Tanks Multiplayer

Documentation

V1.2

Scripting Reference

http://www.rebound-games.com/docs/tanksmp

1 Getting Started

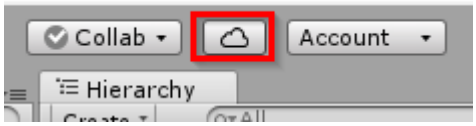
The following steps in this chapter describe how to enable a multiplayer service for use with this asset. Without enabling a multiplayer service, the game will not function. Therefore, please read at least the first chapter of this documentation.

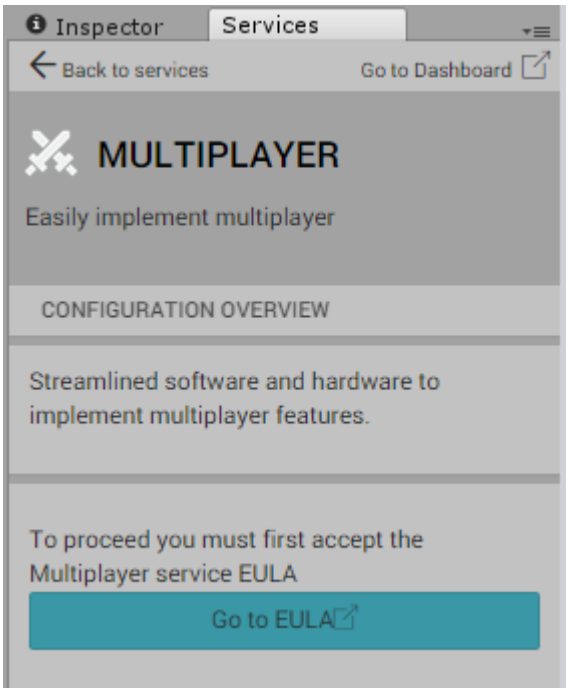
You can choose between Unity Networking (UNET) or Photon as a multiplayer and matchmaking service. For a detailed comparison about features and pricing, please refer to the official pages for UNET or Photon. The features of this asset are the same across both multiplayer services.

For a **video tutorial** on how to get started, please have a look at our YouTube channel.

1.1 Unity Networking (UNET)

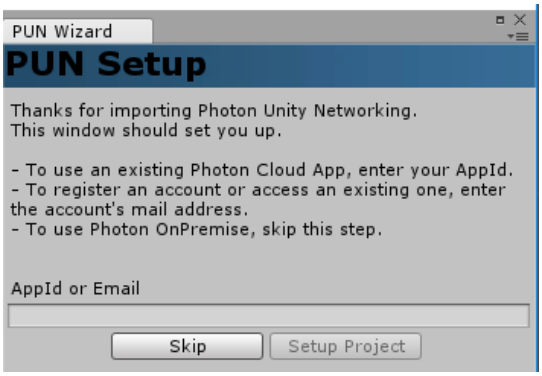
If you read this section, you've decided to go with Unity's Networking Service. In order to activate it, the first step is to open the **Services** panel in Unity and log into your project. If you haven't created a project in the services panel yet, please do so now. Next, select the **Multiplayer** service and click on the **EULA button**. You will be prompted to log in to your Unity account for accepting it. On the following page you need to set the maximum amount of players per room (we use 12 in this project). Also on the dashboard you can see other Multiplayer settings for tracking concurrent users or bandwidth for this project.





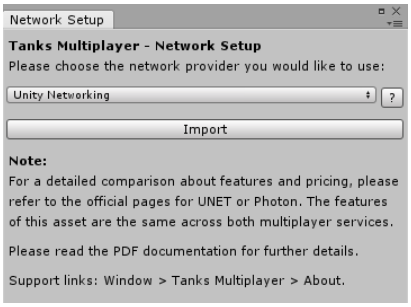
1.2 Photon (PUN)

If you read this section, you've decided to go with Photon as the networking provider. First, open the Unity Asset Store and import either Photon Networking Free or Photon PUN (in case you already own that). After the import finishes, set up your **AppID** for the Photon services in the popup windows that shows up. If it doesn't, navigate to *Window* > *Photon Unity Networking* > *PUN Wizard* > *Setup Project*. If you already have created an app for the Photon Cloud, you will find your app ID on the Photon Cloud dashboard.



1.3 Scene

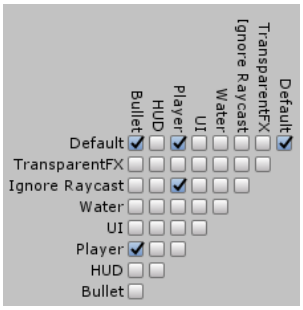
Also, each networking solution comes with its own **network components** only available for that particular networking scenario. There are several prefabs in the project, which require having those network components attached to them. For UNET, such a component would be the NetworkIdentity script. For Photon, there is PhotonView. We have created an **editor window** that automates the setup of all scripts, prefabs and components for you. Please navigate to *Window* > *Tanks Multiplayer* > ***Network Setup***.



First, you have to select the network provider you would like to use with this asset via the dropdown. After pressing the Import button, this window will automatically close itself.

You are ready to play the game now - load the 'Intro' scene!
Go ahead and build one or two instances to play against each other (or yourself).

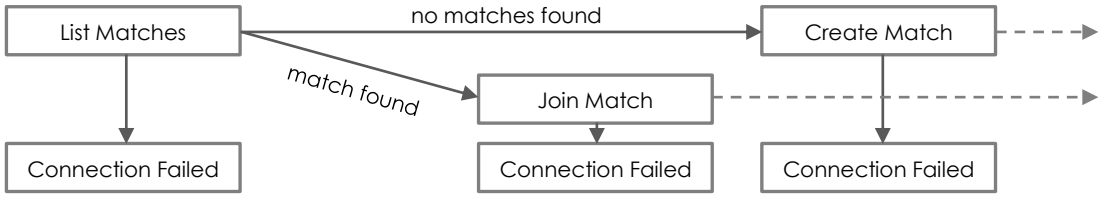
We will just take another minute here to go over the scene settings used in the **Game scene**. You can locate it in the project panel under *Tanks Multiplayer* > *Scenes*. In this scene, different layers are used for the environment, scene and player objects. This is done so that our customized **physics collision matrix** (*Edit* > *Project Settings* > *Physics*) ensures collisions between certain objects only occur where we want them to.



Default : static environment with colliders to create game world boundaries
TransparentFX: team spawn area boundaries, should not collider with anything
IgnoreRaycast: powerups, should collide with players but nothing else
UI: all visual game elements on the canvas, no collision
Player: only collides with environment or bullets, not with other players
HUD: visual player elements, hidden on player death, no collision
Bullet: only collides with environment or players, not with other bullets

2 Matchmaking Login

You may have noticed that in the **Intro scene**, there is a **Play button** that directly tries to enter an online game via matchmaking. The code for this lies in the NetworkManagerCustom script. If something fails during the connection, an error window shows up. In summary, the matchmaking workflow looks like this:



In the further course of this document and especially in network-related chapters, you will find additional **text boxes** that are labeled with either UNET or Photon. These are specifically directed at a networking solution, so you only have to read the one that is relevant to you.

UNET

On the NetworkManagerCustom script inspector, you can define some network specific settings, such as the maximum size of a match (under Network Info). Additionality, the player prefabs you are going to use for your game need to be assigned in the 'Spawn Info' section, under 'Player Prefab' and 'Registered Spawnable Prefabs'. Other networked prefabs (like bullets) do not go here, because these are being handled exclusively by our PoolManager during the game.

Photon

The NetworkManagerCustom script has some additional variables in the inspector to match UNET functionality, but also a constant variable hidden in code to define the amount of teams – named 'initialArrayLength'. You have to change its value if you want a different team count.

2.1 Game Settings

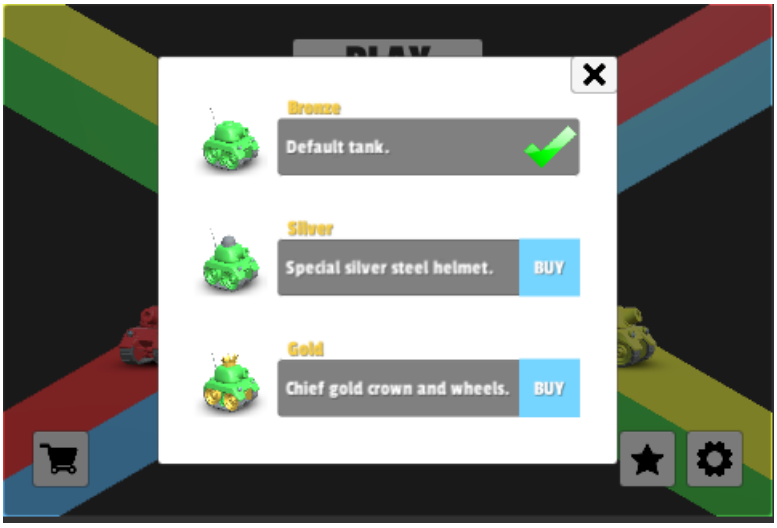
In the **Intro scene**, the player has the option to customize some game settings in the **Settings window** while running the game. This window contains an input field for entering the player's name, which is synchronized over the network and put over the player's model once connected. Also, selecting the desired network play mode is possible. Further in-depth explanation of the network mode implementation is listed in the Advanced section. Other settings in this menu only affect the local user, such as controlling music or sound effect volume.

Selected settings are being saved on the device at the point of closing the Settings window, then loaded on scene launch. The UIMain script handles all of this and provides default values in its Start() method (in case of first app launch) as well.

Right next to the Settings button you will find a button that allows users to **rate your app**. It does so by opening the App Store page for your app via your project's bundle identifier.

2.2 Player Selection

Your players can choose what model they want to play with. This happens in the **Shop window**, by selecting available **player models**. If you are selling models for in-app purchases, these models have to be bought first. You can find more details on billing with Unity IAP in the Services chapter.



The IAPProduct script, which is attached to each shop entry, handles the purchasing and selection state via a Toggle functionality for all player models within a ToggleGroup. In the inspector of an IAPProduct, a selection value needs to be defined to identify which model has been selected by the player at runtime. This value gets saved on the user's device. The current active model selection index along with the player name entered in the game settings are being sent to the host of a game, wrapped in a custom JoinMessage (extends MessageBase) class while connecting. As a result of this client-side request, the host knows exactly which player prefab to spawn by matching the selection value to the list of available player prefabs on the NetworkManagerCustom inspector and also the name of the player.

When the client connection to the 'Game' scene has been made, the host (via NetworkManagerCustom) finally assigns a team to the newly created player object. The logic behind this can be found in the GameManager script. In simple words: it iterates over all existing teams, tries to find the team with the lowest count of team members and assigns the new player to that team. More on what the GameManager does is described later in its own section.

3 Player

This chapter explains how the client and server handle **user input**, fulfilling **shot requests**, **registering damage** taken and **respawning players** upon death. Most of these things are done **authoritative** (where possible), meaning that the server should be in control about what happens next at all times. By sending requests to the server instead of executing them directly on each client, there is a higher barrier for cheaters to send fake events.

Each player prefab (located under *Tanks Multiplayer* > *Prefabs*) has the Player script attached to it, handling all of the networked user interaction. Please see its scripting reference for a detailed description of all public variables.

Photon

In order to instantiate prefabs over the network, Photon requires placing them in a folder called 'Resources'. Player prefabs are thus located under Tanks Multiplayer > Prefabs > Resources.

3.1 Movement

There are two different **input controls** implemented for moving the player object. On mobile devices operated via touch, one **joystick handle** is responsible for keeping track of the player's position relative to the camera (so the angle of the camera does not matter). The player position can be changed by dragging the joystick around. The exact movement speed and listening to the actual drag input events is done by the Player script, but the input direction changes are computed by the UIJoystick script that is attached to the joystick gameobject itself. This means that the Player script only takes this direction and applies it to the player's Rigidbody at the movement speed specified.

The second input control scheme is directed at Web Players and desktop platforms, i.e. non-touch devices. On these platforms, the joystick controls are hidden (except in the Unity Editor for debug purposes) and movement is controlled via **arrow keys** on your keyboard instead.

UNET

In order to synchronize player position changes over the network, player prefabs have a NetworkTransform component attached to them, listening to Rigidbody changes in 3D space. The synchronization happens at the sync rate specified, where usually 10-12 is a feasible rate.

TIP: To save some more bandwidth, you can disable the rotation syncing for axes that don't change in your game. In this asset for example, it makes sense to set the rotation axis on all NetworkTransform gameobjects (bullets and player prefabs) to Y-only.

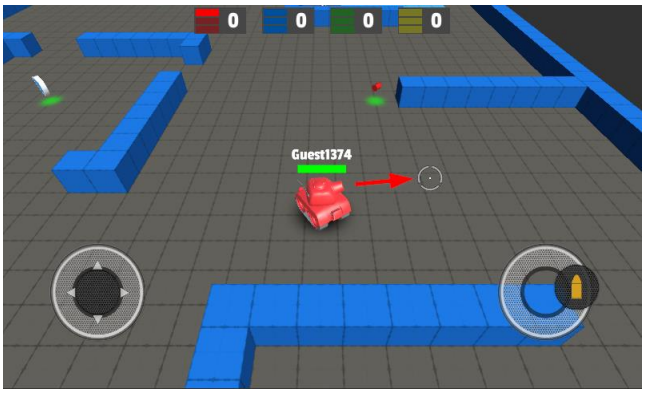
Photon

Player movement in Photon is being sent across the network by using their PhotonTransformView component attached to player prefabs. Similar to the UNET version, this script lerps between rotation updates to make them more smooth. The only difference is that the movement speed is not defined in the Player script, but in the inspector for this script directly. Also, a PhotonView component is needed on the prefab to observe the movement synchronization behavior.

3.2 Shooting

Just like movement, shooting has different **control schemes** for mobile/touch and non-touch devices too. On mobile platforms, there is **another joystick** on the screen that calculates an input direction for the Player script, but this time the direction is used to rotate the player's turret and for automatic shooting after a short delay. The delay is in place so that the user is able to position the turret before firing off the first bullet.

On non-touch devices, the **mouse position** on the screen is used to rotate the turret. Firing a bullet is triggered by clicking the **left mouse button** (or holding it down). To further illustrate the current turret rotation and shot direction, on desktop builds the mouse cursor has been replaced with a crosshair like icon, set under Edit > Project Settings > Player – Default Cursor. Since mobile devices don't have a cursor visible at all times, there is an **aiming indicator object** set on the UIGame script that is instantiated and attached to the player's turret at runtime:



In the event of shooting, a client processes its input in the Player script and determines to shoot a bullet. So it sends a shot request to the server. In our case we send the position of the bullet along with it. The server receives the request, processes it and spawns a bullet over the network. This is the signal for all other clients to spawn their own local copy of the bullet using our pooling system. At this point, a new bullet has been created successfully.

UNET

For shooting, we make use of a very common approach in UNET to send something along the network: Commands. Commands are methods with the [Command] attribute that can be used on clients to send events or data to the server. They can have parameters, but be aware of the bandwidth consumed.

Photon

Photon does not have a separate attribute like Commands (see the UNET version above) for sending data to the server. This is included under the [PunRPC] attribute which is explained in 4.1.

Only the server is responsible for determining whether a bullet hit a player or not. This is because even though collisions are registered on all clients, e.g. to spawn particle effects, only the server executes the logic for calculating player damage. It does so by getting the damage value from the bullet and applying it to the player's health value (or shield, if any). Read more about the type of these variables in the next section.

There are three types of different bullet prefabs in this asset, but they all have the same Bullet script attached to them. While there is endless ammunition for the default bullet a player can shoot, the other two bullets have to be collected via powerups and also have limited ammunition – read more about powerups in the Advanced section. The selected bullet is controlled by the currentBullet variable on the Player script. Changing bullets only means changing this index variable, as all available bullet prefabs are referenced in an array on the Player script per player prefab. This also means that different player prefabs can have different bullets, allowing for highly customized player prefabs and awesome gaming opportunities.

3.3 Properties

There are several **networked variables** on the Player script which are getting updated by the server whenever they change. Other players need to be **immediately aware** of variables (current health, shield amount, etc.) that change very often, but there are also variables where the latest update is **not that important** (e.g. turret rotation) or **not network synced** at all (e.g. max health, movement speed etc.). Different approaches are used to sync them over the network.

UNET

Commands have been explained already. Another way to distribute variable values is the [SyncVar] attribute, written before the variable declaration. Few SyncVars are exposed in the Player script's inspector, but there are also a few hidden ones. As per Unity scripting reference, SyncVars can only be changed by the server and are synchronized at least 10 times per second, if needed.

There is one mixed combination of Commands and SyncVars we are using this asset: turret rotation. As the turret is only controlled by the client, but SyncVars can't be changed directly on clients, the client sends a Command to the server with its updated rotation value. So now the server updates the turret rotation SyncVar with this value and distributes it across the network.

Photon

Photon introduces automatically synced variables that are very easy to use, but can be updated by anyone in the network. So for our authoritative network approach, we have to make sure that only the server accesses them. These synced variables are called Player Properties and exist for each networked player object/client. Changes are always synced immediately i.e. on the next network update. For event-specific changes (as with health, shield hits etc.) that don't change every frame, this functionality is of great use. All custom player property definitions and accessors are contained in its own class, the PlayerExtensions script.

In case something changes very often, Player Properties might not be the best fit. This is true for the turret rotation value fully controlled by the client, which could change constantly. For this scenario OnPhotonSerializeView comes in handy. This method is called 10 times per second on PunBehaviours to synchronize ever changing variables across the network.

4 Managers

Managers in Tanks Multiplayer are scripts which handle a core functionality of the game, providing access to its functions via static methods (so they can be called from any other script) or by getting direct access via their GetInstance() methods. We will go over all important manager scripts in the following sections to get an understanding of their various purposes.

4.1 Game Manager

Let's start with the managing script for the actual game logic. In the inspector for the GameManager script in the **Game scene**, you are able to define **highly important game aspects** such as the number of teams in the game, the color they should be visualized in, as well as the spawn area for each team and the maximum score count for one round. For a full description of each variable, please see its scripting reference.

Hidden from the public inspector variables, the GameManager also takes care of the score count and team fill for each team, stored in separate lists with their size equal to the team size. If a player died meaning a team scored, the respective value in the score list for that team goes up. The same happens for the fill list when a new player connected, but here it can also decrease on player disconnects. The last part is fully handled by the NetworkManagerCustom script, as it is aware of all player connection states. What the GameManager does now is to reflect these value changes in the UIGame by updating the labels and sliders responsible for showing current scores and team fill to the end user.

UNET

For storing multiple values in one variable, as done in arrays and lists, UNET offers the functionality of synced lists. The SyncList attribute is basically the same as a list of SyncVars. For syncing the score and team fill across the network, here we make use of SyncLists for the first time.

Photon

In the context of storing individual values for each player across the network, we've made use of Player Properties in section Properties already. Photon offers another type of networked storage for variables that are not bound to a specific player, but to the game (room) as a whole. This feature is called Room Properties. As with Player Properties, Room Properties are synced as soon as possible and is this matter they are perfect for storing scores and team fill of the game as done here. Room property definitions are contained in the NetworkManagerCustom script.

The GameManager script is also the main access point for checking whether the game has ended (in case the maximum score limit has been reached) and because it has a reference to the UIGame, it toggles displaying the player death and game over windows too. With the player respawn logic showing a delay timer or video ad in-between player deaths, that's basically everything needed for handling the game logic in one place. You can find more details on our video ad respawn workflow and Unity Ads in the Services chapter.

UNET

In this context we are now going to have a look at the last way of sending events or data across the network in UNET. Syncing variables and lists have been mentioned already, but there is another attribute complementary to Commands, called [ClientRPC]. Instead of getting invoked by clients and executed on the server (like Commands), ClientRPCs are invoked by the server and called on all clients (including the server).

This type of network call is used on the Player script, so that the server can tell all clients that a player got killed/respawned or that the game has ended. The latter is just an event without parameters for all clients, in terms of "the game is finished, your GameManager script can show the game over screen now".

Photon

Photon uses the same RPC approach like Unity Networking (see above), only the terminology is different. When using this type of RPC, authoritative methods that should be invoked on the server are tagged with the [PunRPC] attribute. Clients are then able to send request-like messages for execution on the server by raising a RPC call on a PhotonView component, providing the name of the PunRPC method. This principle is highly used in the Player script.

4.2 Audio Manager

The AudioManager script is attached to an indestructible gameobject in the 'Intro' scene living throughout scene changes, **handling long** (background music) **and short** (one-time effects) **audio playback** across the scenes. By using this manager we are offered with a unique component for playing all kind of AudioClips easily accessible via static methods. We don't have to provide AudioSource components in the scenes either, because the manager object has two of them - one for music, one for one-shot sounds) already attached to it. For one-shot clips which are usually instantiated and destroyed by default, we are also making use of our own object pooling behavior to reduce garbage collection even more. Except for background music clips which could be used across scenes and are referenced in the inspector of the AudioManager directly, all one-shot clips are referenced in their originating scripts (bullet sounds in the Bullet script, player sounds in the Player script etc.) instead.

4.3 Pool Manager

Pooling objects means **reusing gameobjects** instead of instantiating and destroying them every time when they are needed. As Unity's garbage collector is very performance-heavy and could result in serious hiccups and dropping frames during the game, this asset implements strict pooling (activating/deactivating) for gameobjects needed multiple times in one scene. Thus instantiate and destroy should be replaced by PoolManager.Spawn and .Despawn.

Setting up object pooling in your scene is very easy:
- have a container gameobject with the PoolManager script in the scene
- beneath this container, add one or multiple child gameobjects and name them as you wish
- add the Pool script to these child gameobjects and enter your configuration in the inspector
- if the prefab specified has a NetworkIdentity it is considered as networked pool automatically

Done!

5 Advanced

Since the previous chapters covered the basics on how this asset is structured and where networking code is used, this chapter goes into more detail on the advanced networking topics.

5.1 Network Mode

Right at the beginning in the settings menu of the Intro scene, you have the option to select a network mode you want to play with. Each network mode uses a completely different network layer, but they are all handled internally by the NetworkManagerCustom script automatically, so that players do not get confused with difficult interfaces or additional buttons such as entering IP addresses (except on Photon LAN) and so on.

Online: selected by default. This is using the cloud matchmaking servers for automatic match creation and joining as described earlier in this document, which is the simplest way to get right into a battle and play against other players. The timeout for finding and joining a match is set to 10 seconds, which could be reached due to connectivity or device networking issues.

LAN: local area network mode does not forward traffic to the cloud servers, but rather keeps it locally in a private network, where only other devices in the same network are able to host and join a battle. A common use case is hosting a game that can only be joined by your friends.

UNET

This asset supports local discovery using Unity's NetworkDiscovery component in order to detect other devices in the same network and join a game automatically. If no match can be found after exceeding the timeout value, then the searching device creates a new LAN match itself. Note that on Windows/Mac LAN mode requires disabling firewalls for detecting other devices.

Photon

Please note that with Photon there is no explicit LAN mode. You have download and run a Photon Server on your own machine/server, connected to the internet, in order to set up a private network. You can find out more about this process here. If you select LAN mode in the Intro's settings menu, an additional InputField will show up so you can enter your server address.

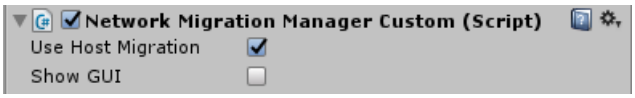
Offline: playing against bots. This mode is simulated by creating a private LAN game but without making it public for other devices. The big advantage of this is that there are no changes to existing network code necessary. Bots are controlled by the PlayerBot class, a derived script of the Player class, which uses Unity's navigation system for movement. They are then getting spawned by the BotSpawner after the player joins the game scene.

5.2 Host Migration

Host Migration describes a concept where a game **continues to run** with minimal interruption when the **match host left**. The technique behind this is that after the host disconnected, one of the connected clients takes over the role of hosting the game to keep it alive. The disconnected host could even join the same game again as a client later. By using host migration, terminated games because of temporary lost connections and host rage quits should be prevented. For using host migration efficiently, it should be noted that when switching hosts, only data that is available on the client becoming the new host will be kept in the game. This in turn means that any data that is only available on the server is lost after a host migration. So you should plan in advance what data should be synced across all clients and build your client/server networking code around that principle.

UNET

Unity Networking offers a NetworkMigrationManager component enabling host migration. Please read a detailed summary containing its workflow on this page. The default component does not provide automatic migrations without UI buttons though, so we've created a new script based on the NetworkMigrationManager named NetworkMigrationManagerCustom. It is attached along our NetworkManagerCustom script in the 'Intro' scene. You can toggle the use of host migration by enabling the checkbox on it.
Note: UNET does not support host migration via online matchmaking yet, only for LAN games.



Photon

Host migration is supported by default on all platforms, but certain limitations still apply, such as keeping data in sync between all clients for a successful migration. Since we mostly use player and room properties which can be accessed by all clients, we don't run into issues in this regard.

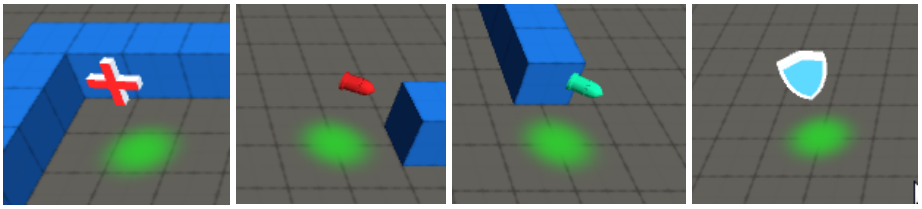
5.3 Powerups

Powerups are objects that can be collected by players to **boost their effectivity** in the game. The Powerup script is working like a **template** for different powerups, such as for the PowerupBullet or PowerupHealth scripts. If you would like to create new powerups, simply extend your script from Powerup and override its Apply method. This method is responsible for what to do when a player collected the powerup, so all of your powerup logic should go in here.

Prefabs for all types of powerups are located in the project panel under *Tanks Multiplayer* > *Prefabs*. Just like bullets, powerups have their own Pool in the Game scene making use of our PoolManager in their spawn behavior. The actual spawning instruction is coming from an ObjectSpawner in the scene, which lets you define the exact position and respawn delay.

Networked variables in this asset (see chapter Properties) are compatible with host migration already, as they are being distributed to all clients. Powerups are a great example of something that is **not compatible** with host migration **since the beginning**, because they rely on timed gameobject states i.e. hierarchy changes (active/deactive) in the scene. The way we are syncing gameobject states (and respawn timer) is via the managing ObjectSpawner script. In this case, we have three possible outcomes to synchronize:

- ✓ **Active powerups** should be visible to all clients: each ObjectSpawner and Powerup instance have a NetworkIdentity component attached, so all clients are aware of their existence in the scene already. Their relationship is saved in a NetworkInstanceId on the Powerup script. When the host spawns a powerup, each client stores a reference to the local gameobject it just spawned on the ObjectSpawner.
- ✓ **Inactive powerups** should be invisible to all clients: despawn instructions over the network are handled by our PoolManager automatically. Joining clients will create an inactivate object by themselves due to how UNET handles NetworkIdentity objects.
- ✓ **Current respawn timer**: once a powerup has been collected, the server then runs a coroutine on the ObjectSpawner waiting for the respawn delay to pass. If the host disconnects now, the new host would not know where it left off i.e. when to respawn the powerup. Because of this, the ObjectSpawner saves the remaining respawn time on all clients on despawn. This ensures that we are fully aware of all host migration scenarios.



6 Unity Services

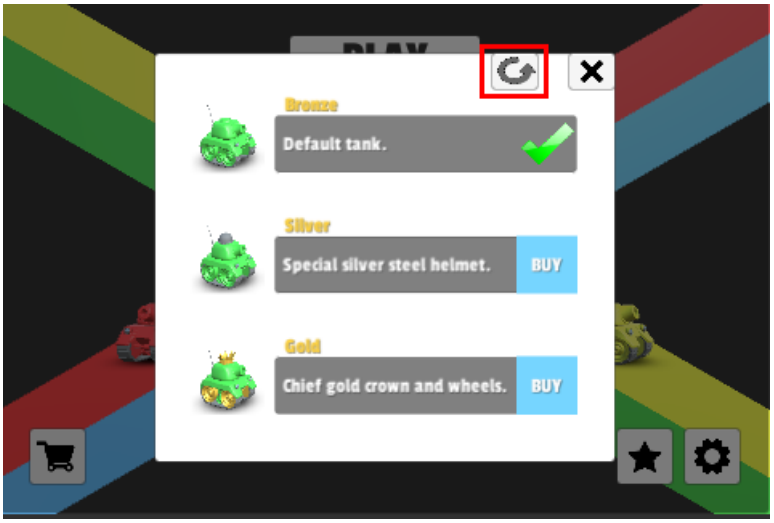
According to their own landing page, Unity offers a **range of services** to help game developers engage with their players, understand how they are playing your app and monetize them in the most efficient way. In this asset, we've tried to integrate as many services as possible, in the areas it made the most sense. **Note that none** of the following services **are required** to run the game. You can just turn them off in Unity's in-editor services panel, and nothing will complain.

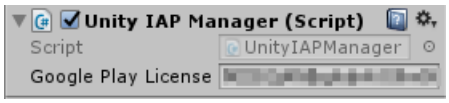
6.1 IAP

This asset makes use of Unity IAP and integrates a very basic implementation for in-app purchases. If you would like to look into a fully featured shop system with real money and virtual currency purchases, please have a look at our Simple IAP System asset on the Unity Asset Store. Unity IAP requires Analytics to be enabled too. It will get automatically turned on if you try to enable Unity IAP. How to enable Unity IAP is explained here.

For in-app purchases to work, you must first create an app in your respective App Store account, then create products there. Note that depending on the App Store you are deploying to, certain limitations apply when testing in-app purchases. For example, Google Play requires your app to be submitted and live as alpha/beta version (not draft).

With your app and products created in the App Store, open the Intro scene and have a look at our IAPProducts shop items in the inspector again. They have an identifier and type defined in there. This should be your identifier and type used when creating your products on the App Store. You will also notice that when opening the shop window on iOS, there is an additional button at the top (highlighted in the following screenshot). This button handles restoring purchases, because as per Apple requirements, your app would get rejected without having it.





The UnityIAPManager script handles initiating and processing of in-app purchases. For Google Play, the developer license key should be entered in its inspector. If you are using shop entries, like with the IAPProducts showcased in this asset, you already have a buy button linked to them in the UI. If you would like to initiate a purchase directly via code instead, you can do so by calling UnityIAPManager.PurchaseProduct. In this case you will have to add the product identifier to the Purchasing builder in the Start() method manually though.

6.2 Analytics

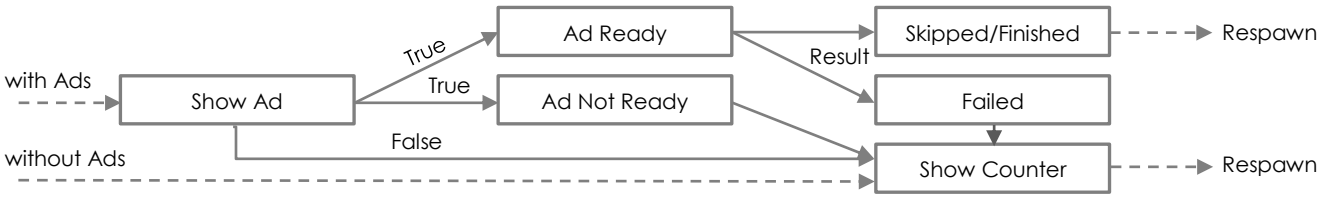
Planned for an update.

6.3 Ads

How to enable Unity Ads is explained here (you can skip the coding parts).

Instead of pushing users to in-app purchases, video ads (or ads in general) are a great alternative to monetize Free2Play games. Especially not being intrusive and limiting the ad amount per user session have been proved to maximize ad revenue. In this asset, when a player dies a respawn counter gets displayed. If Unity Ads is enabled, we are replacing the respawn counter with a video ad once in every round. One ad per round (~0.5 Cent/user) should be sufficient to pay the server bill for that round. The percentual chance for showing a video ad increases on each player death, until one ad has been shown after a maximum of 6 attempts/deaths. See the UnityAdsManager script for detailed comments on the ad chance calculation. The scenario fully handles failed ad requests too.

To summarize, the video ad workflow on player death looks like this:



6.4 Everyplay

With the use of Everyplay, your players can record their gameplay and share it to common social networks or the Everyplay hub, to build a gaming community around your game and go viral. At the time of writing this documentation, Everyplay is not part of the Unity services catalog directly from within the editor, but available on the Asset Store as a separate plugin.

How to import and enable Unity Everyplay is explained here (skip the coding parts).

You have probably noticed that there is another button visible in the Intro scene, maybe only when the game is not running. This is because Everyplay turns itself off if it is not supported on the device. Our UI reacts to that and if not supported, doesn't show any Everyplay buttons at all.

The button in the scene is an access point to the Everyplay community hub, which lists recorded gameplay videos for your game as well as some others. Adding the community hub is a requirement in the guidelines published by Everyplay. In the Settings window, there is a new checkbox visible on supported devices as well – here your users can toggle the automatic recording of gameplay videos (no worries, sharing them is still a manual step by the user). It is enabled by default. The manager for all of this functionality is called UnityEveryplayManager.

An advanced Everyplay feature is rendering a thumbnail of the recorded gameplay video during the game. We've integrated this for the game over screen and combined it with the sharing dialog as a big rectangle button so your users should not be able to miss it.

7 Contact

As full source code is provided and every line is well-documented, please feel free to take a look at the scripts and modify them to fit your needs.

If you have any questions, comments, suggestions or other concerns about our product, do not hesitate to contact us. You will find all important links in our 'About' window, located under *Window > Tanks Multiplayer*.

If there are any questions, maybe our support forum or Unity thread has the answer!

For private and/or open questions, you can also email us at
info@rebound-games.com

If you would like to support us on the Unity Asset Store, please write a short review there so other developers can form an opinion. Again, thank you for your support, and good luck with your apps!

Rebound Games