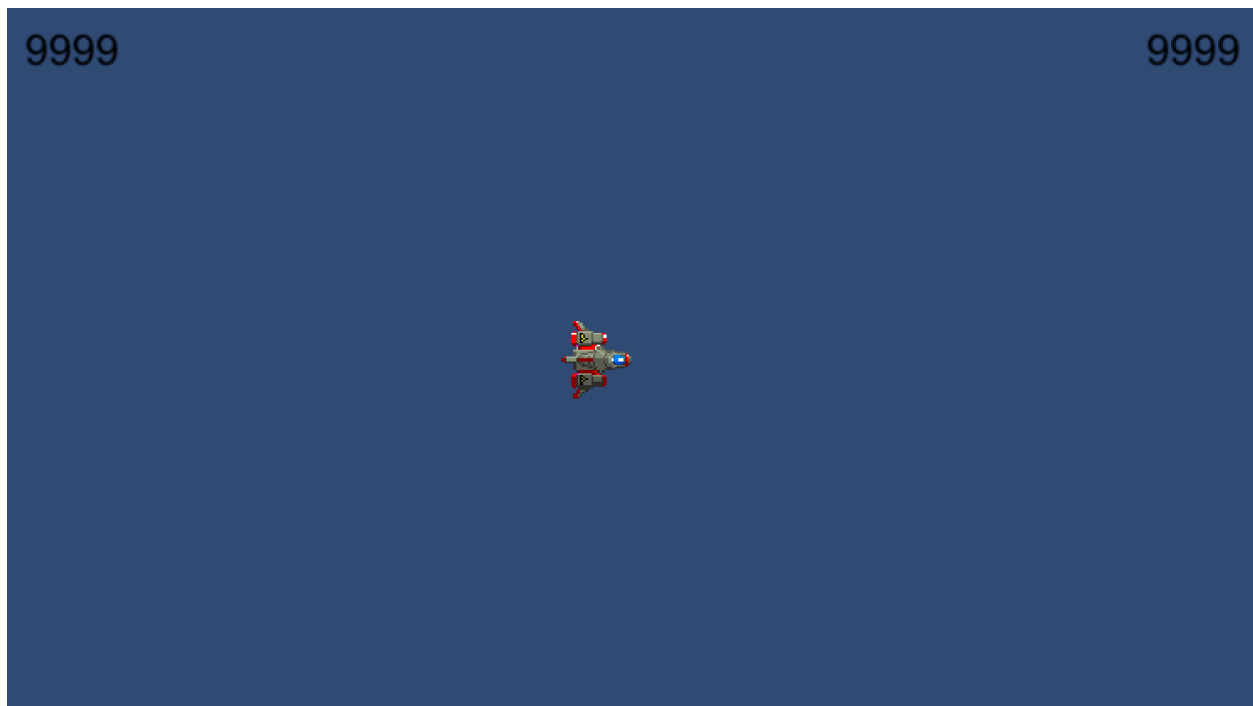
Because we want our Money to be anchored and aligned Upper-Right, we will Align our Text to the Right.



Similarly, we will use an Upper-Right Anchor, adjusted Pivot settings, and Position values in our RectTransform.

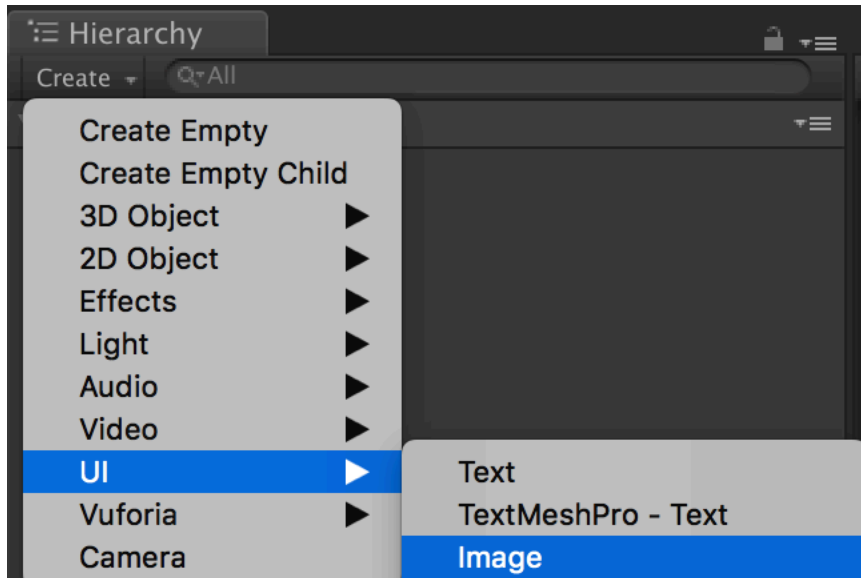


Score is Anchored Upper-Left. Money is Anchored Upper-Right.

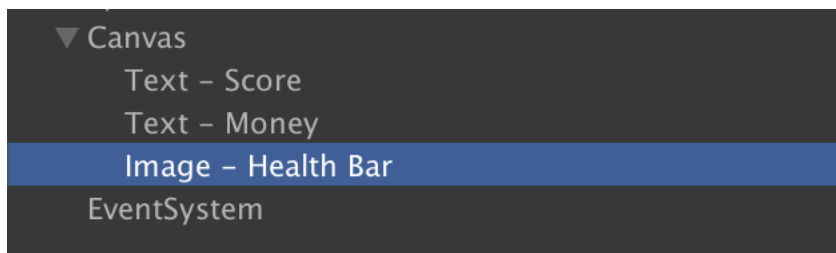


5) Create Health Bar Image

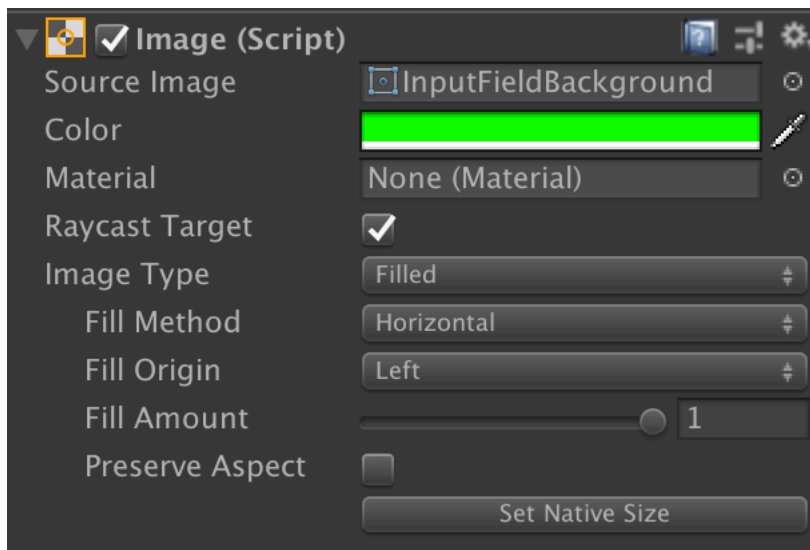
We can create Images the same way we create Text objects.



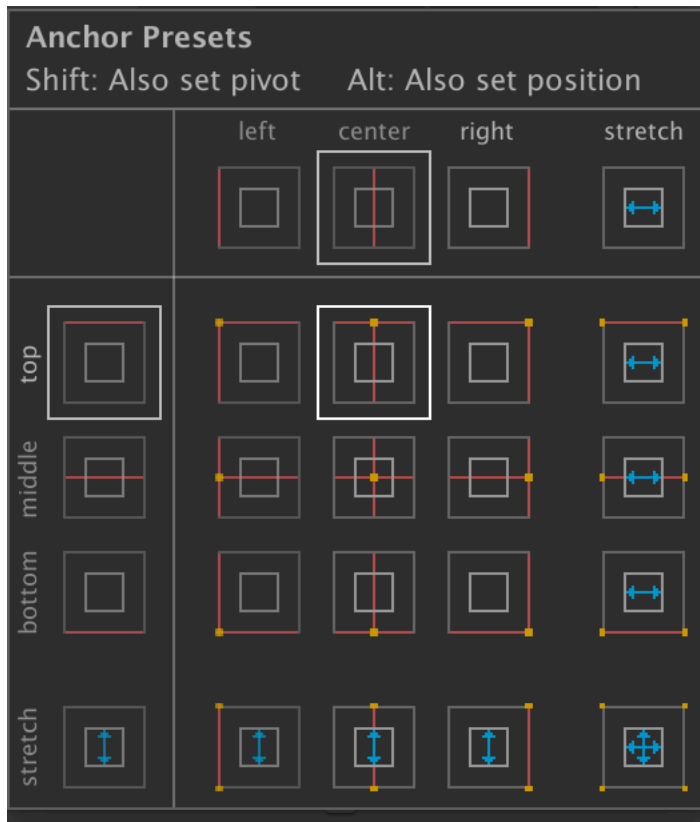
Rename our Image so we know what it is for.



Adjust the Image object settings for the look we want. Filled image types allow us to partially show portions of an image based on its fill amount. This is great for dynamic health.



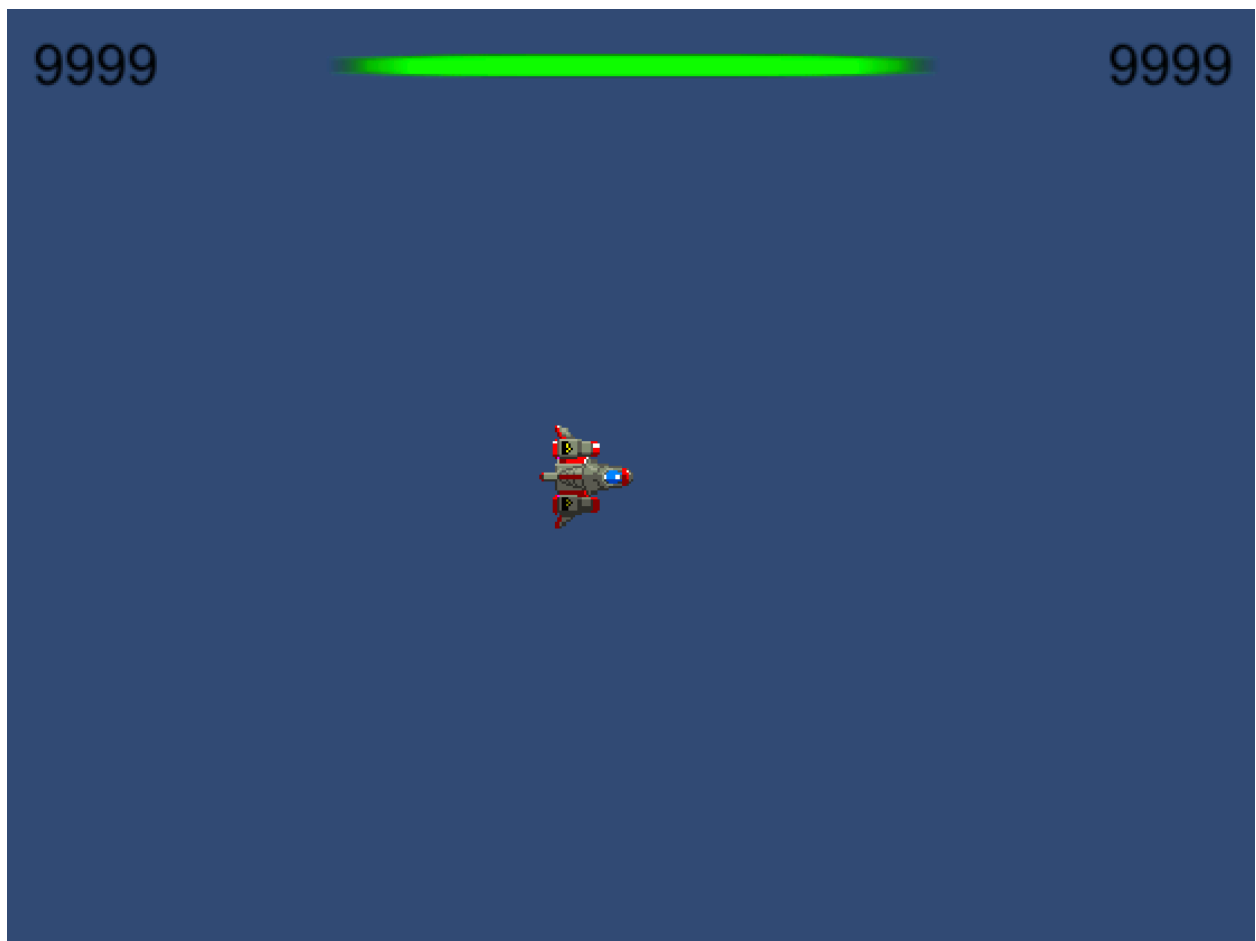
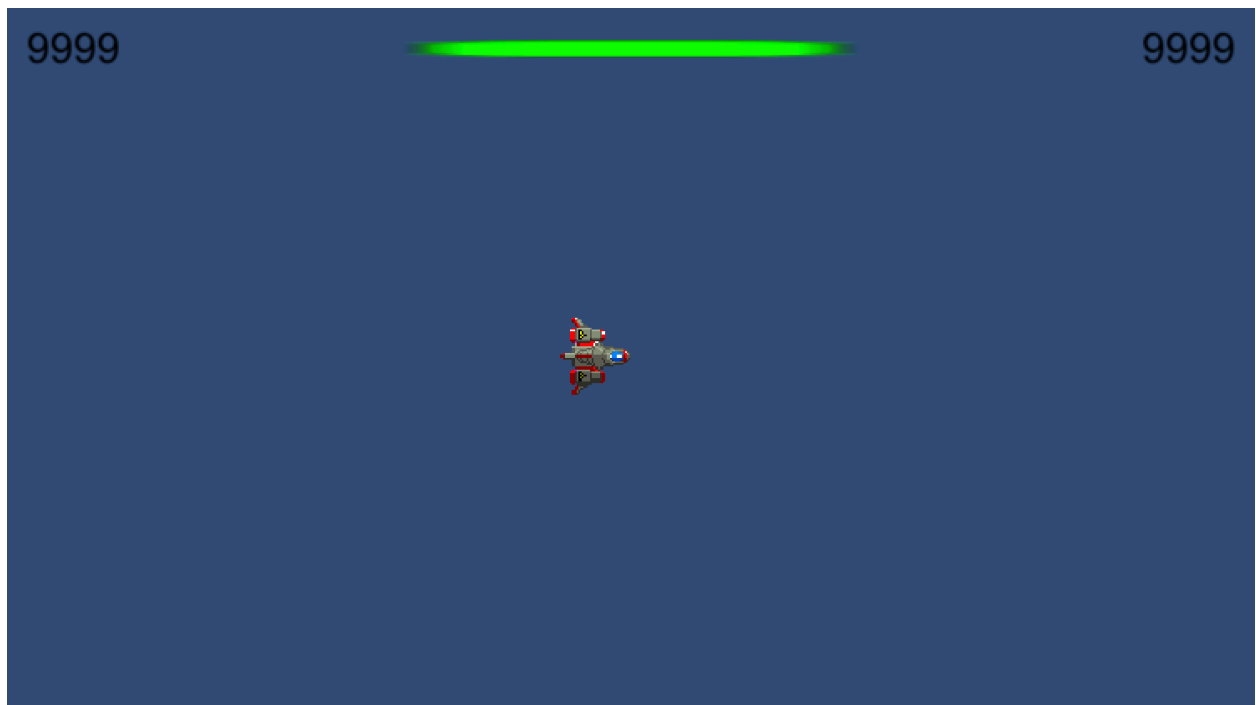
We will Anchor our health bar Top-Center.



These settings will position it Top-Center.

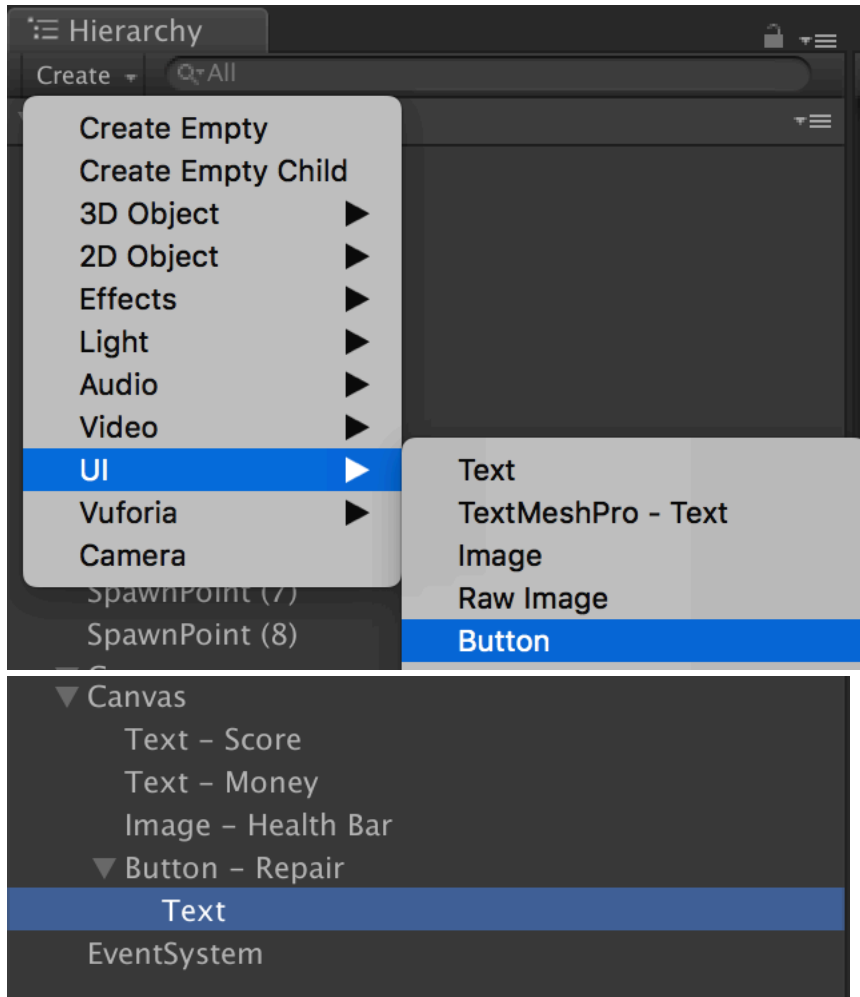


Notice how the three UI elements (Score, Money, and Health Bar) stay anchored to the Top-Left, Top-Right, and Top-Center no matter what Aspect Ratio is chosen.

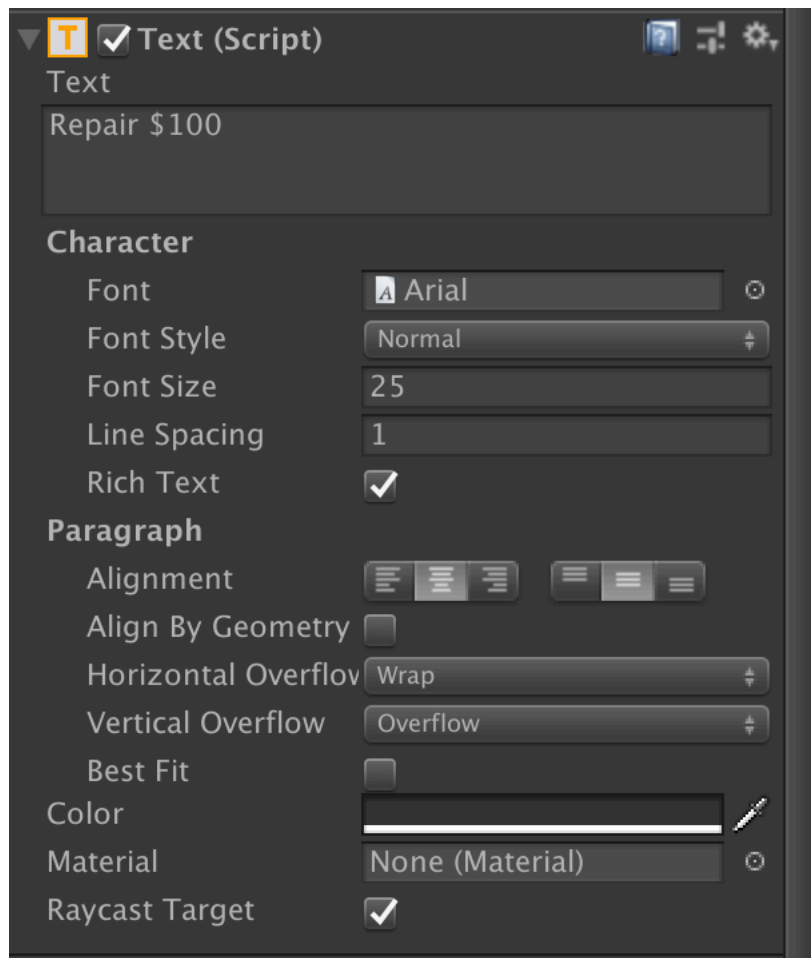


6) Create Upgrade Buttons

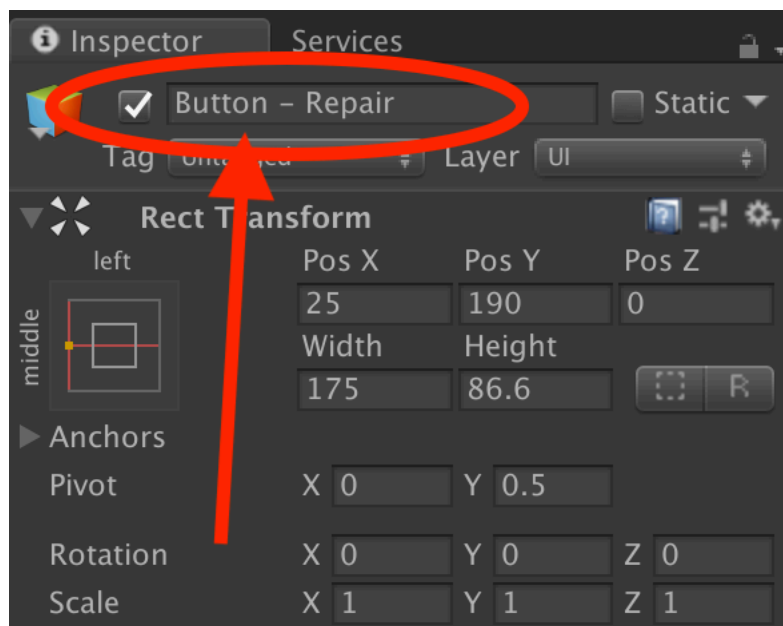
Creating Buttons works similar to other UI elements. Button GameObjects contain a Button Component and Image Component on the parent along with a Text Component in a child object. All of these play a part in customizing a Button.



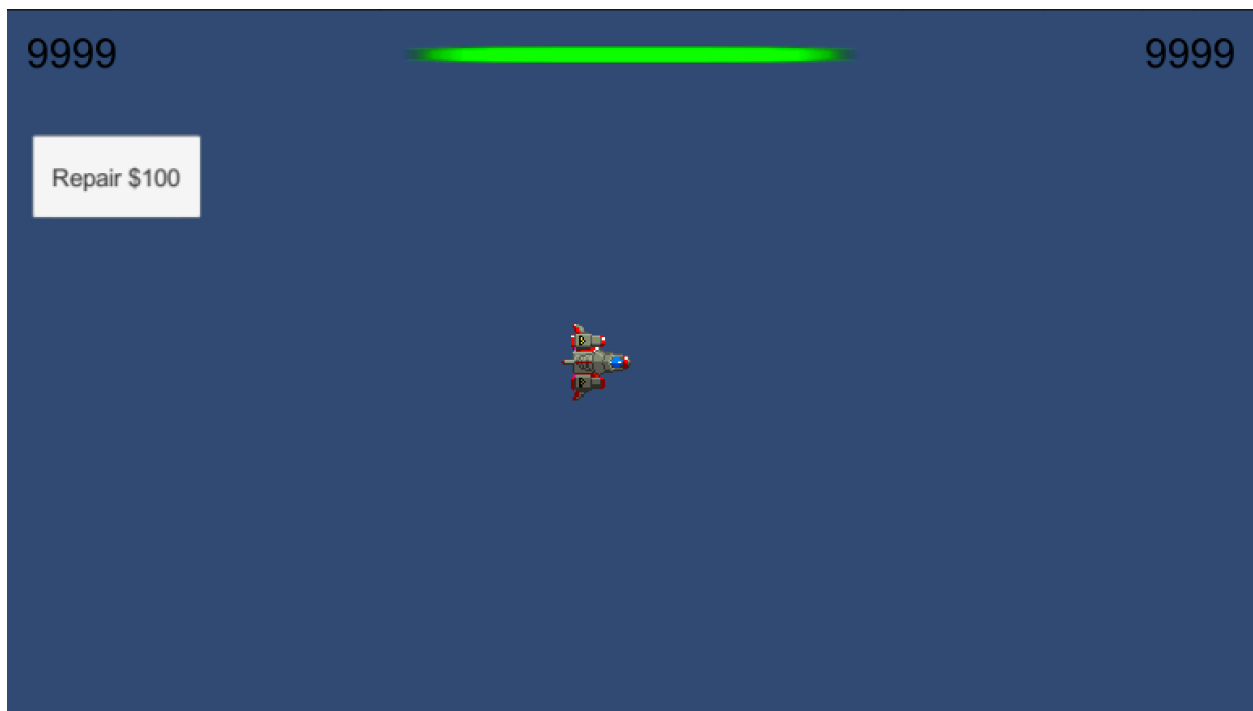
We setup our button's Text with placeholder text and formatting.



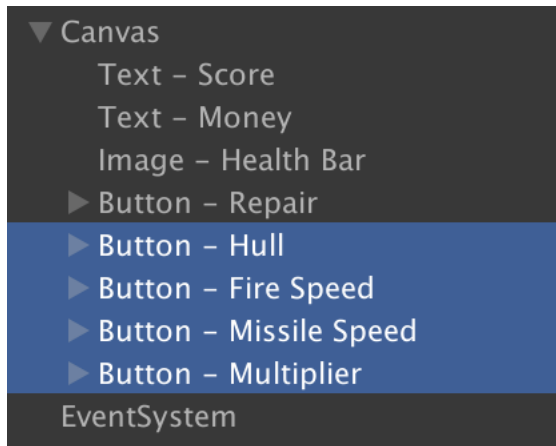
Switch to the Parent Button GameObject and position it as appropriate for our game.



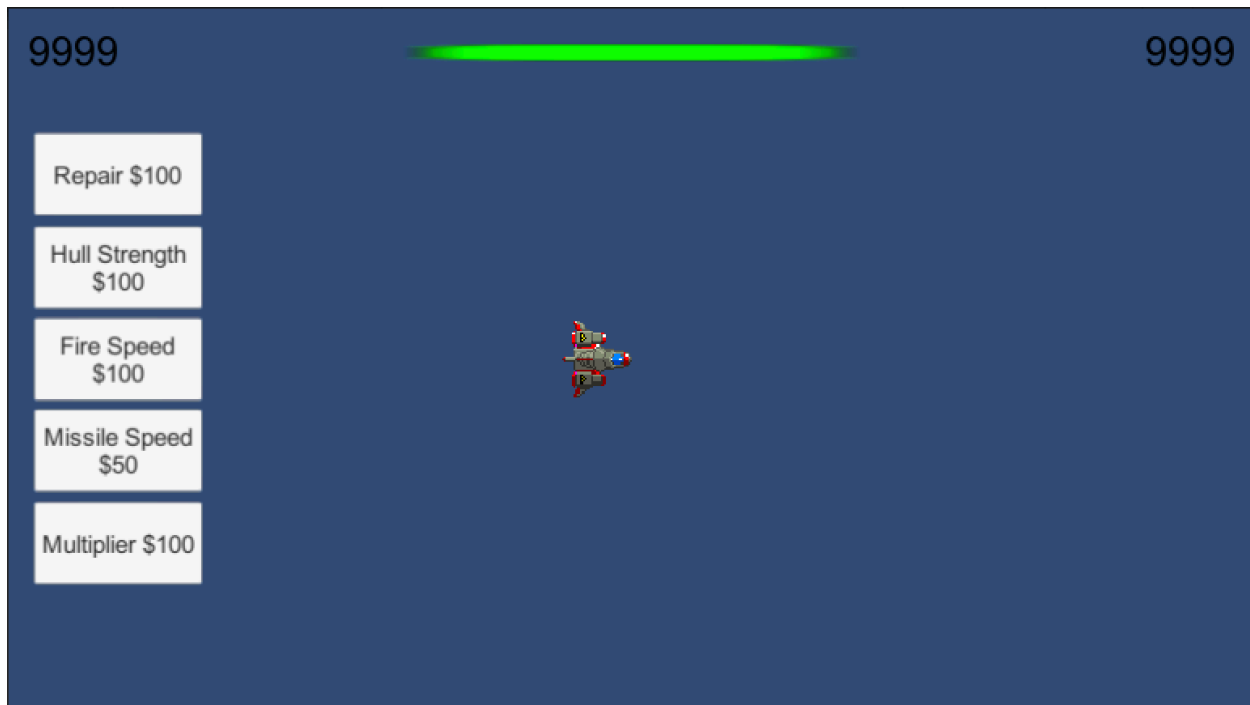
We will be using a Left-Center anchoring for all of our UI buttons.



Duplicate this process for four more buttons.



Position them accordingly. All of them should be anchored Left-Center.



Keep in mind: **These are placeholder values. The prices listed here will likely be incorrect after final game balancing and MUST be updated to reflect the true cost of upgrades.**

This completes the UI layout of our quest. The remaining steps will add dynamic text and interactivity.

7) Hook Up Health

Notice that we are in the Ship Class. You must add the UI namespace to access UI classes.

```

2 | using System.Collections.Generic;
3 | using UnityEngine;
4 | using UnityEngine.UI;
5 |
6 | public class Ship : MonoBehaviour {

```

We need an outlet to control our UI Image object. We also add properties for keeping track of health and maximum health.

```
using UnityEngine.UI;

public class Ship : MonoBehaviour {

    // Outlet
    public GameObject projectilePrefab;
    public Image imageHealthBar;

    // State Tracking
    public float firingDelay = 1f;
    public float healthMax = 100f;
    public float health = 100f;
```

We modify our existing Update event with a conditional that requires the player to have health.

```
void Update() {
    if(health > 0) {
        transform.position = new Vec
    }
}
```

A TakeDamage function will handle damage. We need to check for less-than zero health because the player can take so much damage at once that health is never exactly zero and actually goes into the negative. Fill amounts for images are on a scale of 0 to 1, which makes it really easy to use ratio calculations.

```

void TakeDamage(float damageAmount) {
    health -= damageAmount;
    if(health <= 0) {
        Die();
    }

    imageHealthBar.fillAmount = health / healthMax;
}

```

A death function will let us cleanup and transition to a dead state.

```

void Die() {
    StopCoroutine("FiringTimer");
    GetComponent<Rigidbody2D>().gravityScale = 0;
    GetComponent<Rigidbody2D>().bodyType = RigidbodyType2D.Dynamic;
}

```

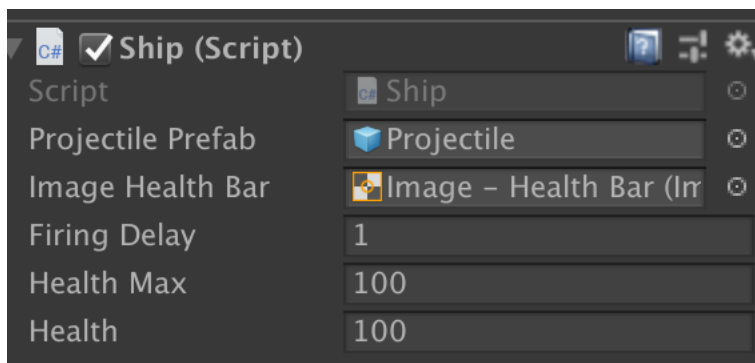
On collision we will take damage which could lead to death.

```

void OnCollisionEnter2D(Collision2D collision) {
    if(collision.gameObject.GetComponent<Asteroid>()) {
        TakeDamage(10f);
    }
}

```

We created a Public Outlet which requires us to Fill-In-The-Blank.



Notice how the health bar is only partially rendered based on our health ratio.



8) Hook Up Score

Notice that we are in the GameController class. We will add the Namespace for UI, add a Public UI Text Outlet, and a property for keeping track of score.

```
using UnityEngine;
using UnityEngine.UI;

public class GameController : MonoBehaviour {

    public static GameController instance;

    // Outlets
    public Transform[] spawnPoints;
    public GameObject[] asteroidPrefabs;
    public GameObject explosionPrefab;
    public Text textScore;

    // Configuration
    public float minAsteroidDelay = 0.2f;
    public float maxAsteroidDelay = 2f;

    // State Tracking
    public float timeElapsed;
    public float asteroidDelay;
    public int score;
```


A Points Earning Function will allow us to centralize point calculations. A dedicated Display Update function will let us update all the UI in one function.

```
public void EarnPoints(int pointAmount) {
    score += pointAmount;
}

void UpdateDisplay() {
    textScore.text = score.ToString();
}
```

We will call our UpdateDisplay function every frame so it stays up to date.

```
void Update() {
    // Increment passage
    timeElapsed += Time.deltaTime;

    // Compute Asteroid
    float decreaseDelay;
    asteroidDelay = Mathf.Max(0, asteroidDelay - decreaseDelay);

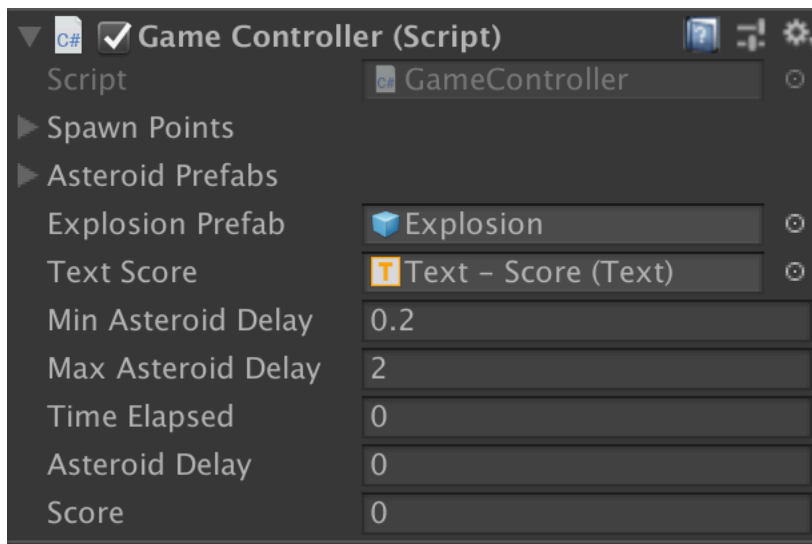
    UpdateDisplay();
}
```

Scores are reset to zero at the start

```
void Start() {
    StartCoroutine("AsteroidSpawner");

    score = 0;
}
```

Any time you make a Public Outlet, you must remember to Fill-In-The-Blank.



In your Projectile C# script, we earn points when Asteroids are destroyed.

```
void OnCollisionEnter2D(Collision2D collision) {
    if(collision.gameObject.GetComponent<Asteroid>()) {
        Destroy(collision.gameObject);
        Destroy(gameObject);

        GameObject anExplosion = Instantiate(GameContro
        Destroy(anExplosion, 0.25f);

        GameController.instance.EarnPoints(10);
    }
}
```

9) Hook Up Money

Money works almost identically to Score. It requires the same kind of Outlet and Integer Property.

```

public class GameController : MonoBehaviour {

    public static GameController instance;

    // Outlets
    public Transform[] spawnPoints;
    public GameObject[] asteroidPrefabs;
    public GameObject explosionPrefab;
    public Text textScore;
    public Text textMoney;

    // Configuration
    public float minAsteroidDelay = 0.2f;
    public float maxAsteroidDelay = 2f;

    // State Tracking
    public float timeElapsed;
    public float asteroidDelay;
    public int score;
    public int money;

```

It gets reset when score does.

```

void Start() {
    StartCoroutine("Asteroid

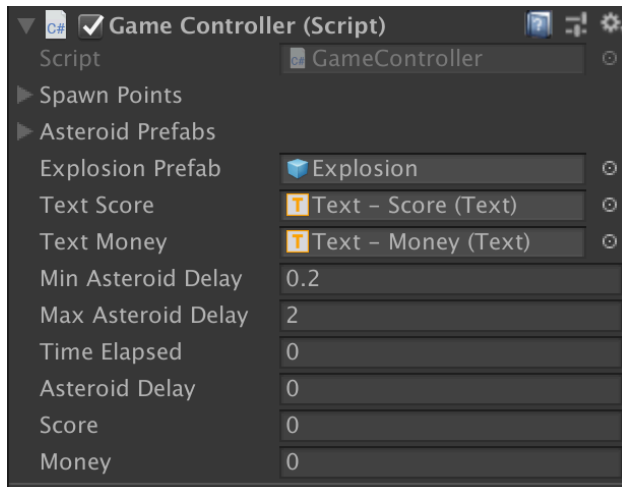
    score = 0;
    money = 0;
}

```

You earn Money at the same time and amount you earn Score.

```
public void EarnPoints(int pointAmount) {
    score += pointAmount;
    money += pointAmount;
}
```

Fill in the outlet for Money.



We must also update the UI for money the same way we do for score.

```
void UpdateDisplay() {
    textScore.text = score.ToString();
    textMoney.text = money.ToString();
}
```

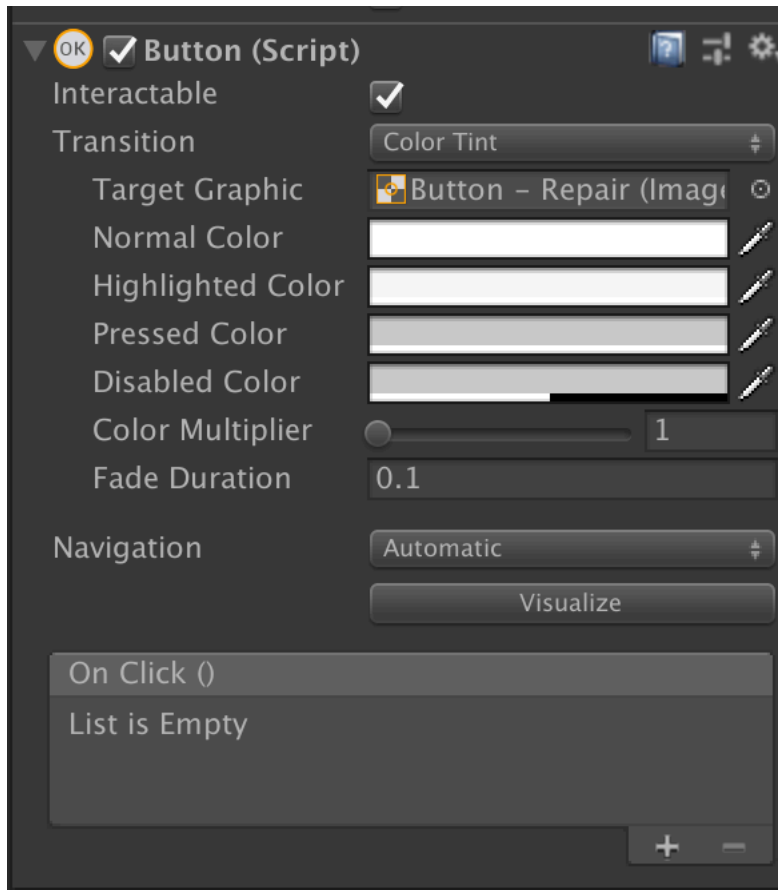
10) Hook Up Repair Upgrade

In your Ship class, we will create a function for Hull Repairs.

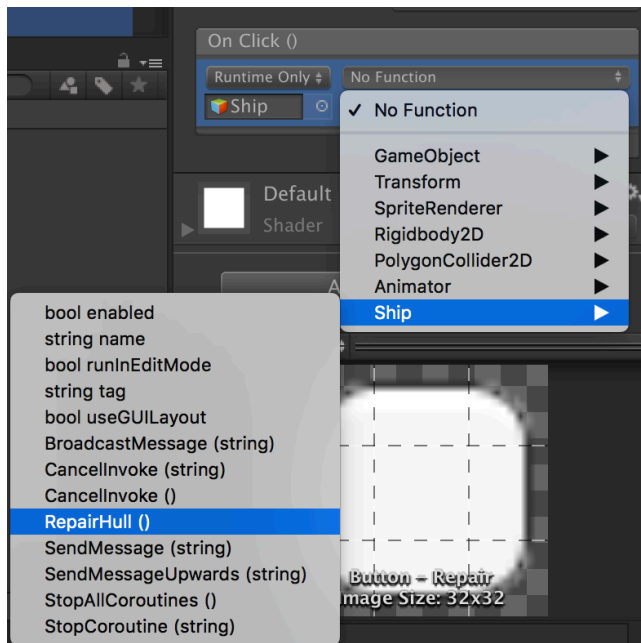
```
public void RepairHull() {
    int cost = 100;
    if(health < healthMax && GameController.instance.money >= cost) {
        GameController.instance.money -= cost;

        health = healthMax;
        imageHealthBar.fillAmount = health / healthMax;
    }
}
```

Buttons have a lot of settings for aesthetics along with a section for setting up click interactivity.



Click the + to add a Click Event. Drag the Object containing your Function into the blank. In our case, the Ship has the Function for Repairing Hull. From the Drop-Down, select the Component and finally the Function you would like the button to trigger.



11) Hook Up Hull Strength Upgrade

The Hull Strength Upgrade Button is unique in that the Button Text changes every time we use the upgrade. To alter the Text, we need to create a Public Text Outlet to reference it in code.

```
public class Ship : MonoBehaviour {

    // Outlet
    public GameObject projectilePrefab;
    public Image imageHealthBar;
    public Text hullUpgradeText;
```

Our UpgradeHull Function shares a lot of similarities with the RepairHull function. Both calculate a cost, check if enough money is available, subtract the money from our wallet, and grant the player some special stat.

UpgradeHull is unique in that we also update the Button Text to reflect the cost of the next upgrade purchase.

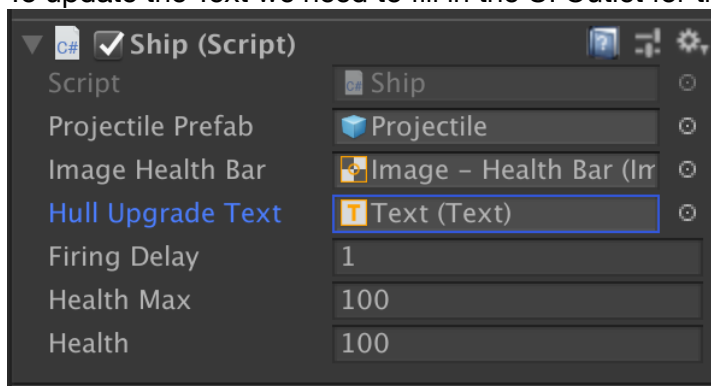
```
public void UpgradeHull() {
    int cost = Mathf.RoundToInt(healthMax);

    if(GameController.instance.money >= cost) {
        GameController.instance.money -= cost;

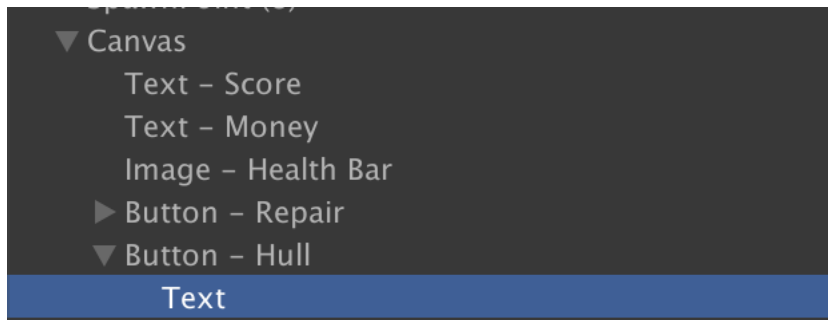
        health += 50;
        healthMax += 50;
        imageHealthBar.fillAmount = health / healthMax;

        hullUpgradeText.text = "Hull Strength $" + Mathf.RoundToInt(healthMax).ToString();
    }
}
```

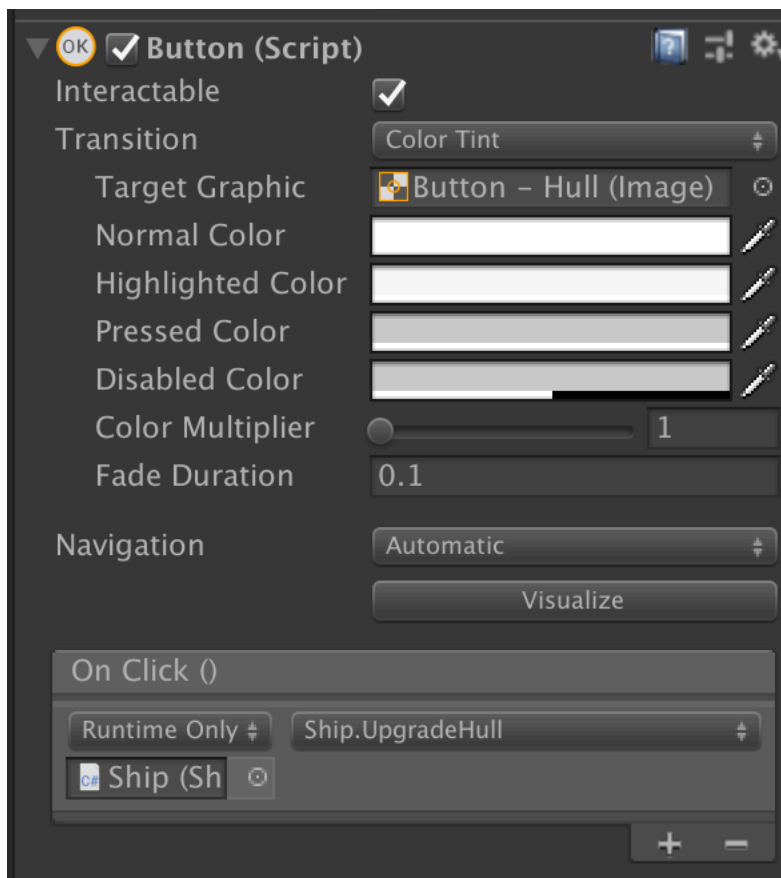
To update the Text we need to fill in the UI Outlet for the Text we are updating.



Be sure to drag the correct Button Text object into the blank.



We also need to hook up the Button Event the same way we did for the previous button.



12) Hook Up Fire Speed Upgrade

Just as before, create your Public UI Text Outlet.

```
public class Ship : MonoBehaviour {

    // Outlet
    public GameObject projectilePrefab;
    public Image imageHealthBar;
    public Text hullUpgradeText;
    public Text fireSpeedUpgradeText;
```

Create your Upgrade Function.

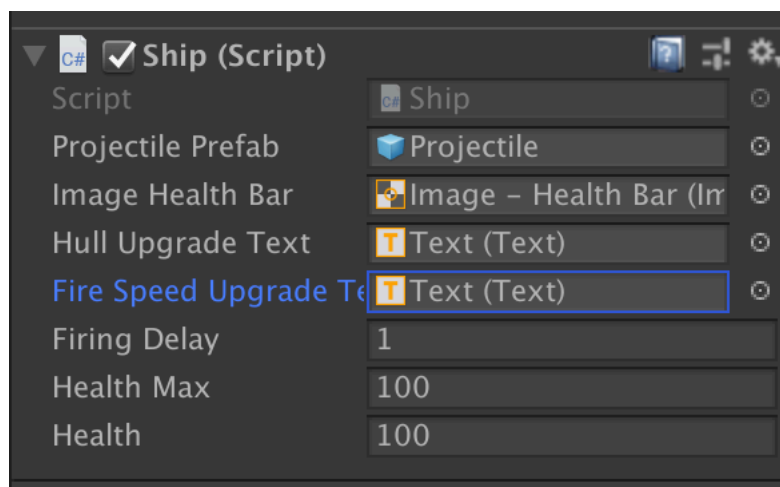
```
public void UpgradeFireSpeed() {
    int cost = 100 + Mathf.RoundToInt((1f - firingDelay) * 100f);

    if(GameController.instance.money >= cost) {
        GameController.instance.money -= cost;

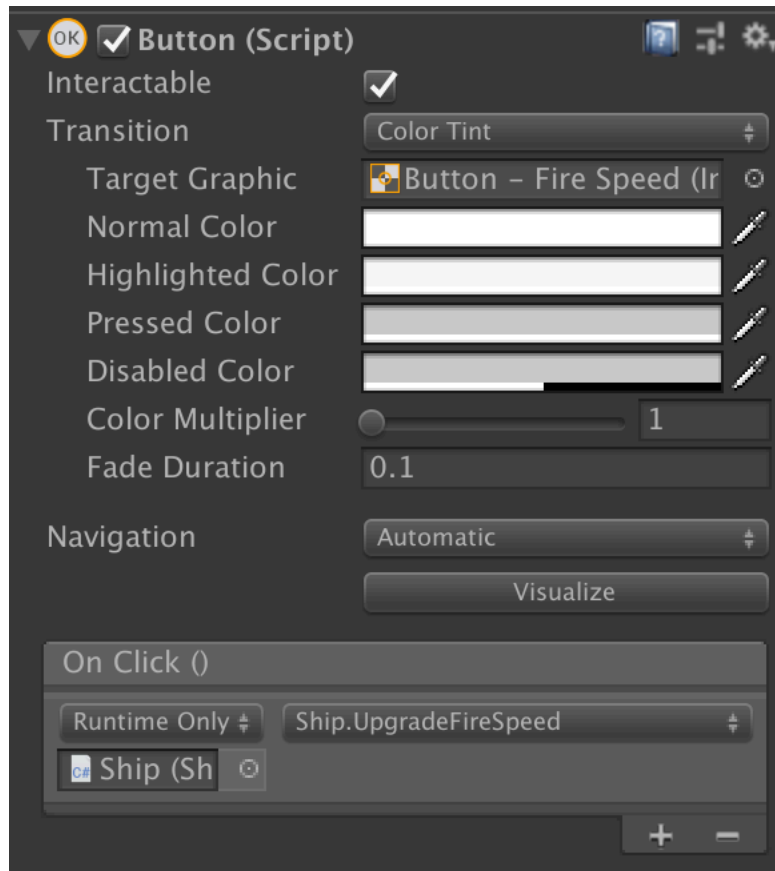
        firingDelay -= 0.05f;

        int newCost = 100 + Mathf.RoundToInt((1f - firingDelay) * 100f);
        fireSpeedUpgradeText.text = "Fire Speed $" + newCost.ToString();
    }
}
```

Fill in the Outlet.



Setup the Button Event.



13) Hook Up Missile Speed Upgrade

The next two upgrades work the same but are in a different class. We put these in the GameController class because the GameController is easily accessible by other Game Objects.

We create a Public UI Text Outlet so our Button Text can be dynamic.

A Missile Speed property will increase as we upgrade it.

```

public class GameController : MonoBehaviour {

    public static GameController instance;

    // Outlets
    public Transform[] spawnPoints;
    public GameObject[] asteroidPrefabs;
    public GameObject explosionPrefab;
    public Text textScore;
    public Text textMoney;
    public Text missileSpeedUpgradeText;

    // Configuration
    public float minAsteroidDelay = 0.2f;
    public float maxAsteroidDelay = 2f;

    // State Tracking
    public float timeElapsed;
    public float asteroidDelay;
    public int score;
    public int money;
    public float missileSpeed = 2f;

```

In our Projectile script, we will update the variables to make use of our new GameController property.

```

void Update() {
    // Some dynamic projectile attributes
    float acceleration = GameController.instance.missileSpeed / 2f;
    float maxSpeed = GameController.instance.missileSpeed;

```

Back in GameController, we create an Upgrade Function similar to all of our other Upgrade Functions.

```

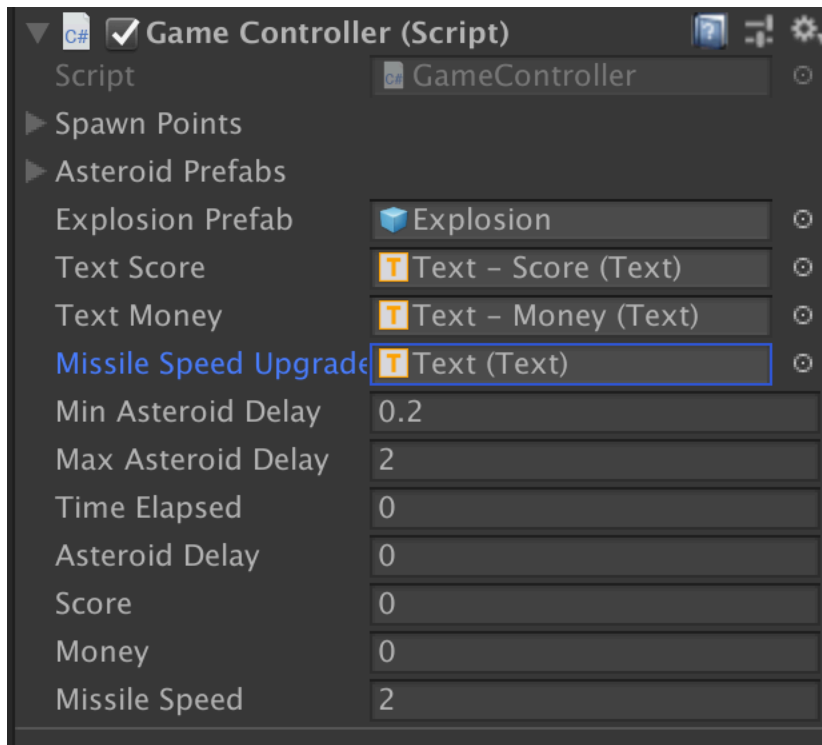
public void UpgradeMissileSpeed() {
    int cost = Mathf.RoundToInt(25 * missileSpeed);
    if(cost <= money) {
        money -= cost;

        missileSpeed += 1f;

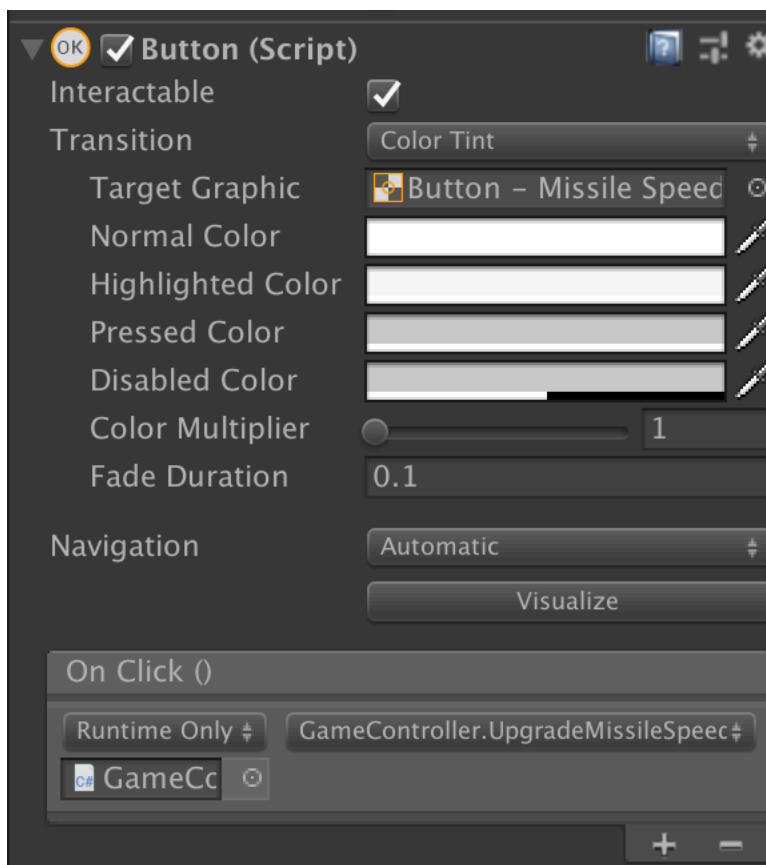
        missileSpeedUpgradeText.text = "Missile Speed $" + Mathf.RoundToInt(25 * missileSpeed).ToString();
    }
}

```

Fill in the Public UI Outlet blank.



Setup the Button Event. We are talking to the GameController object this time instead of Ship.



14) Hook Up Score Multiplier Upgrade

Similar to before, we create a Public UI Text Outlet and a number Property to keep track of our upgrade stat.

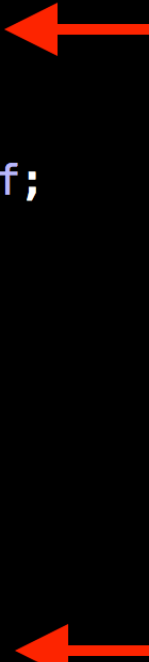
```
public class GameController : MonoBehaviour {

    public static GameController instance;

    // Outlets
    public Transform[] spawnPoints;
    public GameObject[] asteroidPrefabs;
    public GameObject explosionPrefab;
    public Text textScore;
    public Text textMoney;
    public Text missileSpeedUpgradeText;
    public Text bonusUpgradeText;

    // Configuration
    public float minAsteroidDelay = 0.2f;
    public float maxAsteroidDelay = 2f;

    // State Tracking
    public float timeElapsed;
    public float asteroidDelay;
    public int score;
    public int money;
    public float missileSpeed = 2f;
    public float bonusMultiplier = 1f;
```



We modify our EarnPoints function to make use of this Bonus Multiplier stat. Now we earn more points the more we upgrade.

```
public void EarnPoints(int pointAmount) {
    score += Mathf.RoundToInt(pointAmount * bonusMultiplier);
    money += Mathf.RoundToInt(pointAmount * bonusMultiplier);
}
```

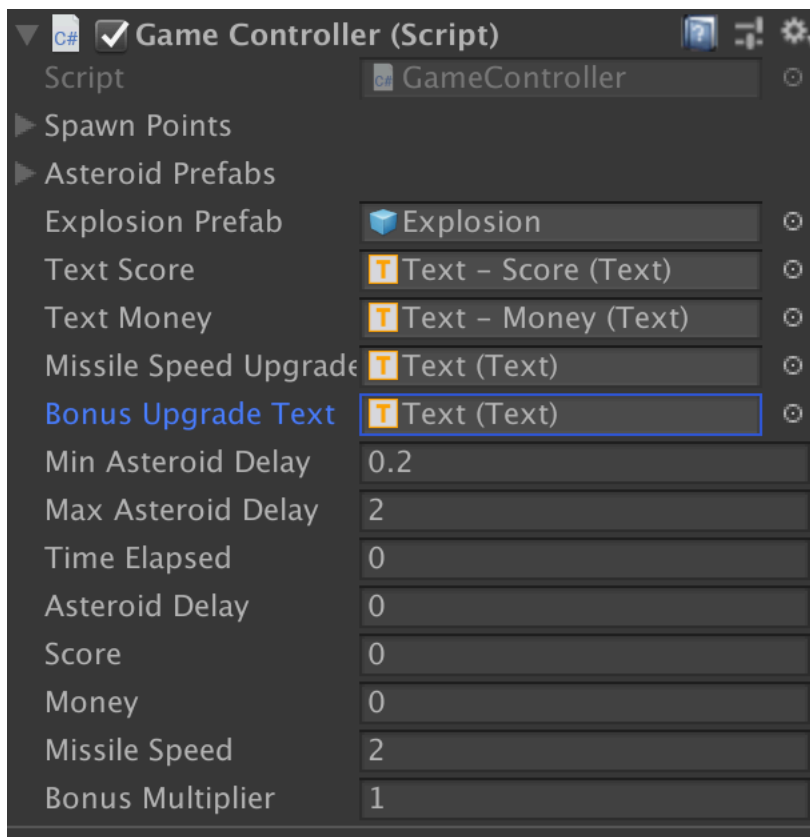
An UpgradeBonus Function mirrors the other Upgrade Functions.

```
public void UpgradeBonus() {
    int cost = Mathf.RoundToInt(100 * bonusMultiplier);
    if(cost <= money) {
        money -= cost;

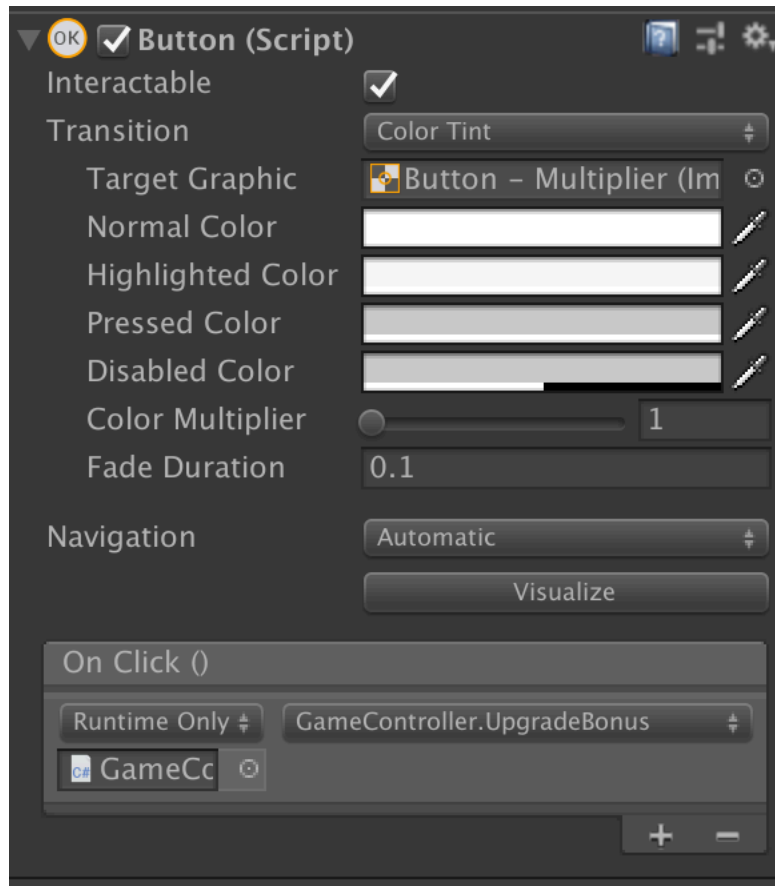
        bonusMultiplier += 1f;

        bonusUpgradeText.text = "Multiplier $" + Mathf.RoundToInt(100 * bonusMultiplier).ToString();
    }
}
```

Fill in the blank for the Outlet.



Finally, setup the Button Event.



15) Playtest and Balance for Game Feel

Try adjusting your upgrade bonuses and upgrade costs for a more balanced Game Feel.

16) Finalize UI to match game balance

When we first design the UI, we often use placeholder text because we don't always know the final balanced game values. Once any tweaks to your upgrade equations are finalized, ensure that the default button text matches the starting costs for any upgrades. **Do NOT leave them at their incorrect placeholder values.**

