

Coin Pusher Pro Asset - README

Created by Alan O'Toole

This README document includes helpful hints, tutorials, and a description of how the scripts work together.

If you have any questions or comments please let us know by emailing us at:
Alan@AlanOTOole.com

Thank you!

How do I enable Unity Ads for Coin Pusher Pro?

Coin Pusher Pro supports Unity Ads right out of the box! You will find a complete **AdManager.cs** file that handles all sorts of things automatically for you when it comes to rewarding your players with coins and more.

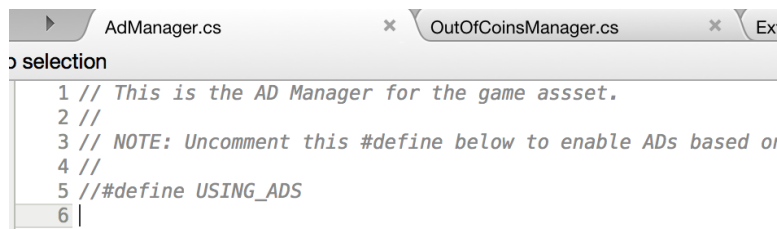
Note: You will need to enable the Unity Ads before the **AdManager** will work!

Here is how:

1. Open the Ads configuration window from "Window -> Services -> Ads".
2. Turn on Ads and answer the questions from Unity.
3. You're done!

Once you enabled the Unity Ads above, open up **AdManager.cs**.

At the top of the file, you will see a commented out `///define USING_ADS`. Uncomment that line by removing the `///` and you're done! The **AdManager** is now enabled and running.



```
AdManager.cs x OutOfCoinsManager.cs x Ex
> selection
1 // This is the AD Manager for the game asset.
2 //
3 // NOTE: Uncomment this #define below to enable ADs based on
4 //
5 // #define USING_ADS
6 |
```

For more information on Unity Ads, check out the manual here:
<http://docs.unity3d.com/Manual/UnityAdsHowTo.html>

How to use the IAP Manager?

Note: The **IAP Manager** that comes built into Coin Pusher Pro utilizes the Unity services for In App Purchases and Analytics. You can find more about what Unity has by following this link: <https://unity3d.com/services/analytics/iap>.

You will need to make sure that you enabled the Unity Services In App Purchasing before you can continue using the **IAP Manager**.

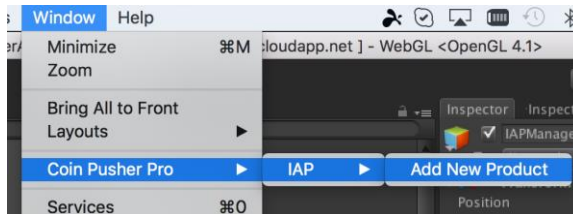
The **IAP Manager** inside of Coin Pusher Pro handles everything to allow you to easily add your own IAP products quickly. You simply define a new product, drop it in, and you're done.

This is an overview of how the **IAP Manager** process works:

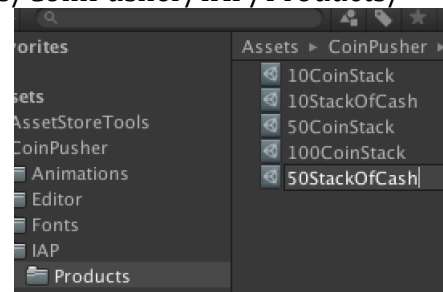
1. Create a **IAPProduct** that you want to sell
2. Fill out the **IAPProduct** information. This includes an internal ID and the IDs you use on the stores (typically these are reverse domain format: com.testdomain.productname)
3. Add this product to the **IAP Manager**
4. Add the **IAPProduct** to a button click using the **buyProduct(IAPProduct product)** function included in the **IAPManager**.

How to add a new IAP Product?

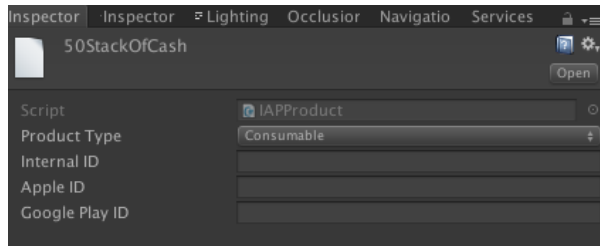
1. Go to **Window -> Coin Pusher Pro -> IAP -> Add New Product**



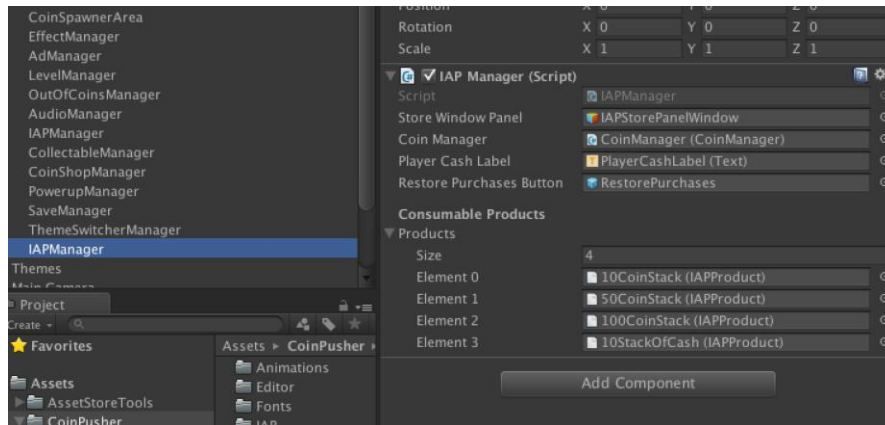
- a.
2. This will create a new **IAPProduct** inside of the folder "Assets/CoinPusher/IAP/Products/"



- a.
- b. Rename this new **IAPProduct** to something unique like above.
3. Fill in the details on the new product on the Inspector



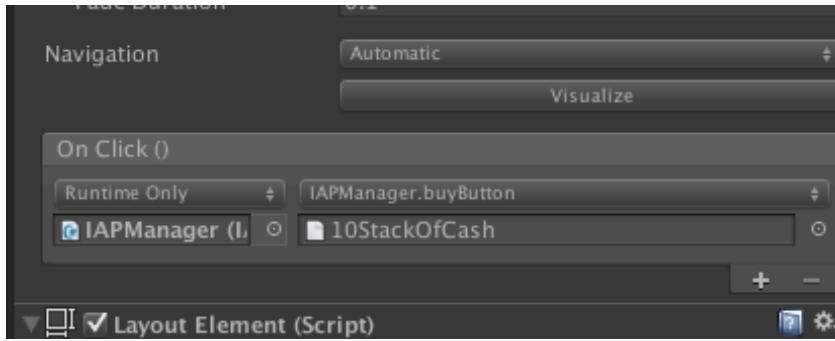
- a.
 - b. The details you use here will reflect the unique ID you set up in both your Apple portal and Google Play portal for this product. Typically this is reverse domain format.
 - c. You can also change the type of Product by changing the drop down. Consumable, Non Consumable, or Subscription.
 - d. The **Internal ID** needs to be a unique string; Coin Pusher Pro only uses this.
4. In the project hierarchy, expand the **CoinManager** and look for **IAPManager**.
 5. Drag and drop your new **IAPProduct** you created onto the array of existing products



- a.
 - b. You can also expand this array manually and use the selector circle to add it. The choice is yours.
6. You are all done! Your product is ready to be purchased.

How to setup the new IAPProduct for purchasing?

In this example, we are setting up a UI Button's **onClick** to call the function in **IAPManager buyButton(IAPProduct product)** passing it the **IAPProduct** that you just made.



From the above screenshot, you can see that we dragged and dropped **10StackOfCash** onto the exposed variable field for the method **IAPManager.buyButton(...)**.

Once you have that set up, you're all done! When the user clicks this button, the **IAPManager** will take care of the rest for you.

You can also call **IAPManager.buyButton(IAPProduct product)** or **IAPManager.buyButton(string product)** from code to give you more control on how purchasing is performed.

How do I define what happens to when they buy the IAPProduct?

Once the player purchases your new product, you will then need to actually provide them the content they paid for. This is handled in the function **public PurchaseProcessingResult ProcessPurchase (PurchaseEventArgs args)** inside of **IAPManager.cs**.

This function will be called whenever a user either:

- Purchases a new product
- Restores the previous purchases
 - Note: On the Apple platforms, the user needs to manually hit the button that processes the previous purchases. Any other platforms, like Google Play, this process happens automatically when the app or game is installed.

```

// Summary
public PurchaseProcessingResult ProcessPurchase (PurchaseEventArgs args)
{
    // We know need to give the user whatever they bought, we compare against the product.internalID on the asset in IAP/Products
    switch(args.purchasedProduct.definition.id)
    {
        case "10CoinStack":
            // Now provide the player the amount of coins they bought!
            coinManager.addCoin(10, true);
            break;

        case "50CoinStack":
            // Now provide the player the amount of coins they bought!
            coinManager.addCoin(50, true);
            break;

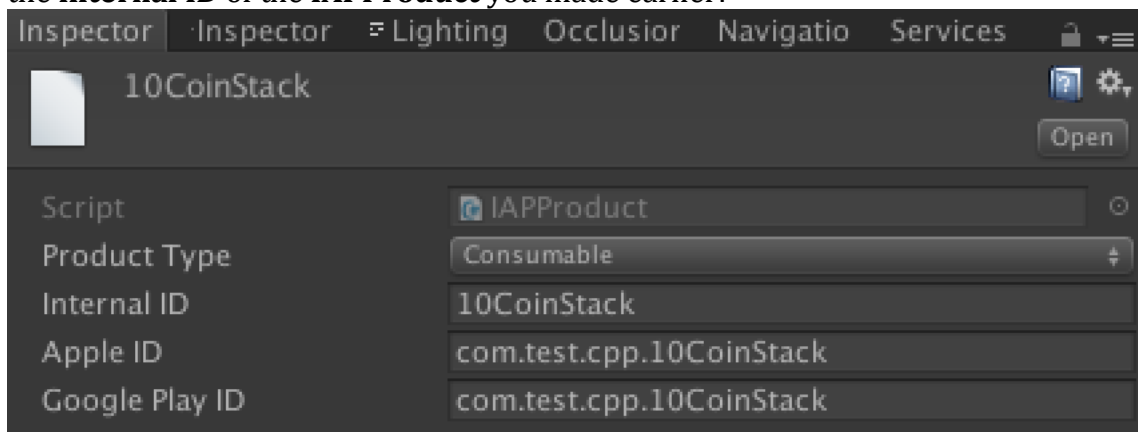
        case "100CoinStack":
            // Now provide the player the amount of coins they bought!
            coinManager.addCoin(100, true);
            break;

        case "10StackOfCash":
            // Now provide the player the amount of cash they bought!
            coinManager.addCash(10);
            break;
    }

    // Here is where we would set up any products that were non consumables
    switch(args.purchasedProduct.definition.id)
    {
        case "ValentineLevel":
            Debug.Log("Just purchased or restored the valentine level. You would either give them the level now or restore it since t
            break;
    }
}

```

In the above screenshot, you will see two sets of **switch** statements. The first switch statement handles the consumable objects while the second handles a non consumable product. Inside of each **case** statement, each string used needs to match the **internal ID** of the **IAPProduct** you made earlier:



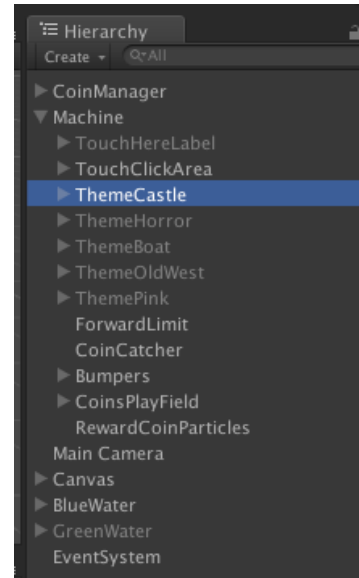
When these two match up in this function, the product is either just purchased or being restored.

Inside of that each **case** statement you would then write your code to provide the user with the item they purchased.

How do I change the machine skin (or theme)?

The Coin Pusher Pro Asset includes several pre-loaded themes. These themes are labeled as “Theme<TYPE>”, for example, **ThemeCastle** and **ThemeHorror**.

To change to a different theme, disable the current theme by disabling the current machine game object and enabling a new one.

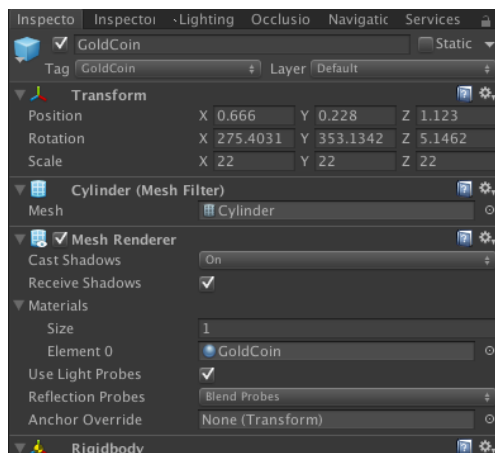


How do I make a new coin?

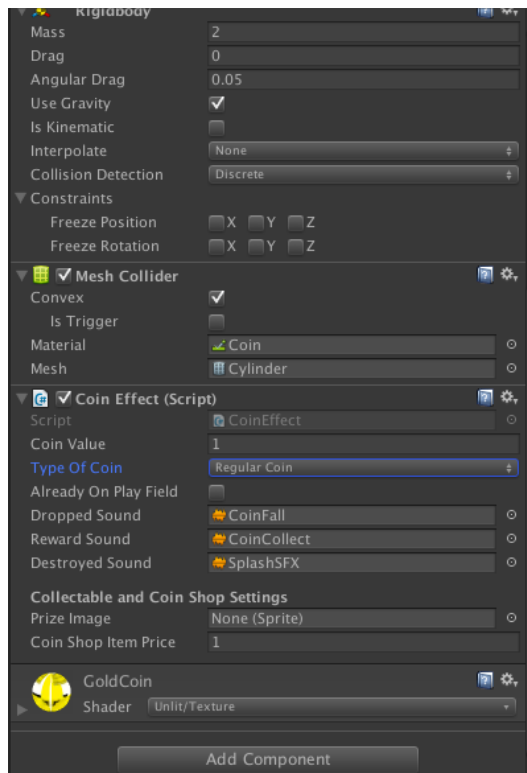
1. Start with “Duplicate **GoldCoin**”.



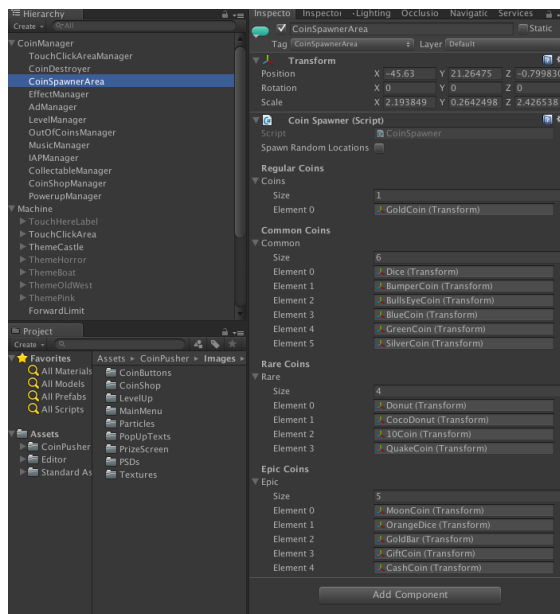
2. Assign a new material to the new coin.



3. Edit the **CoinEffect** properties.

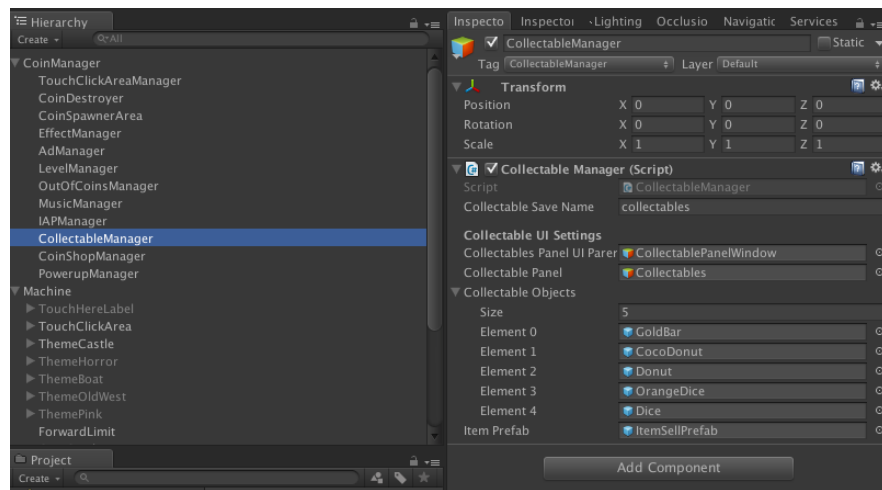


4. Assign the new coin to the **CoinSpawnerArea** under its appropriate group type. The group type, which dictates the frequency of which each coin is spawned, can be regular, common, rare, or epic.



How do I make a new collectable item?

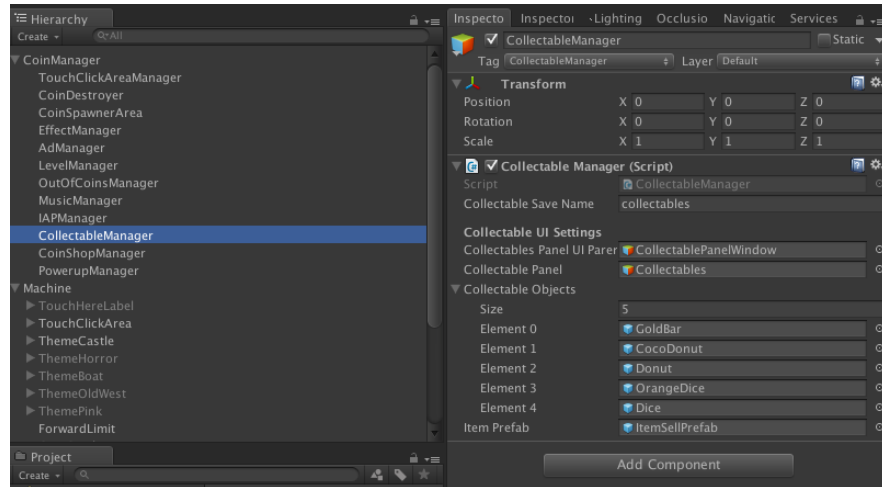
1. Under the prefabs, go into the “CollectableItems” folder and duplicate, for example, the Dice. (You have the option to use a pre-loaded item, or one of your own. Directions for adding your own are explained in the next two sections.)
2. Assign any new materials to the item.
3. Edit the **CoinEffect** properties according to your preference. (The “Type of Coin” menu lists several pre-loaded options. Directions for adding your own are explained later in this document”.)
4. Open the **CollectableManager**. Expand the “Collectable Objects” array by 1 (one) element.



5. Assign the new collectable item prefab you made to this spot.
6. Go to the **CoinSpawnerArea** gameobject and add this new collectable item to any one of the available arrays. (The group types dictate the frequency of which each coin is spawned, i.e., regular, common, rare, or epic.)

How do I make a custom coin or collectable item using my own model with the preset Coin Effects and Type of Coin?

1. Starting with your object in Unity, add the following:
 - a. **Collider** – This can be whichever collider makes sense for your object.
 - b. **Rigidbody** – TIP: You can reference the other prefabs in the prefabs folder for guidance on this element. Also, make sure the “using gravity” option is checked.
 - c. **CoinEffect** – Edit the CoinEffect properties according to your preference. (The “Type of Coin” menu lists several pre-loaded options. Directions for adding your own are explained in the next section.)
2. Open the **CollectableManager**. Expand the “Collectable Objects” array by 1 (one) element and assign the new collectable item you made to this spot.



3. Go to the **CoinSpawnerArea** gameobject and add this new collectable item to any one of the available arrays. (The group types dictate the frequency of which each coin is spawned, i.e., regular coins/items, common, rare, or epic.)

How do I make a custom coin or collectable item using my own model, and customize the Coin Effects and Type of Coin?

1. Starting with your object in Unity, add the following:
 - a. **Collider** – This can be whichever collider makes sense for your object.
 - b. **Rigidbody** – Make sure the “using gravity” option is checked. (TIP: reference the prefabs in the prefabs folder for guidance on this.)
 - c. **CoinEffect** – This is the script that creates the magic of making an item a coin or a collectable. Make sure you include a reference to this script and customize it how you see fit.
2. If you have added coin effects that fall outside of the preset “Type of Coin” options, you can easily add your own by doing a bit of C# coding.
 - a. Open up **CoinEffect.cs** and look for the **enum Effect** section (at the top):

```

11
12 // Let's define the type of effect this coin is
13 public enum Effect {
14     RegularCoin,
15     BumperWallCoin,
16     BullseyeCoin,
17     CashCoin,
18     GiftCoin,
19     QuakeShakeCoin,
20     CollectableDonut,
21     CollectableCocoDonut,
22     CollectableDice,
23     CollectableOrangeDice,
24     CollectableGoldBar
25 }
26 public Effect typeOfCoin;
27

```

- b. You can add a new effect at the bottom of the “**enum Effect**” declaration.
 - c. Open **EffectsManager.cs** and look for the **public void runEffect(...)** function.
 - d. This function consists of a **switch** statement that handles the different types of effects. This is where you would add your own code and create your own effects. (You can also interface to the local **EffectsManager** and do awesome things like shake the screen from here.)
3. Follow the same steps above for adding this new coin or collectable item to the **CoinSpawnerArea**. If your item is a collectable, you’ll also need to add it to the **CollectableManager** array (see steps above.)

How to add a new PowerUp?

For this example, we will make a new, “Super” Gift Coin that will give the user 10 in-game dollars in addition to shaking the screen (earthquake!)

1. Select the Gift Coin prefab and duplicate it. Rename it “Super Gift Coin”.
2. To add custom rules (10 in-game dollars and shaking the screen) we will need to define a new, custom Type of Coin. To do this, open the script **CoinEffect.cs**.
3. Once in **CoinEffect.cs**, look for **public enum Effect**. Within that, find **enum** and add your new type right below the existing GiftCoin, calling it **SuperGiftCoin**.
4. Open **EffectsManager.cs** and scroll down to **public void runEffect(..)**. This function handles the performing of any defined effects for the coin.
5. Inside the **switch** statement, look for **case Effect.GiftCoin**. Right below the **break;** of that case add in the following new case:

```
case CoinEffect.Effect.SuperCoin:
    this.triggerGiftCoin();
    this.triggerEarthShakeEffect();
    playRewardSFX();
    break;
```
6. Now go back to your new prefab “Super Gift Coin” and, in the **inspector**, change the drop down **Type of Coin** to the new **SuperCoin** you created.
7. On that same **inspector** window you’ll see an area below, on the **CoinEffect**, to define a **Prize Image**. You will need to include a prize image here that is shown both, in the **CoinShop** (when buying it), and as the PowerUp button on the screen during play.
8. Go to the **CoinShopManager**, extend the **CoinShopItems** array by one (1) and drag and drop your new prefab into the last slot.

How do I edit the pusher speed?

1. In the **Machine** gameobject, expand the machine you are working on. (For example, expand **ThemeHorror**.)
2. Scroll down and find **PushBar**. (You will see a script called **Pusher.cs**)
3. Change the speed at which it pushes by adjusting the **MoveSpeed**.
NOTE: You will have to do this for each machine.

Script References

This section defines what the purpose of each script is.

CoinManager.cs

This is the main script that will handle all coin related operations. It keeps track of the amount of coins the player has, handles the reloading of new coins, and controls the UI for the coin amounts.

IAPManager.cs

This script handles all platform related IAP. It also contains the master array of **IAPProduct**'s that can be purchased. Reference this script for sever **buyButton(...)** functions that you can use to allow the user to purchase your items.

IAPEditor.cs

Editor only code to expose the adding of new IAP products through the Window in Unity.

TouchClickManager.cs

This handles the touch/click inputs and converts them into the world space for dropping a coin.

TouchClickArea.cs

This is attached to any empty game object. It defines the area that the players can click/touch that will drop an item.

CoinDestroyer.cs

This is attached to the gameobject that will catch any coins that fall off of the play field (and are not caught in the coin collector bucket.)

CoinSpawnerArea.cs

This is the gameobject where the coins/collectables will be spawned. This also controls which coins/collectables can be spawned in the entire asset.

EffectsManager.cs

This is the effect manager. It's where you will define any new effects. The preloaded, defined effects are the earthquake coin, bull's-eye coin and bumper coin. This also handles popups for showing which effect was preformed.

AdManager.cs

This includes a template to add the Unity ads system (if you want to include ads in your game.)

LevelManager.cs

This is the level manager and it handles the level UI bar as well as keeps track of the player's progression through their current level.

OutOfCoinsManager.cs

This manages what happens when the player is out of coins. It handles notifying the user and prompting them to watch an ad for more coins or, if you include it, in-app purchasing of more coins.

CollectableManager.cs

This manager handles all collectable items in the game. It is where you would also add a collectable item to the array to display it in the Prize Shop for resale.

CoinShopManager.cs

This is the manager for the Coin Shop. It is where you will define what items the player can purchase for in-game cash. This handles all of the buying transactions.

CoinShopItemInformation.cs

This is attached to the prefab for any item that will appear in the Coin Shop. It defines information about the item for sale.

PowerupManager.cs

This handles the usage of any on screen powerups and the on screen UI buttons (the powerup panel).

PowerupItemPrefabInformation.cs

This is the powerup item information that is attached to the prefab.

BucketCoinCollector.cs

This handles the collection of coins into the bucket and reports the correct values, both, to the **CoinManager** for score keeping, and to the **EffectManager** to engage effects.

HideAfter.cs

This hides an object after X seconds.

RemoveAfter.cs

This removes an object after X seconds.

Extras.cs

This includes helpful functions that were designed for this asset.

MainMenuManager.cs

This is a shell written for the main menu. (Very basic layout but included for you to expand as needed.)

PlayerPrefsSerialize.cs

This is a class that will help serialize the **PlayerPrefs** data for saving. It handles the saving and loading of the serialized data.

Pusher.cs

This pushes the push bar in the machine (back and forth) using limits set up in the scene.

AudioManager.cs

This manages all of the audio sources in the asset and controls the ability to mute them using the UI button.