Tanks!!! Reference

Section 1 - Introduction

Tanks!!! is a multiplayer game based on Unity's matchmaking service. Players can compete in three different game modes (Last Man Standing, Deathmatch and Team Deathmatch) and earn in-game currency. This currency can be used to buy different tanks (with balanced stats) and decorations for their tanks (which are purely cosmetic, allowing players to express their individuality). There is also a set of 10 training levels which provide unique challenges and reward the player with in-game currency.

This document is a high-level overview of the workings of the Project, aimed at users who have downloaded and installed the Unity **Tanks!!!** package.

The game itself is available on multiple app stores. If you haven't already played it, download it and try it before reading this guide to understand the game flow in detail.

- App Store (iOS)
- Mac App Store (macOS)
- Windows Store (Windows desktop)
- Google Play (Android)

Section 2 - Getting started

This section explains how to connect your Project to the various Unity Services and get the Project ready to be run.

> Note: This Project was originally created, tested, and released with Unity 5.5.1P2, available from https://unity3d.com/get-unity/download/archive.

2.1 Unity Project

When you create a Unity Project, Unity gives you the option to select an organization, which is based on your Unity account. If the organization drop-down is not present, this means you are not signed in to your Unity account, and you need to sign in before you continue.

Open your new Project in Unity. In the Unity Editor, click the Cloud button in the top-right corner of the Unity Editor (see Figure 1) to open the Unity Services window.

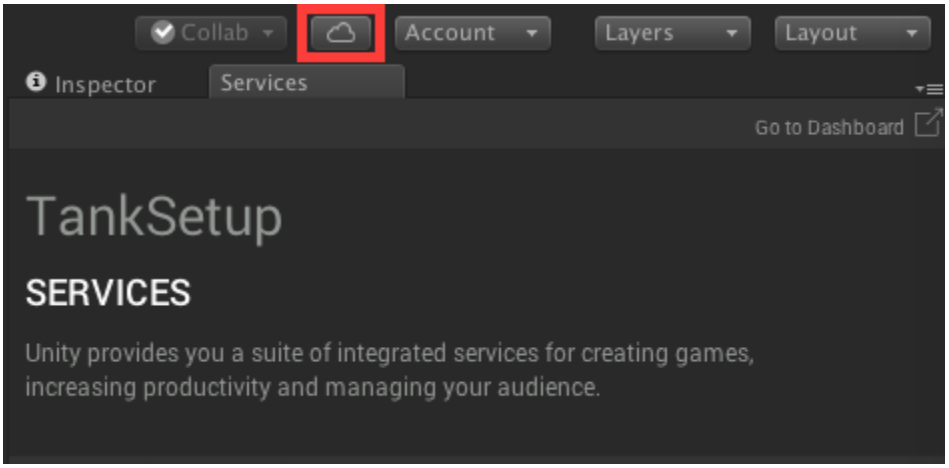

Figure 1: The Unity Services button, highlighted in the red box

The Unity Services window looks like this:

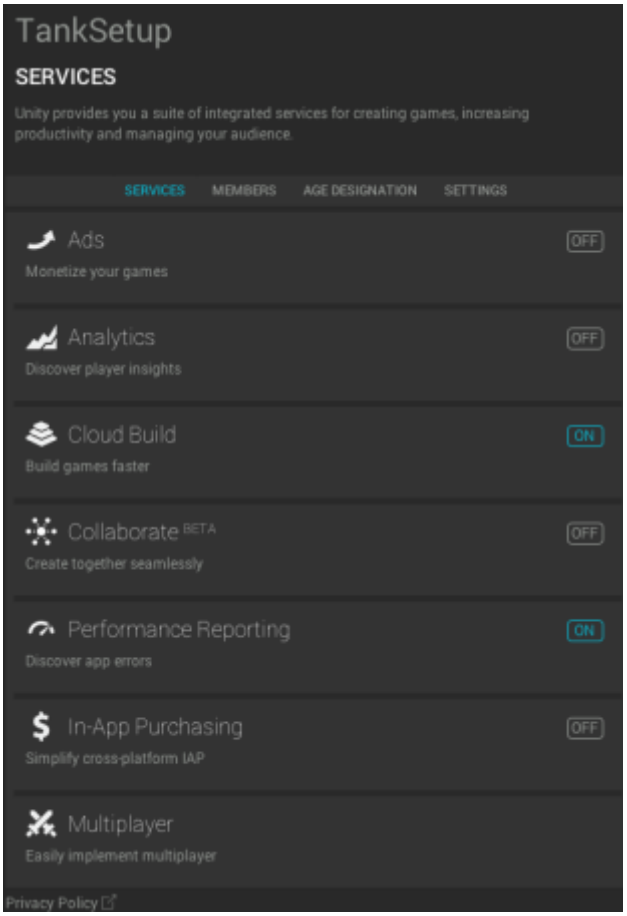

Figure 2: The Unity Services Window. In this screenshot, Cloud Build and Performance reporting are enabled, while everything else is disabled.

From here, click **Settings**. By default, your Project is linked to a Developer Cloud Project. The Settings menu allows you to rename the Project, change the organization, or unlink the Project from your Unity ID. This last option is useful if you need to associate it with a Project that has already been set up on the Developer Cloud.

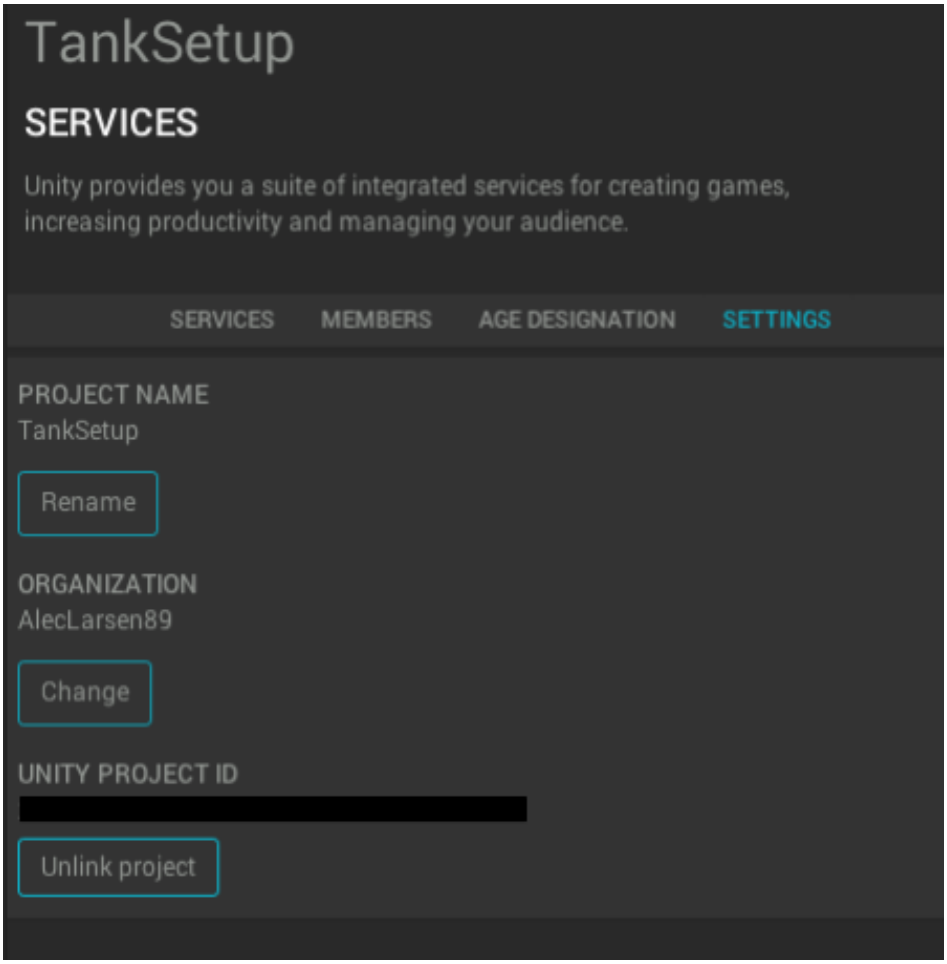

Figure 3: Unity Services settings - the linked Developer Cloud Project.

All Unity Services are managed using the Unity Services Dashboard.

2.2 Collaborate

Unity Collaborate is Unity's fully integrated Service to allow you to share, save, and synchronize your Project. Collaborate also allows users linked to the Developer Cloud Project or the organization to become contributors on the Project.

To enable Collaborate, open the Services window and click the **Services** tab, then click **Collaborate**. This opens the Collaborate window; to enable the Service, click the toggle at the top next to **Create together seamlessly**. This Service is especially useful if you are working as part of a team. For further information, see Unity documentation on Unity Collaborate.

When Collaborate is enabled, the iconography in the Project hierarchy changes. Items either have a blue plus (+) above them to represent that they are new, or green ellipsis (...) to indicate that a change has been made (see Figure 4).

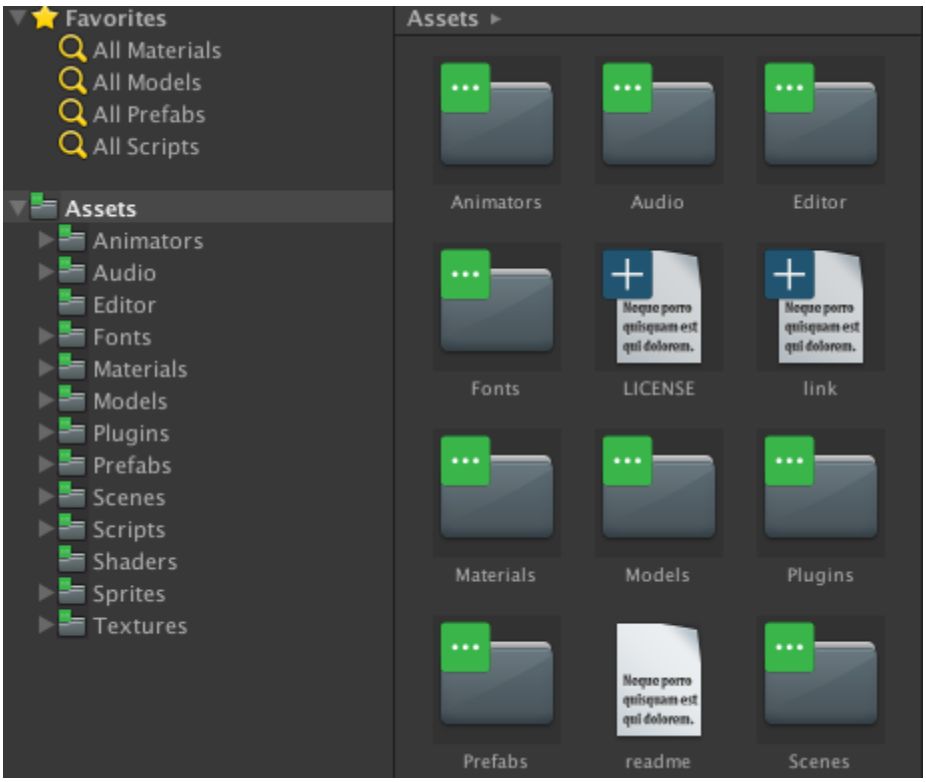

Figure 4: Iconography in the Project hierarchy to represent file changes

2.3 Multiplayer

In Unity, go to the Services window and click on the **Services** tab, then click **Multiplayer**. The Multiplayer window appears as shown in Figure 5, below. Click **Go To Dashboard** to open the Unity Services Dashboard in a browser window, with the Multiplayer configuration setup (see Figure 6). For further information, see Unity documentation on Setting up Unity Multiplayer.

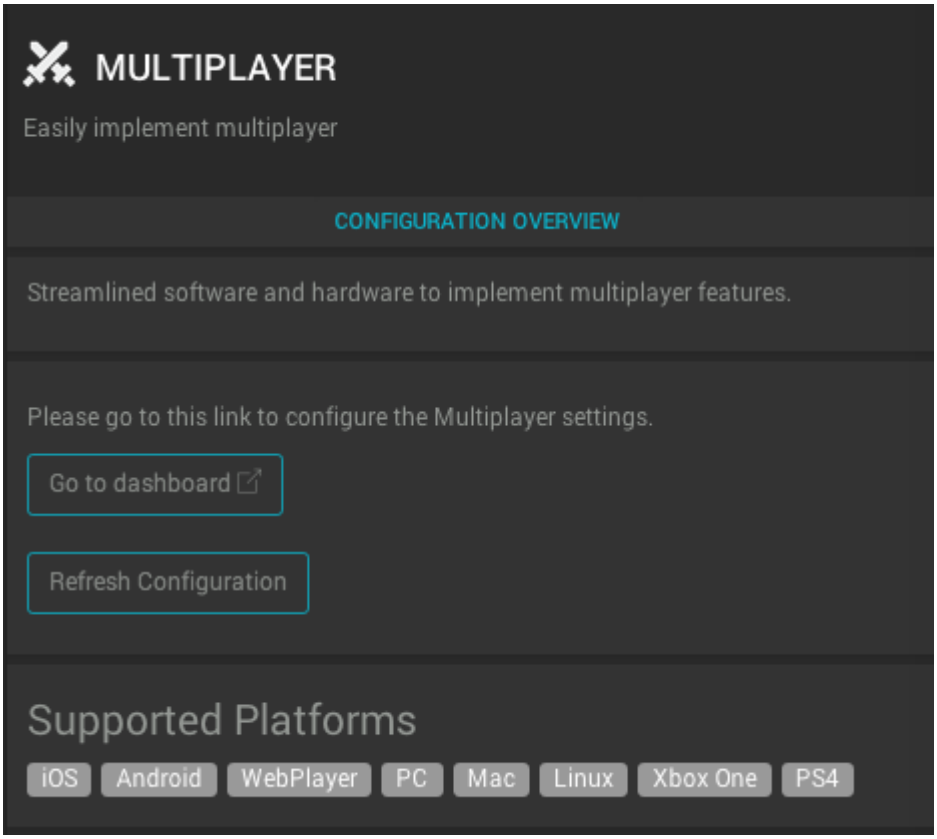

Figure 5: The Multiplayer services window

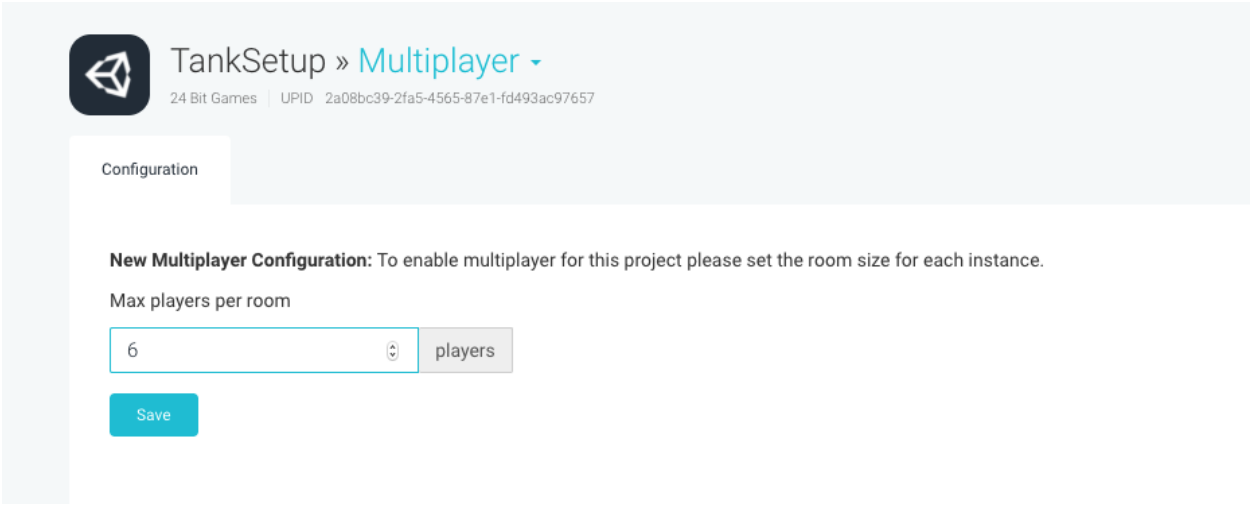

Figure 6: Multiplayer configuration in the Unity Services Dashboard

Note: If you do not set the player room size, the game will report “Failed to create game” to the player when attempting to Create Game.

2.4 Performance Reporting

Performance reporting provides crash and error reports of your game via the in-browser Unity Services Dashboard. It is a useful tool to determine the stability of your game and should be enabled early in the development life cycle.

To enable this, open the Services window, select the **Services** tab, and click **Performance Reporting**. This opens the Performance Reporting window. To enable the Service, click the toggle at the top next to **Discover app errors**. For further information, see Unity documentation on Unity Performance Reporting.

2.5 Analytics

Unity Analytics provides insight into how players are playing your game, and how you can tweak the gameplay experience.

To enable Unity Analytics, open the Services window, select the **Services** tab, and click **Analytics**. This opens the Analytics window. To enable the Service, click the toggle at the top next to **Discover player insights**. For further information, see Unity documentation on Unity Analytics.

Use the Unity Services Dashboard to configure Unity Analytics to the correct version (see Figure 7).

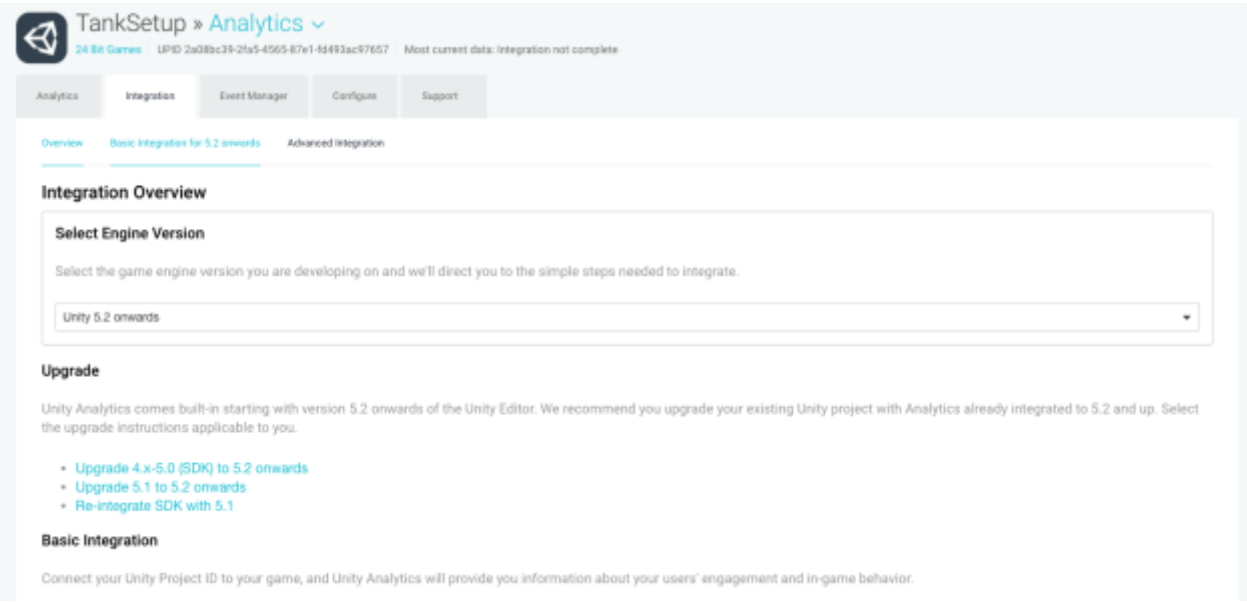

Figure 7: Configuring Analytics via the web portal

Unity provides some basic analytics calls out of the box, but generally you should always track custom events (see **Section 6: Optimizing Your Game**). Unity also has a Heatmaps functionality, which allows you to track custom analytics events that occur at specific positions in world space and at certain times. To download the Heatmaps code and read the documentation, see the Unity Technologies BitBucket page.

2.6 Ads

Unity Ads provides a built-in service that allows you to display paid adverts in your Android or iOS game in exchange for a portion of advertising revenue.

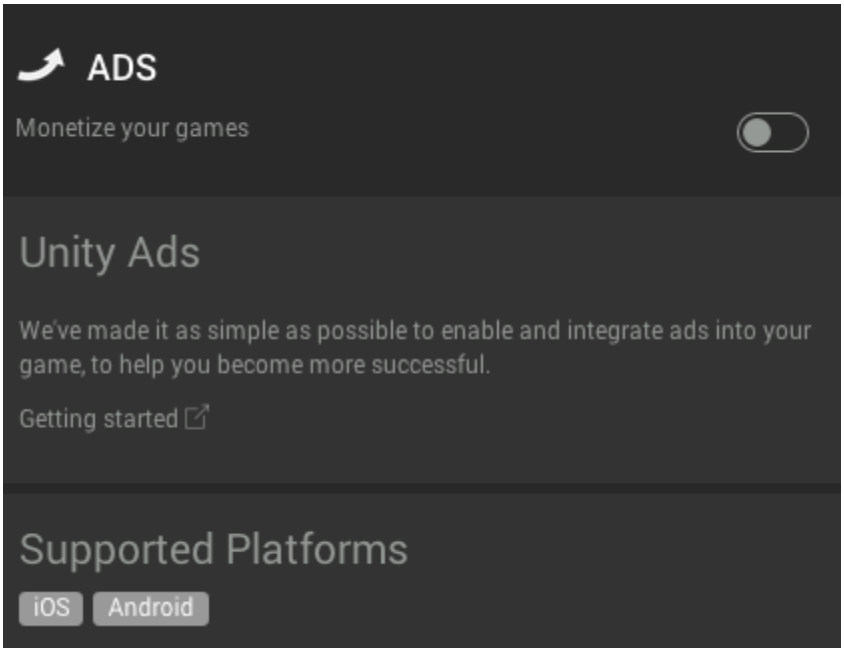

Figure 8: Enabling Ads

To enable Unity Ads, open the Services window, select the **Services** tab, and click **Ads**. This opens the Ads window. To enable the Service, click the toggle at the top next to **Monetize your games**. For further information on using Unity Ads, see Unity documentation on How to use Unity Ads.

Be aware that Unity Ads is only available for Android and iOS devices, and can be tested with emulation in the Unity Editor when it is set to compatible build targets.

Although disabled by default, **Tanks!!!** contains an implementation of the Unity Ads framework. This is handled by the **TanksAdvertController** script situated on the **TanksAdManager** GameObject in the **LobbyScene** Scene.

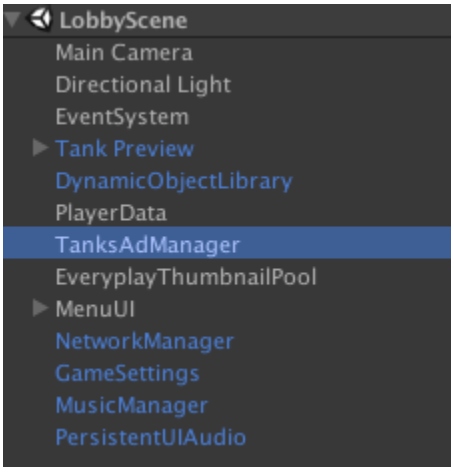

*Figure 9: **TanksAdManager** in the Scene Hierarchy*

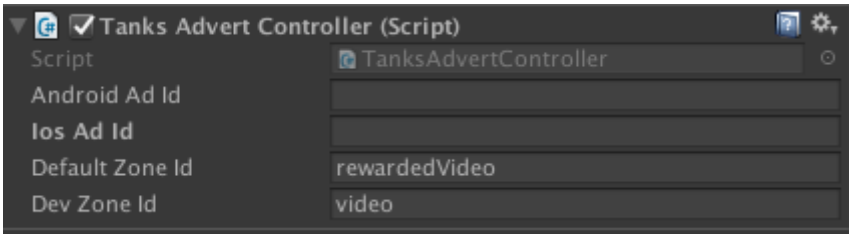

*Figure 10: Properties in the **TanksAdvertController** script*

The script automatically re-enables itself when Ads is enabled for the Project, and the necessary libraries are automatically downloaded. You can then paste the **Android Ad ID** and/or the **iOS Ad ID** assigned to your Project by Unity Ads, as well as any zones you have set up, and use the script to run adverts. Much of the ad-related UI controls and scripting remains in the game and can be re-enabled. Alternatively, you can choose to create your own.

2.7 Project Setup

Before attempting to run or build the Project, you need to set up the physics and add the Scenes to the Build Settings. To set up the physics, in the Unity menu bar go to **Edit** > **Project Settings** > **Physics**. Set up the **Gravity** and the **Layer Collision Matrix** as shown in Figure 11, below. Add all of the Scenes to the Build Settings as displayed in Figure 12, below (see Unity documentation on Build Settings for more information on how to do this).

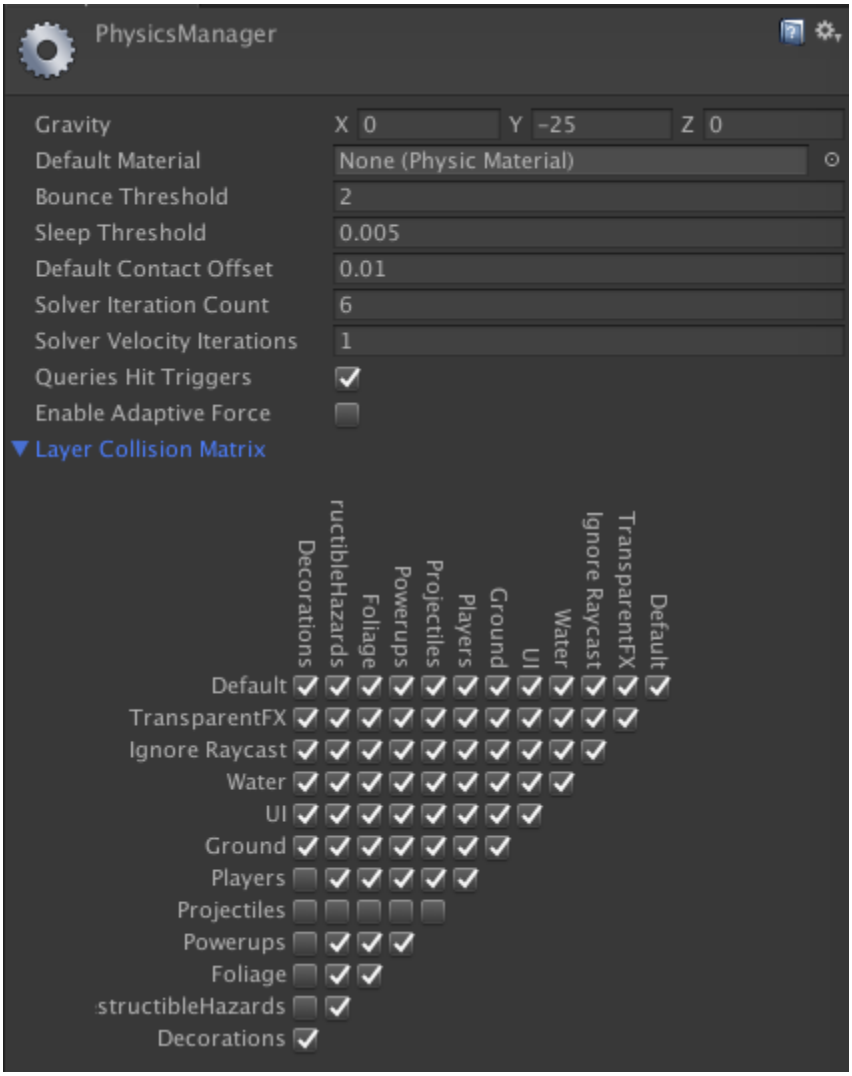

Figure 11: Physics setup, including collision layers

Figure 12: Scenes in the Build Settings.

Section 3 - Architecture

For **Tanks!!!**, a custom *NetworkManager* extends the built-in Unity NetworkManager to include functionality specific to our needs. Single player is treated as a *localhost* direct game to minimize re-architecting of the core game logic.

The main menu of the game is made up of panels, each of which has a class that controls the functionality of that screen.

Tanks!!! main menu

All of the panels are referenced in the *MainMenuUI* object, which controls the transitions between the different screens.

The levels and modes that appear on the respective screens are stored in ScriptableObjects. ScriptableObjects are data containers. They are similar to MonoBehaviours, except they are not attached to GameObjects, and they can be saved as Assets in the Project (for more information, check out the tutorial on ScriptableObjects).

For example, *MapDetails* is a serializable class representing a multiplayer map with serializable fields, including the name, description (simple flavor text), Scene name and preview image. There are many maps, each with a *MapDetails* instance. Another class called *MapList* contains a list of these *MapDetails* instances. *MapList* is derived from *SerializedObject*, so that it can be serialized.

To add an existing Scene as a multiplayer map, simply add it as an entry to the map list. To create a new multiplayer map, it's easiest to duplicate an existing map and use it as a template. The great thing about using ScriptableObjects is that they do not have to be attached to any GameObject or Prefab and, as such, can be kept as a separate configuration layer. This allows you to give non-technical members of your development team access to the configuration without putting any of the Prefabs at risk. In our case, we were able to give our copy editors access to the configuration so that they could adjust names and flavor text to conform to a specific style.

The gameplay is managed by a *GameManager* class, which receives game play settings from the *GameSettings* class. This process is described in detail in **Section 4: Game Management**.

Section 4 - Game management

The *GameManager* class is responsible for running the core game logic, and operates as a state machine (see Unity documentation on State Machine Basics to learn more). It contains a *RulesProcessor* class that enforces game rules. The *RulesProcessor* is a base class that contains functions (called by the *GameManager*) that are overloaded in order to implement the different game modes (**Last Man Standing**, **Deathmatch**, **Team Deathmatch**, **Decoration mini-game**, and **Single Player**). The *GameManager* gets the *RuleProcessor* from the *GameSettings* class, which contains information about the current game (including the game mode, map, and whether the game is single-player or not). The *GameSettings* class is set up/changed when a game is hosted, joined (settings are sent across the network), or when a single-player game is set up. *GameSettings* is a persistent singleton, which means that only one instance can exist and that it persists between game Scenes.

Although the *GameManager* is a *NetworkBehaviour*, the game loop only runs on the server (so that the game is server authoritative), and the *GameManager* simply uses the networking functionality for syncing the display of information on Clients (the HUD, various modals, and the kill log) and syncing the spawning of objects on Clients.

The *RulesProcessor* class has helper functions for handling player death (including how to treat suicides and kills), respawn, and game state, and these are overloaded for the different game modes. For example, when one player kills another:

- **Deathmatch**: The player that did the killing gets 1 point. The player that died respawns after a delay.
- **Team Deathmatch**: The player that did the killing gets 1 point. That player's team gets one point. The player that died respawns after a delay.
- **Last Man Standing**: No one respawns or gets points because respawn and points are handled at round end (when there is one or fewer players left in the game).

See the commented code base to read further differences in the rule processing. This is in your Project's root folder.

As mentioned in **Section 3: Architecture**, single-player games are treated as multiplayer games, with one player (the host) hosted on *localhost* as a direct connection. This is set up like this to minimize the amount of re-architecting required, and makes it possible to share the aforementioned *GameManager* class and differ in the *RulesProcessor* used.

Single-player games, including the decoration mini-game and training levels, use the *OfflineRulesProcessor,* which is a child class of *RulesProcessor*. The decoration mini-game uses the *ShootingRangeRulesProcessor*, and it extends the *OfflineRulesProcessor*, providing flow for unlocking the decorations and setting the player to invulnerable so that it is impossible to kill yourself in the mini-game. The training levels use the *SinglePlayerRulesProcessor*, which also extends the *OfflineRulesProcessor,* and has a list of objectives. Objectives revolve around non-playable characters (NPCs) and points of interest (such as killing an NPC before it reaches the end zone). An *Npc* base class calls a function on the *SinglePlayerRulesProcessor,* which in turn calls a function on each of its objectives.

Similarly, a *TargetZone* object detects when a GameObject enters its triggers and calls a function on the *SinglePlayerRulesProcessor*, which in turn calls a function on each of its objectives. Each objective extends the *Objective* base class and keeps track of whether the player has Achieved or Failed a single-player objective. The *Objective* base class defines a serializable bool for marking whether or not that objective is the primary objective, and an integer for the award currency. A mission is considered failed if the player dies or if the primary objective is failed. A mission is considered a success if the primary objective is achieved. Consult the code base to see current SinglePlayer objective types.

Taking Mission 1 of **Tanks!!!** as an example:

1. Expand **Prefabs** > **SinglePlayer** > **Levels** > **Mission1_MoveAndShoot**
2. Click on **Mission1_Rules_Processor** and expand the **Objective** list under the *SinglePlayerRulesProcessor*
3. Observe the three **Objective** prefabs:
    a. **Mission1_Primary_Objective_Kill**: The primary objective that requires you to kill four NPCs and rewards you 100 in-game currency.
    b. **Mission1_Secondary_Objective_Timed**: A secondary objective that requires you to complete the primary objective in 60 seconds
    c. **Mission1_SecondaryObjective_DontTakeDamage**: A secondary objective that requires the player to not take any damage.

Section 5 - Networking

Most Unity examples for creating multiplayer games encourage a fully server-authoritative approach to shooting: the Client presses the button to fire, which sends a message to the Server, which runs the shooting logic and spawns the projectiles on each Client (see the Unity tutorial on Adding Multiplayer shooting for an example of this).

This works really well in situations with minimal latency. The problem with higher latency environments is that a Client player presses the button to fire and there is a noticeable delay before a bullet is shot, because the Client has to wait for the Server to send it a message to shoot (illustrated in Figure 13).

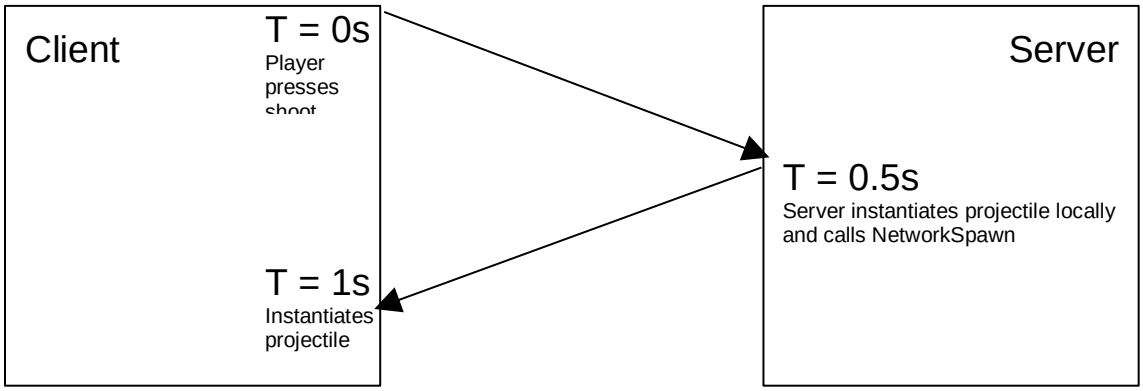

Figure 13: Assuming transmission delay between Client and Server is 500ms both ways, the delay between pressing the fire button and seeing the tank fire is 1s

This is an unacceptable user experience for a combat action game. When developing **Tanks!!!**, we solved this problem in an iterative way. First, we opted to instantiate a projectile on the Client, and to also call a command on the Server to instantiate a projectile on all Clients. The problem with this approach is that there are now two projectiles on the Client that carries out the shooting action, separated by a distance of lag time multiplied by velocity. We added some metadata (an ID and a player ID) to the projectiles, which allows the Client to assess which two projectiles are effectively the same (the one instantiated locally, and the matching one instantiated on the Server). The Client is then able to reconcile the two, position the Server projectile at the same spot as the local projectile, then delete the local projectile.

When the projectile collides with something, it gets destroyed and creates an explosion. These collisions are handled by the Server (because it is authoritative), but the explosion visual still appears on the Client. Essentially, the Server sends a message from the Server saying “Explode”, and the projectile visually explodes on the Client. The problem is that the projectile on the Client is further along its trajectory when it receives the “Explode” message, so the visual of the explosion is in the wrong place (see Figure 14).

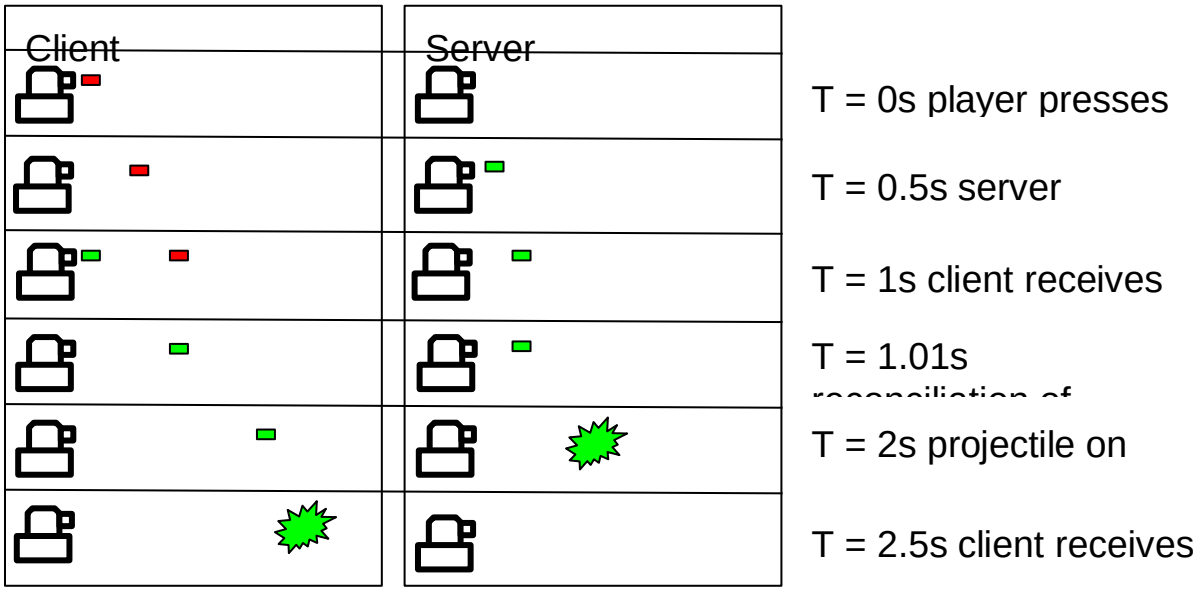

Figure 14: Timeline of a projectile using the initial approach to Client-side prediction

So, there are two projectiles on the Client: one created by the Client on button push (the **Local projectile**) and one created by the Server (the **Network projectile**). Instead of repositioning the **Network projectile** and deleting the **Local projectile,** we simply redefine their purposes:

1. **Local projectile** = the **visual projectile**, a visual representation of the projectile.
2. **Network projectile** = the **logical projectile**, a logic representation of the bullet as per the Server. This is where the explosion takes place.

The idea is that there are still two projectiles running on the Client, but the logical projectile is not visible - it has the sole purpose of ensuring that explosions appear in the correct place. This implementation is a lot more pleasing to the player, because the explosion appears in the correct place, albeit delayed. A problem with this implementation is that the visual projectile can effectively travel past the point the collision occurs, and then the explosion happens (see Figure 15).

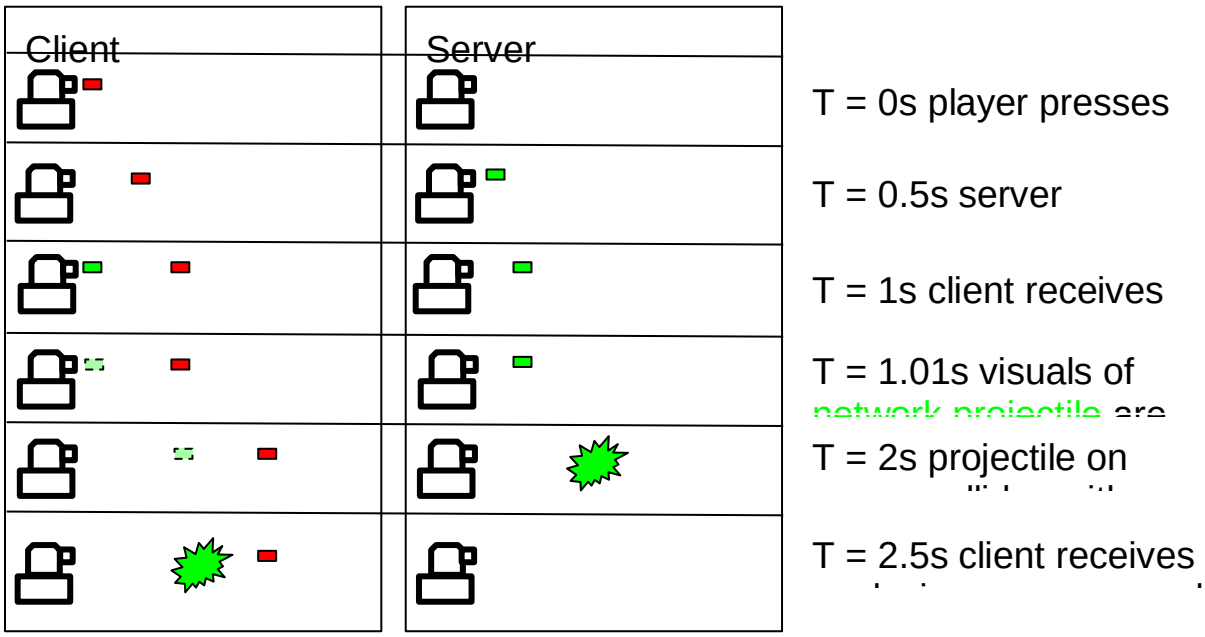

Figure 15: Timeline of a projectile using the second approach to Client-side prediction

We fixed this by handling the collision on the Client as well, without generating the explosion or any of the side effects (damage and physics), because these are handled by the Server. The Client-side collision detection simply deletes the visual projectile.

The final solution to the networked shooting came from the realization that the projectiles themselves do not do the damage - the explosions do. As such, we decided to make the *ExplosionManager* (Scripts/ExplosionManager.cs), which generates the explosions over the network, with the Server determining the position of the explosion.

Now, when the Client shoots, a **local projectile** is created on the Client. The Server then sends a message to all Clients to create the projectile, which the shooting Client ignores (effectively ignoring the network projectile). When the projectile collides, it is destroyed, and if the collision occurred on the Server, then the explosion is created at the correct location on the Server and all Clients. The final result is that the explosion is at the correct place on all Clients, but may appear delayed on the Client doing the shooting. This is a far better experience than waiting for your tank to shoot after you press fire.

Section 6 - Optimizing your game

Tracking various analytics allows you to see how your game is being used, and what changes to make in order to optimize the gameplay experience. Unity provides an easy-to-use framework for integrating analytics, and tracks some events without developers needing to write any code (assuming the Project has Unity Analytics enabled, as outlined in **Section 2.5: Analytics**). Unity Analytics allows you to track custom events with a simple function call: Analytics.CustomEvent.

To implement Analytics in **Tanks!!!**, we created a class called *AnalyticsHelper*, with static functions for wrapping the *Analytics.CustomEvent* call. Each static function is named according to the custom event that it wraps, and is called where appropriate. We gave each custom event sufficient parameters to provide enough information for analysis.

For example, when a player dies during a multiplayer game, this event is tracked. Tracking the occurrence of death does not provide enough information to make gameplay optimization decisions - you also need to know how and where the player died. The death event tracks the following:

- The player's tank type
- The killer's tank type
- The explosion type (which reveals which weapon was used)
- The map the death occurred on.

Given these parameters, you can find out whether a specific tank is unbalanced (for example, if the slow tank does a lot of killing, but hardly ever dies), and make the appropriate adjustments to balance the game.

Unity provides a data explorer as part of the Analytics Dashboard. This allows you (or an analyst on your development team) to see graphs based on various events. Figure 16, below, shows an example of a multiplayer kill event like the one described above, and shows that the most dominant tank is currently the one named **balancedTank** which is the default player tank. Assuming this example data was actual player activity, we could conclude that players were not discovering the other tank types (**heavyTank**, **fastTank**) and would need to make them more appealing. Perhaps they're not being discovered, or the **balancedTank** is far more fun to play. Analytics provide player insights, prompting us to make changes.

Figure 16: Plot of the *MultiplayerTankKilled* event comparing the killer tanks. The chart shows that **balancedTank** has the most kills

For a complete list of the custom analytics events tracked, consult *AnalyticsHelper.cs*.

Heatmaps are specialised analytics with Transform data (position, rotation and scale) and time data, allowing you to recognize problem areas with a specific level. To integrate heatmaps, we created a class with static functions called *HeatmapsHelper* to wrap the complete list of heatmap calls. Unity provides Heatmaps as a Unity package; to download the Heatmaps code and read the documentation, see the Unity Technologies BitBucket page.

Unity provides Performance Reporting as part of its services (see **Section 2.4: Performance Reporting**) and this can be viewed on the Developer Cloud portal on the **Project Dashboard**, under **Game Performance**. From Figure 17, you can see occurrences of issues in the game and use this information to make any required fixes.

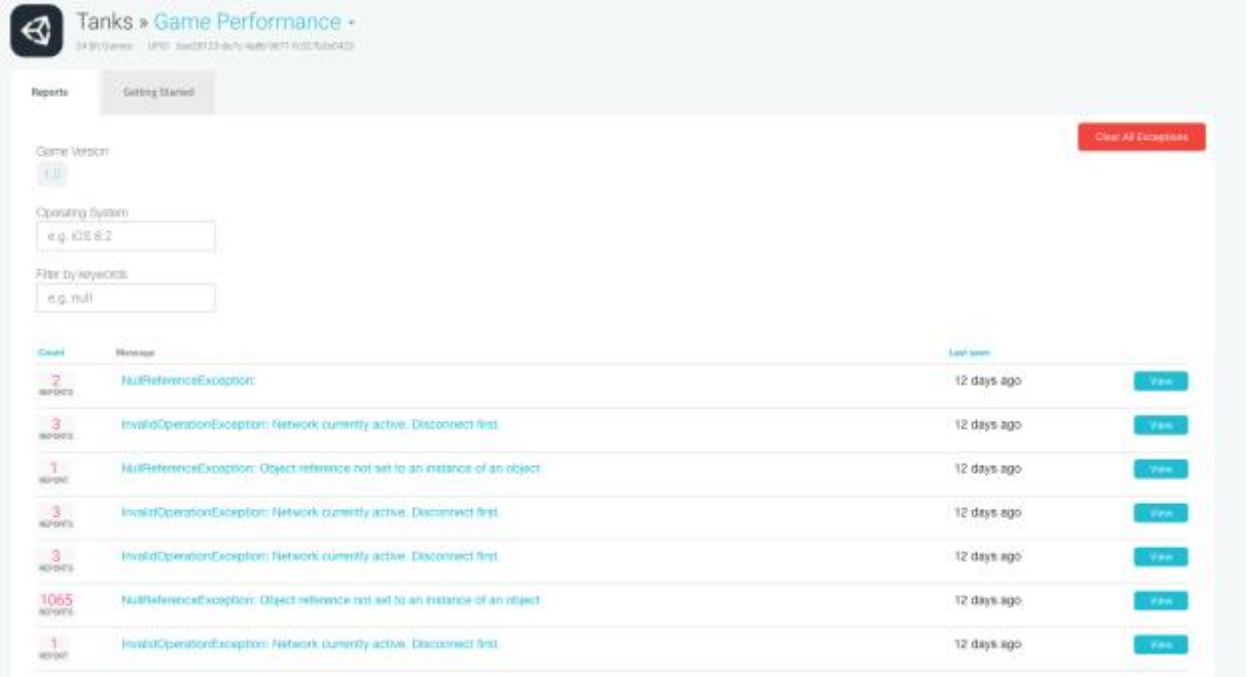

Figure 17: Game Performance dashboard showing the errors