

Integrating Unity IAP In Your Game

Checked with version: 5.3

-

Difficulty: Intermediate

<https://unity3d.com/learn/tutorials/topics/ads-analytics/integrating-unity-iap-your-game>

Unity IAP lets you sell a variety of items directly within your free or paid game including premium content, virtual goods and subscriptions. Unity IAP makes it easy to implement in-app purchases (IAP) in your application for the most popular app stores. The currently supported stores include: the iOS App Store, Mac App Store, Google Play, Windows Store (Universal) and Amazon Appstore.

Background Project

In this article, we are going to look at how to add IAP to an existing game project. This article uses a modified version of the [Survival Shooter project](#). The original tutorial for this game project is [available here](#). The Survival Shooter project was extended as a live training session to create a weapon shop that allowed players to upgrade their starting weapon by spending in-game currency based on their score. To learn more about how to create the weapon shop in the Survival Shooter project, please see the [live training session "Creating an in-game shop"](#).

Required Download

This tutorial article extends the weapon shop training session further, integrating actual IAP where users can buy currency using real money transactions. Please download the .zip file – [Survival Shooter IAP Demo](#) – as it is required to complete this tutorial.

Once the download is complete, unzip the project and open it in Unity. Open the scene **Level 01 5.x IAP** from the **IAPDemo/Scenes** folder in the Project panel.

Setting up Unity Services

Before we begin any scripting to set up Unity IAP in our project, we will need to have our project set up with Unity Services. Open the Services window by choosing **Window > Services** from the top menu.

If you are **not yet logged in**, you will see the following message:

SERVICES

Unity provides you a suite of integrated services for creating games, increasing productivity and managing your audience.



You are
not logged in

Sign in...

Click **Sign In** to log in with your Unity ID.

Project IDs

Every project using Unity IAP will need a **Project ID** from Unity Services.

If you do not have an ID for your project you will need to create one. First select your **Organization** by using the **Select Organization** drop-down. An organization can be either a single user or a group. The default organization will be an organization composed of a single user, set to the Unity Account you've logged in with. Once the Organization has been selected, click the **Create** button to create a Project ID.

If you have already created a Unity Project ID for your project choose '**I already have a Unity Project ID**'.

SERVICES

Unity provides you a suite of integrated services for creating games, increasing productivity and managing your audience.

Create a Unity Project ID

A Unity Project ID enables services for your project.

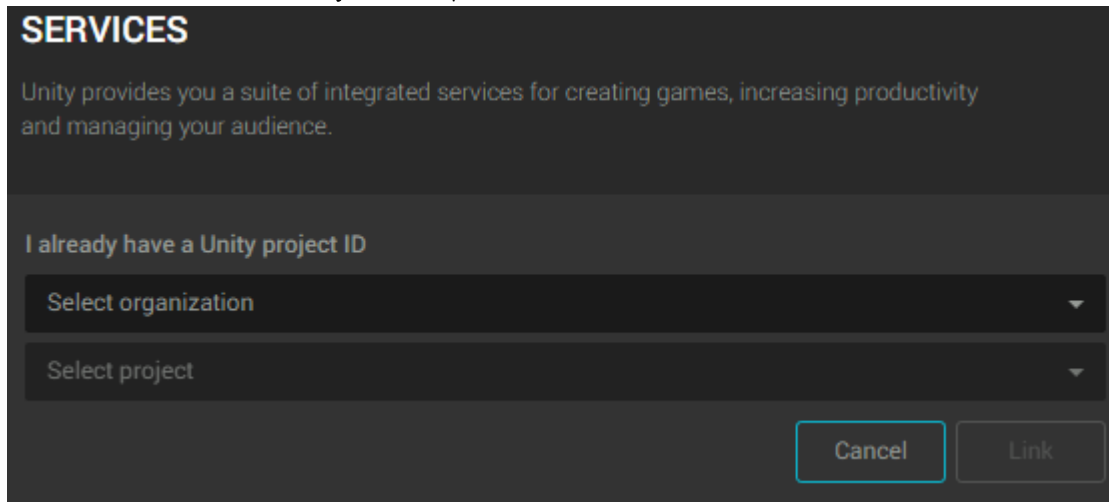
Select organization

Create

[I already have a Unity Project ID](#)

[I already have a Unity Ads game ID](#)

If you already have a Project ID, once your Organization has been selected you will be able to choose from a list of previously created Project IDs. Select an existing Project ID from the **Select Project** drop-down.



SERVICES

Unity provides you a suite of integrated services for creating games, increasing productivity and managing your audience.

I already have a Unity project ID

Select organization ▼

Select project ▼

Cancel Link

Enabling In-App Purchasing

From the list of Services, choose In-App Purchasing.

Survival Shooter IAP Demo

SERVICES

Unity provides you a suite of integrated services for creating games, increasing productivity and managing your audience.

SERVICES

MEMBERS

AGE DESIGNATION

SETTINGS



Ads

OFF

The premier ad monetization solution for mobile games.



Analytics

OFF

Unity Analytics: Understand how your players play!



Cloud Build

Create and share builds automatically. Unity Cloud Build monitors your repos and delivers new builds of your game just like that.



In-App Purchasing

OFF

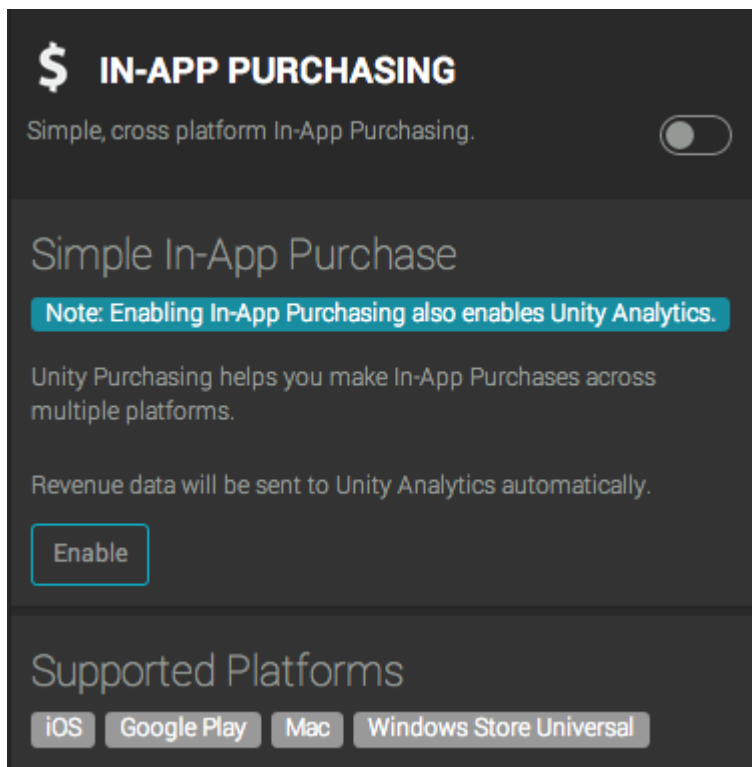
Simple, cross platform In-App Purchasing.



Multiplayer

Access Unity Multiplayer's Matchmaker Service and Relay Server to create networked games for desktop, Xbox One, PS4, iOS, Android and the Unity Web Player.

Next, to Enable In-App Purchasing click the **Enable** button.



COPPA Compliance

The Children's Online Privacy Protection Act, applies to the online collection of personal information from children under 13. The new rules spell out what you must include in a privacy policy, when and how to seek verifiable consent from a parent and what responsibilities you have to protect children's privacy and safety online. You will be prompted with a dialog asking about the target age for users of your app in order to ensure [COPPA compliance](#). If you have already specified a COPPA choice in your Analytics settings, this dialog will not be displayed. Choose the appropriate answer and then click "**Save Changes**".

\$

IN-APP PURCHASING

Simple, cross platform In-App Purchasing.

Designation for Apps Directed to Children Under the Age of 13

In accordance with the Children's Online Privacy Protection Act (COPPA), we require all products that use Unity Analytics to identify whether or not they are directed at children under the age of 13 in the United States. [Learn More](#)

☐

This game is directed to children under the age of 13 in the United States.

☒

This game is NOT directed to children under the age of 13 in the United States.

Cancel

Save Changes

Adding the IAP Package

Unity IAP requires an **imported package** to build your integration upon, this must be added to your project using the **Import** Button shown in the image below.

\$

IN-APP PURCHASING

Simple, cross platform In-App Purchasing.

Welcome

Note: disabling Analytics also disables In-App Purchasing.

Import our store package to get started on:

- Google Play
- iOS App Store
- Mac App Store
- Windows Store

Import

Supported Platforms

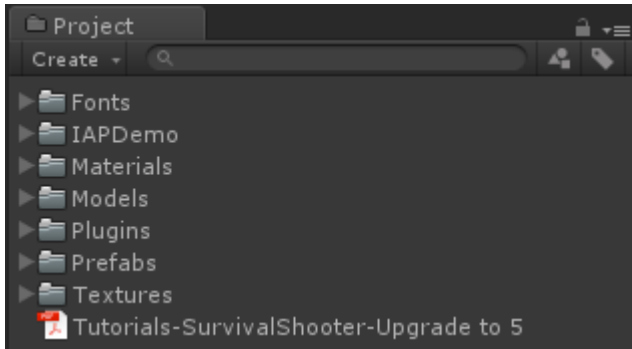
iOS

Google Play

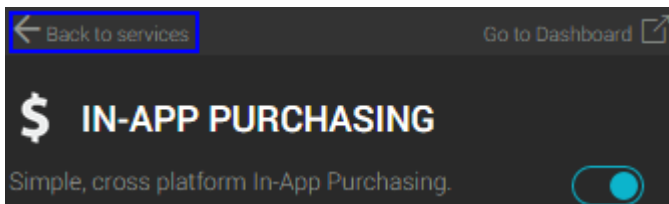
Mac

Windows Store

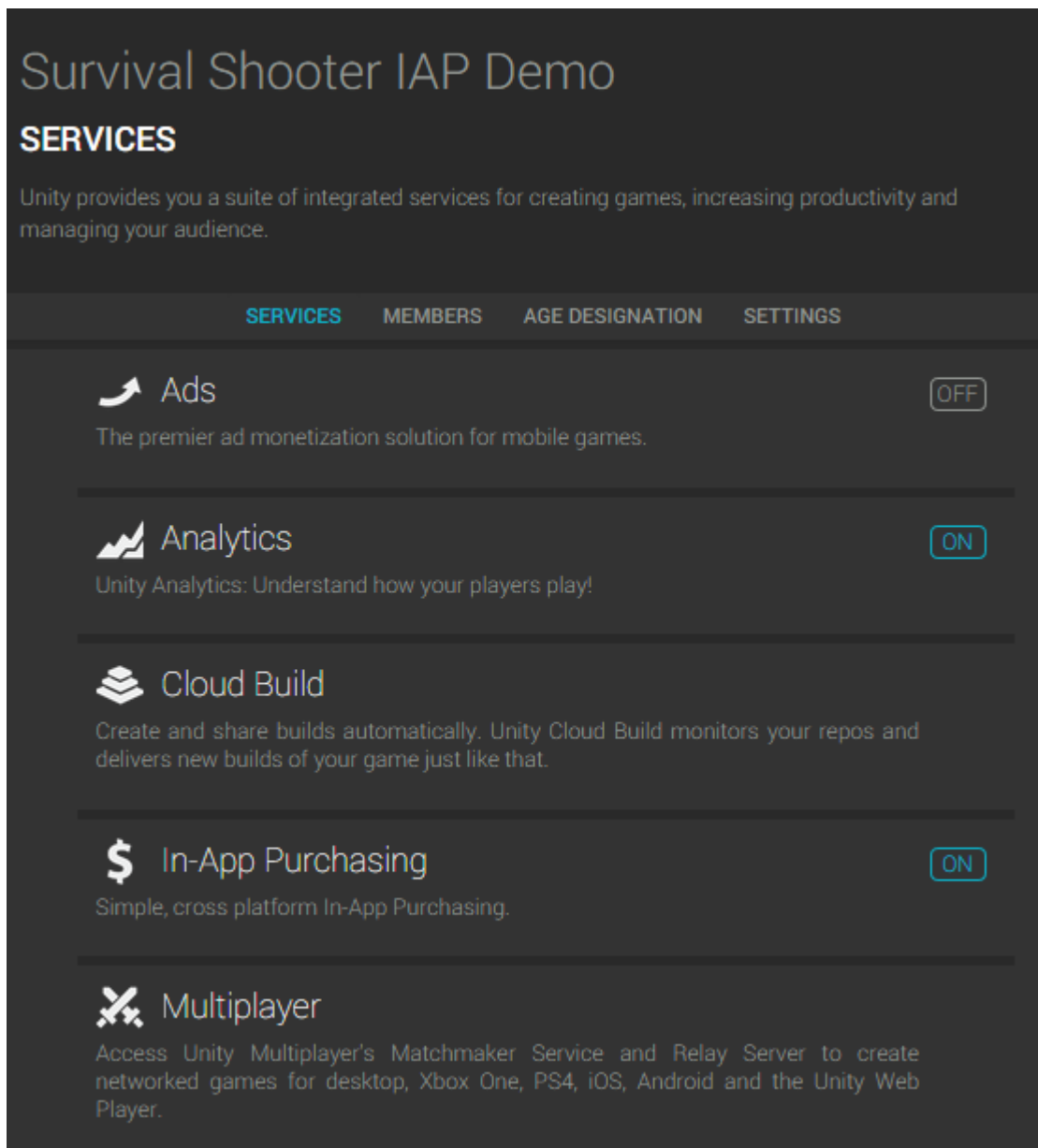
After you have imported the package, you should see a new folder called **Plugins** has been added to your project. This folder contains **UnityPurchasing** assets required to use Unity IAP.



Click 'Back to services' and review the services panel.



You should now see that Analytics and In-App Purchasing are both ON as shown below.



Making a Purchase Script

With Services set up, we can add the required code to our game. First, we will add a script called Purchaser. Purchaser is an example script for this project for working with Unity IAP. Purchaser includes functions which allow us to do the following:

- **InitializePurchasing**: Initializes the IAP builder, adds products that are available for sale and supplies a listener to handle purchasing events.
- **BuyProductID**: A private function which allows us to buy a product we've added using its product ID string.
- **BuyConsumable**, **BuyNonConsumable**, **BuySubscription**: Public functions which allow us to buy products of different types by passing their respective strings to **BuyProductID**.
- **RestorePurchases**: On iOS we can call **RestorePurchases** to restore products previously purchased.

- OnInitialize: Called to check if the app can connect to Unity IAP or not. OnInitialize will keep trying in the background and will only fail if there is a configuration problem that cannot be recovered from.
- OnInitializeFailed: Called when IAP have failed to initialize and logs a message to the console.
- ProcessPurchase: Checks to see if a product purchase was successful and logs the result to the console.
- OnPurchaseFailed: Logs a message to the console telling us when a purchase failed.

In the Project panel, select the IAPDemo folder, then click the **Create** button and make a new C# script called **Purchaser** and paste–replace the entire contents with the following code:

Expand view

Copy code

- [C#](#)

```
using System;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.Purchasing;

// Placing the Purchaser class in the CompleteProject namespace allows it to
// interact with ScoreManager,
// one of the existing Survival Shooter scripts.
namespace CompleteProject
{
    // Deriving the Purchaser class from IStoreListener enables it to receive
    // messages from Unity Purchasing.
    public class Purchaser : MonoBehaviour, IStoreListener
    {
        private static IStoreController m_StoreController;           // The Unity
        Purchasing system.
        private static IExtensionProvider m_StoreExtensionProvider; // The store-
        specific Purchasing subsystems.

        // Product identifiers for all products capable of being purchased:
        // "convenience" general identifiers for use with Purchasing, and their
        // store-specific identifier
        // counterparts for use with and outside of Unity Purchasing. Define
        // store-specific identifiers
        // also on each platform's publisher dashboard (iTunes Connect, Google
        // Play Developer Console, etc.)
    }
}
```

```

        // General product identifiers for the consumable, non-consumable, and
subscription products.
        // Use these handles in the code to reference which product to purchase.
Also use these values
        // when defining the Product Identifiers on the store. Except, for
illustration purposes, the
        // kProductIDSubscription - it has custom Apple and Google identifiers.
We declare their store-
        // specific mapping to Unity Purchasing's AddProduct, below.
public static string kProductIDConsumable = "consumable";
public static string kProductIDNonConsumable = "nonconsumable";
public static string kProductIDSubscription = "subscription";

        // Apple App Store-specific product identifier for the subscription
product.
private static string kProductNameAppleSubscription =
"com.unity3d.subscription.new";

        // Google Play Store-specific product identifier subscription product.
private static string kProductNameGooglePlaySubscription =
"com.unity3d.subscription.original";

void Start()
{
    // If we haven't set up the Unity Purchasing reference
    if (m_StoreController == null)
    {
        // Begin to configure our connection to Purchasing
        InitializePurchasing();
    }
}

public void InitializePurchasing()
{
    // If we have already connected to Purchasing ...
    if (IsInitialized())
    {
        // ... we are done here.
        return;
    }

    // Create a builder, first passing in a suite of Unity provided
stores.

```

```

        var builder =
ConfigurationBuilder.Instance(StandardPurchasingModule.Instance());

        // Add a product to sell / restore by way of its identifier,
associating the general identifier
        // with its store-specific identifiers.
        builder.AddProduct(kProductIDConsumable, ProductType.Consumable);
        // Continue adding the non-consumable product.
        builder.AddProduct(kProductIDNonConsumable,
ProductType.NonConsumable);
        // And finish adding the subscription product. Notice this uses
store-specific IDs, illustrating
        // if the Product ID was configured differently between Apple and
Google stores. Also note that
        // one uses the general kProductIDSubscription handle inside the game
- the store-specific IDs
        // must only be referenced here.
        builder.AddProduct(kProductIDSubscription, ProductType.Subscription,
new IDs(){
            { kProductNameAppleSubscription, AppleAppStore.Name },
            { kProductNameGooglePlaySubscription, GooglePlay.Name },
        });

        // Kick off the remainder of the set-up with an asynchronous call,
passing the configuration
        // and this class' instance. Expect a response either in
OnInitialized or OnInitializeFailed.
        UnityPurchasing.Initialize(this, builder);
    }

private bool IsInitialized()
{
    // Only say we are initialized if both the Purchasing references are
set.
    return m_StoreController != null && m_StoreExtensionProvider != null;
}

public void BuyConsumable()
{
    // Buy the consumable product using its general identifier. Expect a
response either
    // through ProcessPurchase or OnPurchaseFailed asynchronously.

```

```

        BuyProductID(kProductIDConsumable);
    }

    public void BuyNonConsumable()
    {
        // Buy the non-consumable product using its general identifier.
Expect a response either
        // through ProcessPurchase or OnPurchaseFailed asynchronously.
        BuyProductID(kProductIDNonConsumable);
    }

    public void BuySubscription()
    {
        // Buy the subscription product using its the general identifier.
Expect a response either
        // through ProcessPurchase or OnPurchaseFailed asynchronously.
        // Notice how we use the general product identifier in spite of this
ID being mapped to
        // custom store-specific identifiers above.
        BuyProductID(kProductIDSubscription);
    }

    void BuyProductID(string productId)
    {
        // If Purchasing has been initialized ...
        if (IsInitialized())
        {
            // ... look up the Product reference with the general product
identifier and the Purchasing
            // system's products collection.
            Product product = m_StoreController.products.WithID(productId);

            // If the look up found a product for this device's store and
that product is ready to be sold ...
            if (product != null && product.availableToPurchase)
            {
                Debug.Log(string.Format("Purchasing product asynchronously:
'{0}'", product.definition.id));
                // ... buy the product. Expect a response either through
ProcessPurchase or OnPurchaseFailed
                // asynchronously.

```

```

        m_StoreController.InitiatePurchase(product);
    }
    // Otherwise ...
    else
    {
        // ... report the product look-up failure situation
        Debug.Log("BuyProductID: FAIL. Not purchasing product, either
is not found or is not available for purchase");
    }
}
// Otherwise ...
else
{
    // ... report the fact Purchasing has not succeeded initializing
yet. Consider waiting longer or
    // retrying initialization.
    Debug.Log("BuyProductID FAIL. Not initialized.");
}
}

```

// Restore purchases previously made by this customer. Some platforms automatically restore purchases, like Google.

// Apple currently requires explicit purchase restoration for IAP, conditionally displaying a password prompt.

```

public void RestorePurchases()
{
    // If Purchasing has not yet been set up ...
    if (!IsInitialized())
    {
        // ... report the situation and stop restoring. Consider either
waiting longer, or retrying initialization.
        Debug.Log("RestorePurchases FAIL. Not initialized.");
        return;
    }

    // If we are running on an Apple device ...
    if (Application.platform == RuntimePlatform.IPhonePlayer ||
        Application.platform == RuntimePlatform.OSXPlayer)
    {
        // ... begin restoring purchases
        Debug.Log("RestorePurchases started ...");

        // Fetch the Apple store-specific subsystem.
    }
}

```

```

        var apple =
m_StoreExtensionProvider.GetExtension<IAppleExtensions>();
        // Begin the asynchronous process of restoring purchases. Expect
a confirmation response in
        // the Action<bool> below, and ProcessPurchase if there are
previously purchased products to restore.
        apple.RestoreTransactions((result) => {
            // The first phase of restoration. If no more responses are
received on ProcessPurchase then
            // no purchases are available to be restored.
            Debug.Log("RestorePurchases continuing: " + result + ". If no
further messages, no purchases available to restore.");
        });
    }
    // Otherwise ...
    else
    {
        // We are not running on an Apple device. No work is necessary to
restore purchases.
        Debug.Log("RestorePurchases FAIL. Not supported on this platform.
Current = " + Application.platform);
    }
}

//
// --- IStoreListener
//

public void OnInitialized(IStoreController controller, IExtensionProvider
extensions)
{
    // Purchasing has succeeded initializing. Collect our Purchasing
references.
    Debug.Log("OnInitialized: PASS");

    // Overall Purchasing system, configured with products for this
application.
    m_StoreController = controller;
    // Store specific subsystem, for accessing device-specific store
features.
    m_StoreExtensionProvider = extensions;
}

```

```

    public void OnInitializeFailed(InitializationFailureReason error)
    {
        // Purchasing set-up has not succeeded. Check error for reason.
        Consider sharing this reason with the user.
        Debug.Log("OnInitializeFailed InitializationFailureReason:" + error);
    }

    public PurchaseProcessingResult ProcessPurchase(PurchaseEventArgs args)
    {
        // A consumable product has been purchased by this user.
        if (String.Equals(args.purchasedProduct.definition.id,
            kProductIDConsumable, StringComparison.Ordinal))
        {
            Debug.Log(string.Format("ProcessPurchase: PASS. Product: '{0}'",
                args.purchasedProduct.definition.id));
            // The consumable item has been successfully purchased, add 100
            coins to the player's in-game score.
            ScoreManager.score += 100;
        }
        // Or ... a non-consumable product has been purchased by this user.
        else if (String.Equals(args.purchasedProduct.definition.id,
            kProductIDNonConsumable, StringComparison.Ordinal))
        {
            Debug.Log(string.Format("ProcessPurchase: PASS. Product: '{0}'",
                args.purchasedProduct.definition.id));
            // TODO: The non-consumable item has been successfully purchased,
            grant this item to the player.
        }
        // Or ... a subscription product has been purchased by this user.
        else if (String.Equals(args.purchasedProduct.definition.id,
            kProductIDSubscription, StringComparison.Ordinal))
        {
            Debug.Log(string.Format("ProcessPurchase: PASS. Product: '{0}'",
                args.purchasedProduct.definition.id));
            // TODO: The subscription item has been successfully purchased,
            grant this to the player.
        }
        // Or ... an unknown product has been purchased by this user. Fill in
        additional products here....
        else
        {

```

```

        Debug.Log(string.Format("ProcessPurchase: FAIL. Unrecognized
product: '{0}'", args.purchasedProduct.definition.id));
    }

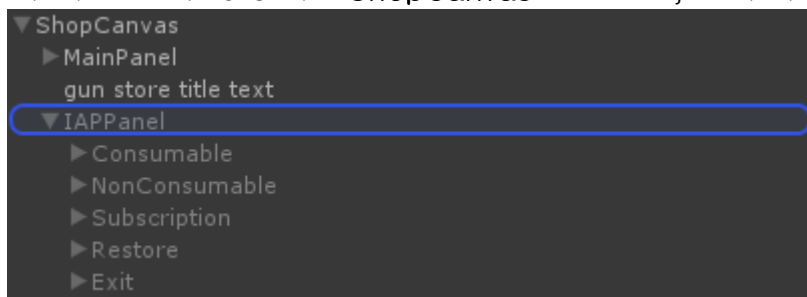
    // Return a flag indicating whether this product has completely been
    // received, or if the application needs
    // to be reminded of this purchase at next app launch. Use
    PurchaseProcessingResult.Pending when still
    // saving purchased products to the cloud, and when that save is
    // delayed.
    return PurchaseProcessingResult.Complete;
}

public void OnPurchaseFailed(Product product, PurchaseFailureReason
failureReason)
{
    // A product purchase attempt did not succeed. Check failureReason
    // for more detail. Consider sharing
    // this reason with the user to guide their troubleshooting actions.
    Debug.Log(string.Format("OnPurchaseFailed: FAIL. Product: '{0}',
PurchaseFailureReason: {1}", product.definition.storeSpecificId, failureReason));
}
}
}

```

Save your script and return to Unity.

From the Project panel drag the Purchaser script onto the game object **IAppPanel** which is a child of the **ShopCanvas** GameObject in the Hierarchy.



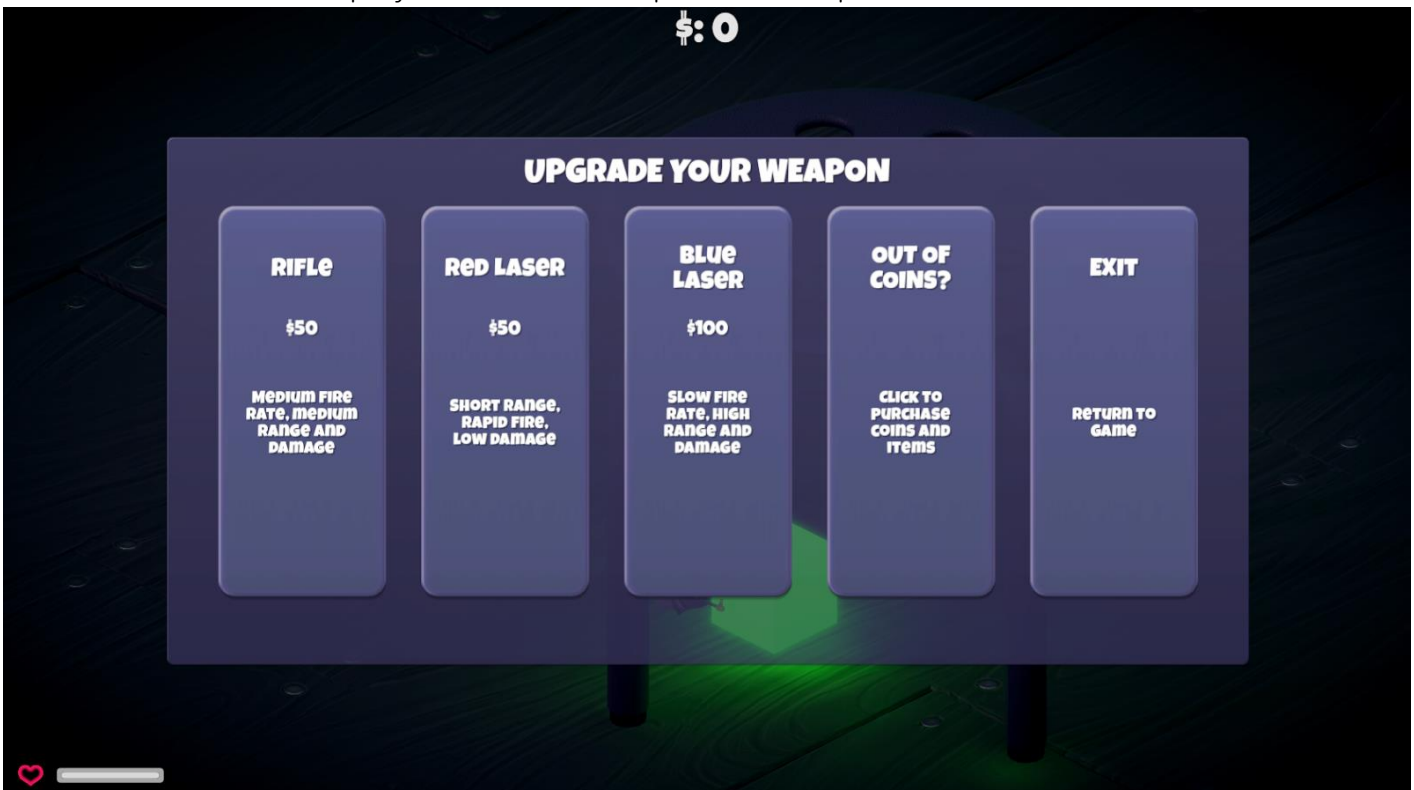
We're going to use our Purchaser script to make an in-app purchase request for a consumable item called '100 coins'. When the item is purchased successfully it will add 100 coins to the player's current score, which they can use to make in-game purchases.

The In-Game Weapon Shop

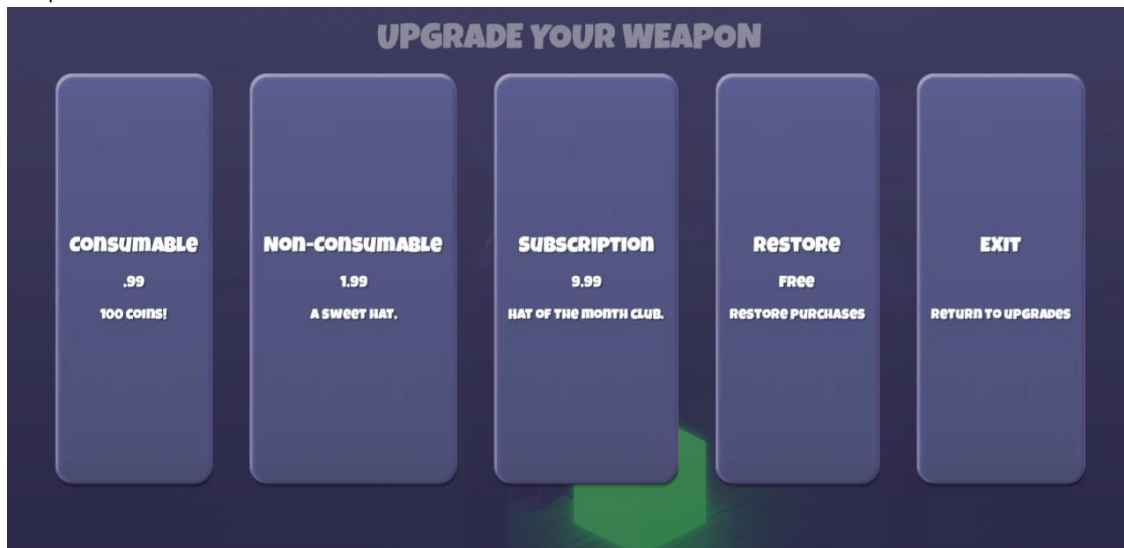
In this project we have given our weapon shop a physical location that the player walks to when they want to buy things. In this case, the shop is represented by a glowing green cube. The player walks up to the cube and the game pauses.



The player is then presented with the weapon shop. If the player doesn't have enough in-game currency to purchase the weapon they want, there is a button labelled 'OUT OF COINS?' which the player can click to open the IAP panel.



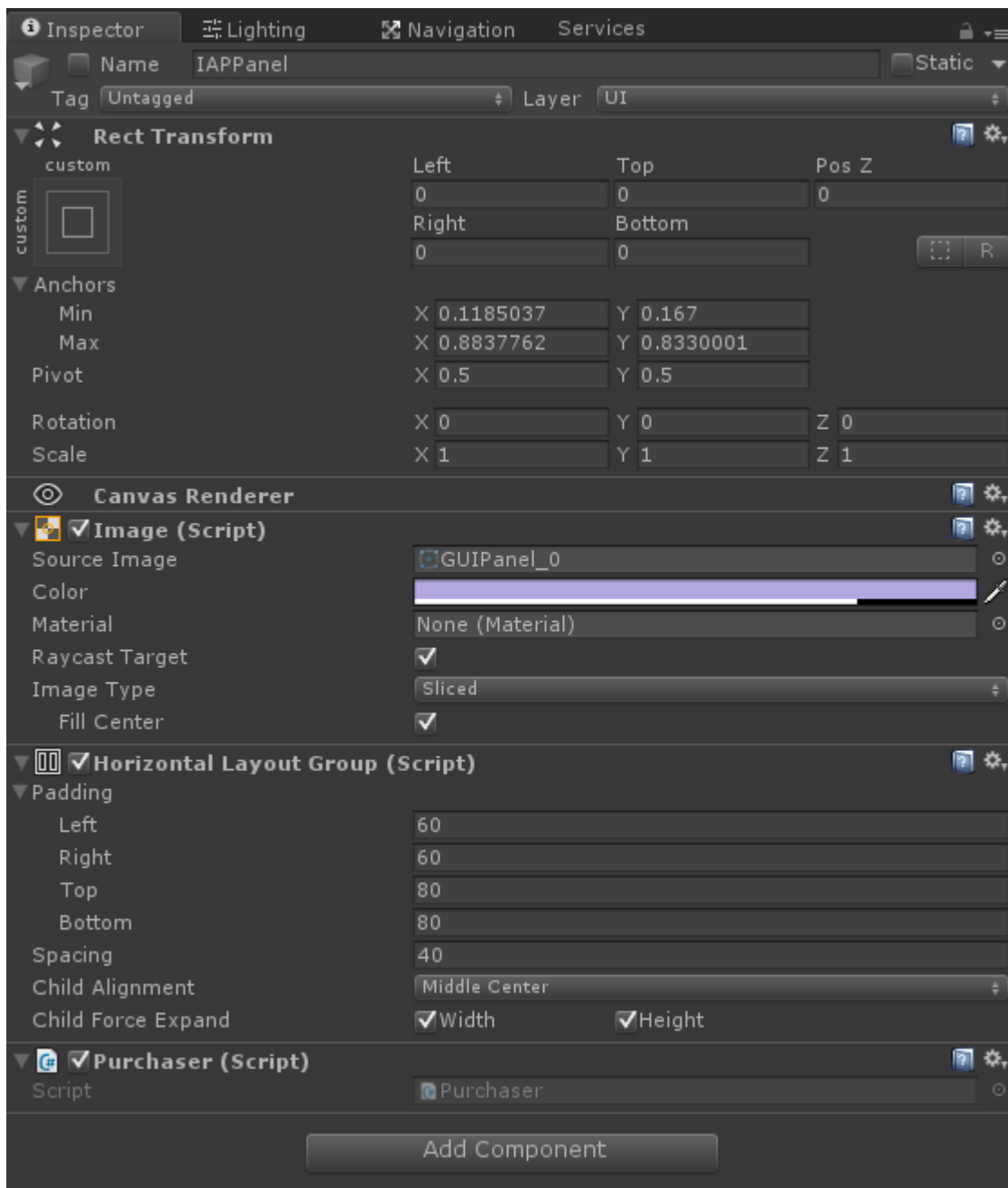
In the IAP panel they have a choice of various IAP products, including a **Consumable** to purchase 100 coins for .99 cents.



Let's take a look at how this is set up.

In the **Hierarchy**, we'll find a GameObject called ShopCanvas. This is a UI Canvas which contains two panels: the **MainPanel** and the **IAPPanel**.

If we highlight the IAPPanel we can see the Purchaser component we just attached to it.

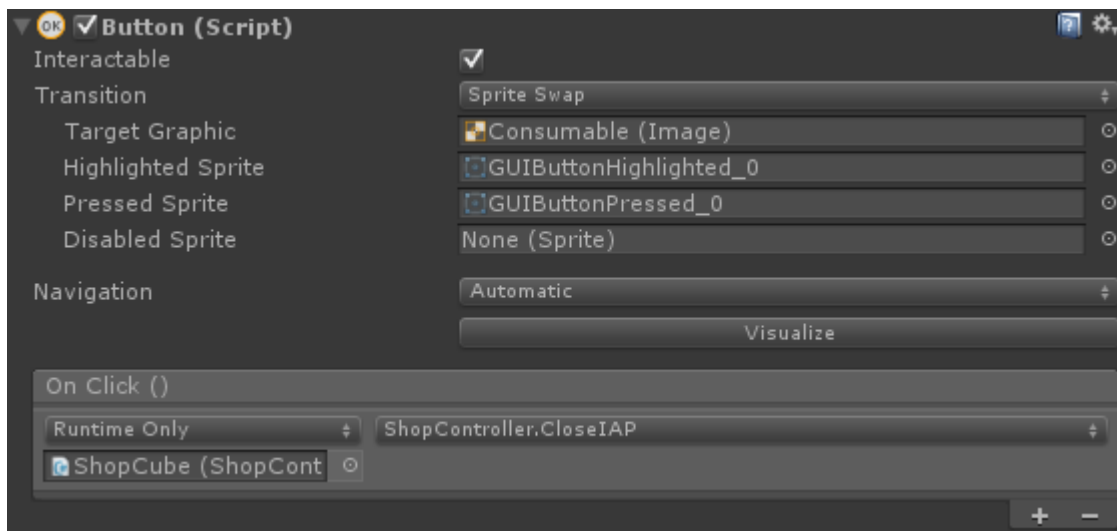


We will call functions of the Purchaser script to make our IAP transactions. We have added a Button GameObject as a child of the IAPPanel called **Consumable**.

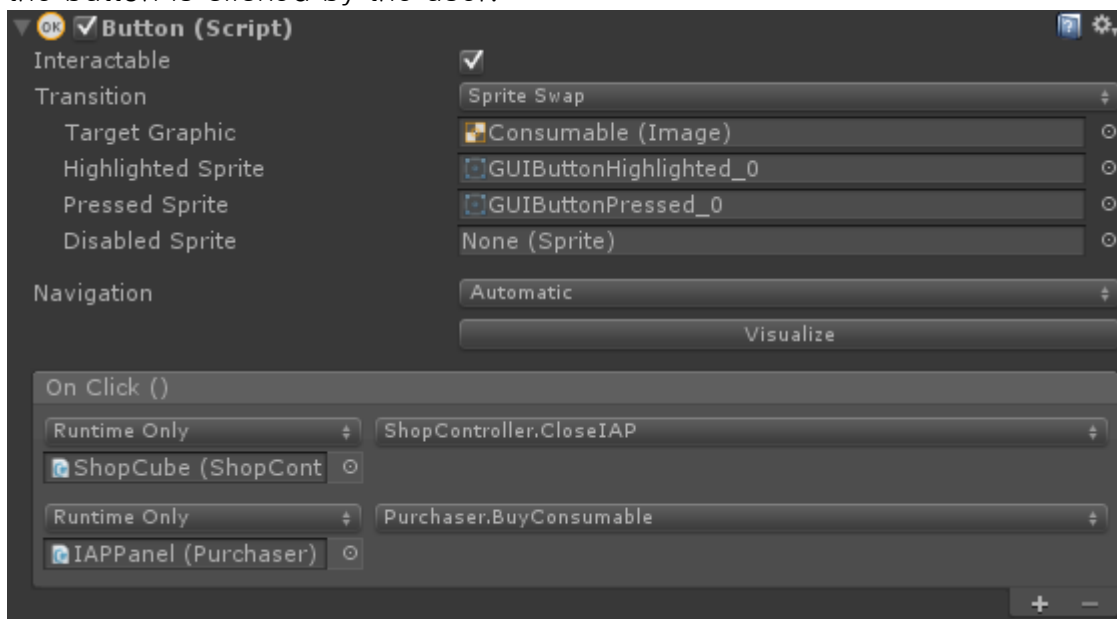
If we highlight our Consumable game object in the **Hierarchy**, we can see the Button component. We will add an OnClick event to this Button which will call a function to buy our consumable item when the user activates it.

Calling Purchase Functions

The first OnClick event has already been set up. This event holds a reference to our ShopCube which contains the ShopController script. When the Button is activated, this event will call the CloseIAP function of ShopController. This function will deactivate the IAPPanel GameObject closing the UI panel. This has already been configured.



On the Consumable Button component, in the OnClick table of events, click the + button to add a new event. The **Purchaser** script has the function we want to call. This script is on the **IAPPanel**. Select the IAPPanel in the Hierarchy and drag it onto the empty Object field in the new entry in the OnClick table. This Object field will say "**None(Object)**". Next, use the drop-down menu to the right of the Object field and select **Purchaser > BuyConsumable**. This will call the **BuyConsumable** function when the button is clicked by the user.




We'll skip setting up the other buttons for this article. If you want to implement them simply repeat the steps above making sure to call the appropriate function for each button – **BuyNonConsumable()**, **BuySubscription()** and **RestorePurchases()**.

Testing the IAP integration

When we play our scene we will need to move our character to the weapon shop (green cube). When the UI appears click **Out of Coins** followed by the **Consumable** button. A series of messages will be logged to the console.


The first message is:

 OnInitialized: PASS
UnityEngine.Debug:Log(Object)

OnInitialized: PASS UnityEngine.Debug:Log(Object)

This tells us that IAP was successfully initialized when we started the scene.

Next, we see the message:

 Purchasing product asynchronously: 'consumable'
UnityEngine.Debug:Log(Object)

Purchasing product asynchronously: 'consumable' UnityEngine.Debug:Log(Object)

This tells us that we attempted to purchase a product asynchronously. In the Unity editor, the purchase will process immediately (see next step). When your app is live and purchases are being made over the network the user will be asked by their operating system to confirm the purchase and there may be a time delay as the purchase processes. Once the purchase has completed, (or failed) the OS dialog will close and the player will be returned to the game.

Next we see the following:

 ProcessPurchase: PASS. Product: 'consumable'
UnityEngine.Debug:Log(Object)

ProcessPurchase: PASS. Product: 'consumable' UnityEngine.Debug:Log(Object)

This tells us that our attempt to process our purchase was a success!

Finally, we see:

 purchase({0}): consumable
UnityEngine.Purchasing.PurchasingManager:InitiatePurchase(Product)

purchase({0}):

consumableUnityEngine.Purchasing.PurchasingManager:InitiatePurchase(Product)

This tells us that the purchasing process has been completed.

To make this simple example work, there is a line in the PurchaseProcessingResult function of the Purchaser script. This line adds 100 to the score variable of the ScoreManager. The ScoreManager is part of the original Survival Shooter game. The code looks like this –

ScoreManager.score += 100;

In a production scenario, you should de-couple the IAP and scoring further.

As mentioned above it is worth noting that when testing in the editor,

PurchaseProcessing will always succeed. In order to test actual purchases in the cloud you will need to build to your target device and create a sandbox for testing.

Please refer to the Unity IAP section of the Unity Manual for more information.

@