Tanks Multiplayer

Documentation

V1.6

Scripting Reference

http://flobuk.com/docs/tanksmp

1 Getting Started

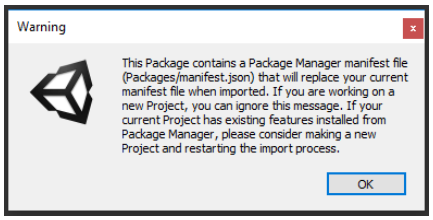
The following steps in this chapter describe how to enable a multiplayer service for use with this asset. Without enabling a network service, the game will not function. Therefore, please read at least the first chapter of this documentation.

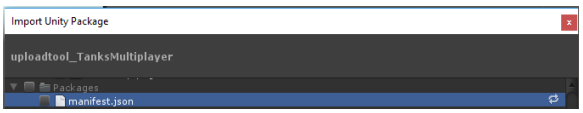
You can choose between Unity MLAPI, Photon or Mirror Networking as a multiplayer and matchmaking service. For a detailed comparison about features and pricing, please refer to the official pages for MLAPI / Photon / Mirror.

For a (dated) **video tutorial** on how to get started, please have a look at my YouTube channel.

1.1 Package Manager

When importing the asset, you will be prompt with a warning that your Package Manager dependencies might be changed. Deselect the manifest.json file on import, if you are importing the asset into an existing project (which already contains your own manifest).





1.2 Unity MLAPI

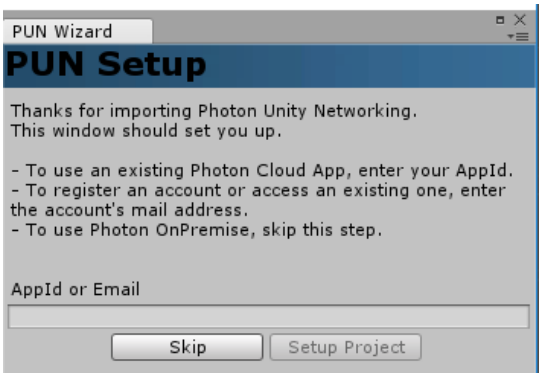
If you read this section, you've decided to go with Unity's new game-object oriented networking service. Please note that it is still experimental, so the integration is labeled as "beta" too.

MLAPI will at some point be available in the Package Manager. While being experimental, you have to add it to your project manually. Please refer to this page for installation instructions.

1.3 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 PUN 2 FREE or Photon PUN 2+ (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.



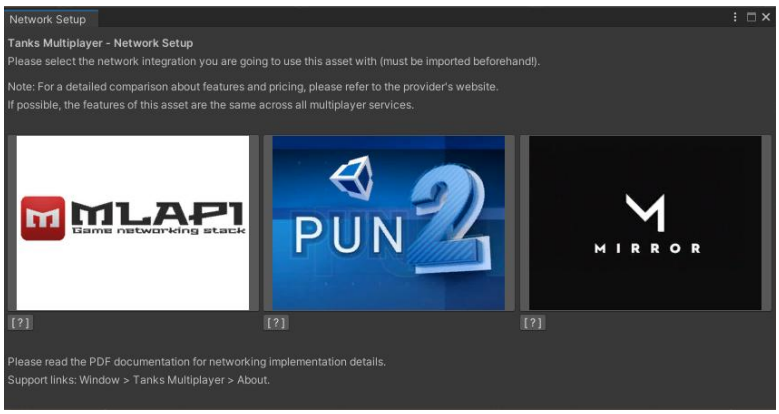
1.4 Mirror Networking

So, you've decided to go with Mirror Networking, a community development of UNET, as your networking provider. Mirror is available for free on the Unity Asset Store. Download and import it into your project.

1.5 Project Setup

After importing the networking solution of your choice, it is time to import the corresponding files of Tanks Multiplayer. Each networking solution comes with its own **network components** only available for that particular networking scenario. Meaning you cannot easily switch later.

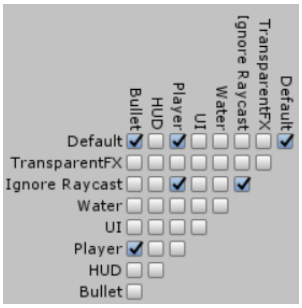
There are several prefabs in the project, which require having those network components attached to them. As an example: for MLAPI it is NetworkObject. For Mirror that would be the NetworkIdentity script and for Photon, there is PhotonView. In order to import the networking scripts please navigate to *Window* > *Tanks Multiplayer* > ***Network Setup***.



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).

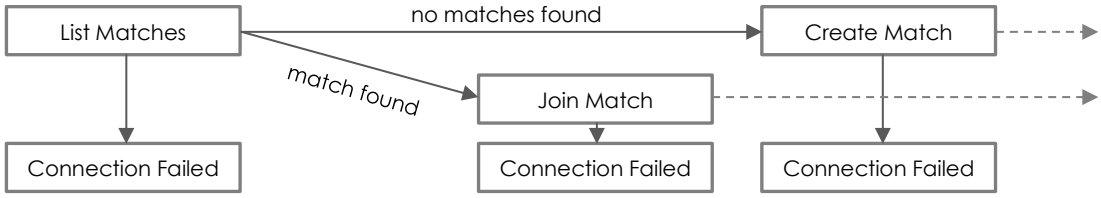
We will just take another minute here to go over the scene settings used in the **Game scenes**. You can locate them in the project panel under *Tanks Multiplayer* > *Scenes*. In these scenes, different layers are used for the environment, scene and player objects. This is done so that our customized **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 collide with anything
IgnoreRaycast: collectibles, should collide with players and collectible-zones
UI: all visual game elements on the canvas, no collision
Player: collides with environment, collectibles 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** like MLAPI, Photon or Mirror. These are specifically directed at a networking solution, so you only have to read the one that is relevant to you.

MLAPI/Mirror

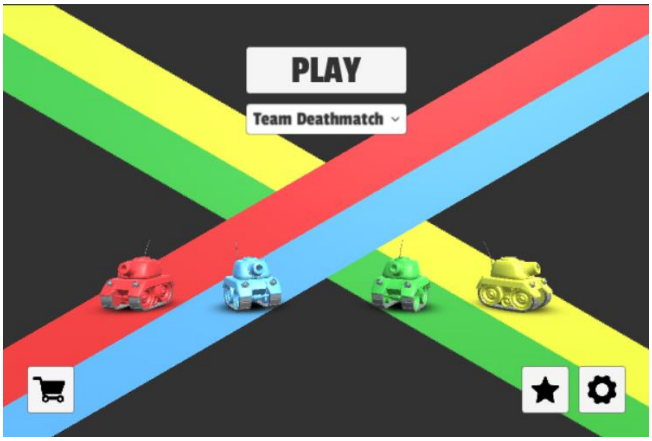
On the NetworkManagerCustom script inspector, you can define some network specific settings, such as the maximum size of a match (under Network Info). Additionality, the player prefabs you are going to use for your game need to be assigned in the 'Spawn Info' section, under 'Player Prefab' and 'Registered 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 variable in code to define the amount of teams for initialization - named 'initialArrayLength'. You have to change its value in the OnCreatedRoom() method if you want a different team count per game mode.

2.1 Game Settings

In the **Intro scene**, below the Play button, you'll find a dropdown for selecting the game mode to be played. Currently, this asset supports Team Deathmatch, which is the default mode, as well as Capture The Flag. Each game mode creates its own list of lobbies for the matchmaking service, so that players can only the game mode they're interested in. Be cautious about adding too many game modes: if your population is low, games might end up not getting filled.



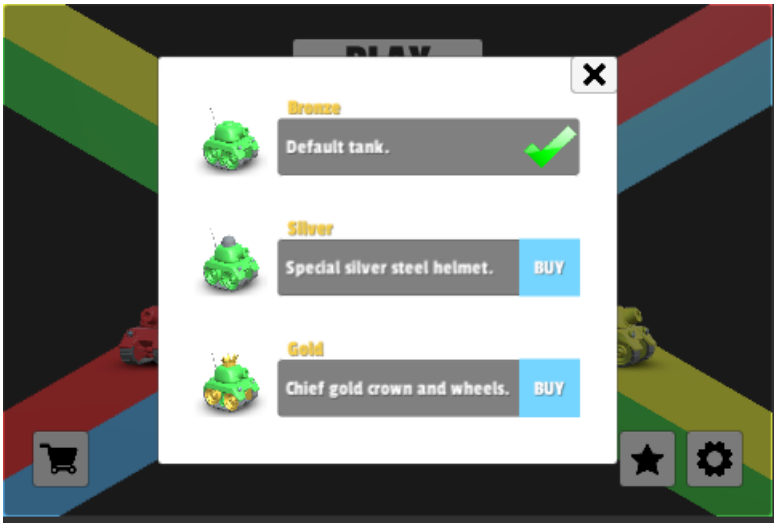
At the top right bottom, the player has the option to customize some game settings in the **Settings window** while running the game. This window contains an input field for entering the player's name, which is synchronized over the network and put over the player's model once connected. Also, selecting 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 in this window are being saved on the device at the point of closing it, then loaded every time 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.

MLAPI/Mirror

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.

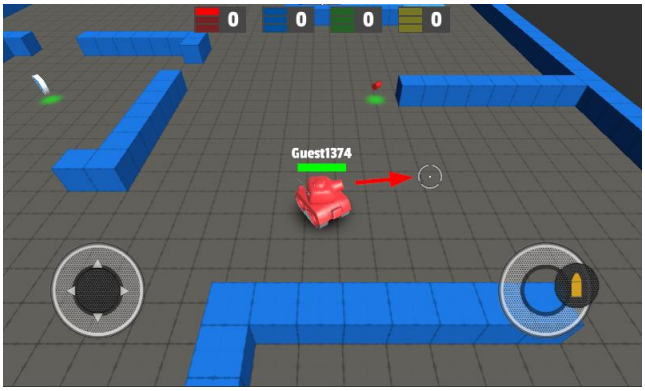
Photon

Player movement in Photon is being sent across the network by using their PhotonTransformView component attached to player prefabs. Similar to UNET/Mirror, 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.

Mirror

For shooting, Mirror makes use of the old Unity Networking approach 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.

MLAPI/Photon

In MLAPI and Photon, calling methods to be executed on the server is not called a Command (see above). Instead, this is called a Remote Procedure Call (RPC). Photon uses the [PunRPC] attribute for that and MLAPI uses [ServerRpc] which is further explained in section 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.

MLAPI

MLAPI features automatically synced variables that are very easy to use, but can be updated by anyone in the network. By default, NetworkVariables can only be updated by the server. All variables that are treated like a state or snapshot should be stored in NetworkVariables.

Mirror

NetworkVariables in Mirror are defined using 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

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 now does is to reflect these value changes to the UIGame by updating the labels and sliders responsible for showing current scores and team fill to the end user.

MLAPI

For storing multiple values in one variable, as done in arrays and lists, this networking solution offers the functionality of saving NetworkVariables in containers such as a NetworkList. For syncing the score and team fill across the network, here we make use of NetworkLists for the first time.

Mirror

When syncing lists of variables in Mirror, you can make use of the SyncList attribute. 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.

MLAPI

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

Mirror

Similar to MLAPI but with different naming:
Method called on clients to be executed on the server: Command
Method called on server to be executed on all clients: [ClientRPC]

Photon

Photon uses the same RPC approach like Mirror (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 has a NetworkIdentity/NetworkObject it is considered as networked 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 trying to use 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.

MLAPI/Mirror

Unfortunately, both solutions do not provide matchmaking servers by default. Mirror supported this once with their ListServer implementation, but it was removed at some point. For MLAPI this is just not ready yet. If you are looking for free online matchmaking, I would recommend checking out the Steamworks or Epic Online Services transports for use with them.

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.

MLAPI/Mirror

This asset supports local discovery using the 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.

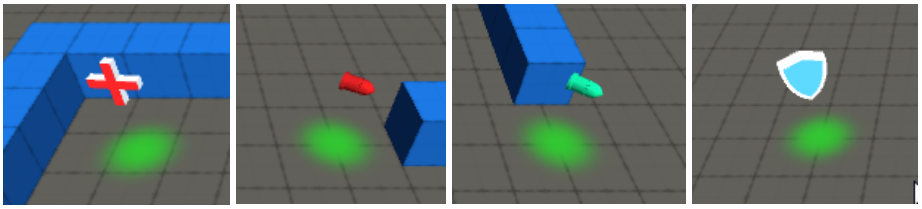
MLAPI/Mirror
Host Migration is not supported by MLAPI or Mirror yet.

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 Collectibles

Collectibles are objects that can be consumed by players to **boost their effectivity** in the game. The Collectible 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 Collectible and override its Apply method. This method is responsible for what to do when a player collects the powerup, so all of your collectible logic should go in here.

Prefabs for all types of collectibles are located in the project panel under *Tanks Multiplayer* > *Prefabs*. Just like bullets, collectibles 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. Collectibles 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/inactive) 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 Collectible instance have a unique network component attached, so all clients are aware of their existence in the scene already. When the host spawns a powerup, each client stores a reference to the local gameobject it just spawned via 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, instructed by the master client or host migration scenario.
- ✓ **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.

MLAPI

Collectibles have a NetworkObject component attached to them, for making the network aware of the gameobject. Their relationship to its ObjectSpawner is then saved using NetworkObjectId in a NetworkVariable. We do this because when networked objects are spawned for other clients, it does it without clear allocation. By using and syncing the NetworkObjectId across all clients, we can match each Collectible with its corresponding ObjectSpawner.
Due to the way spawning of NetworkObject objects is handled for joining clients, we do not have to explicitly cover enabling/disabling powerups as this is done automatically.

Mirror

Similar to MLAPI, Collectibles have a NetworkIdentity component attached to them. Their relationship to the ObjectSpawner is saved in a NetworkInstanceId.

Photon

In Photon, the only object that needs a PhotonView is the ObjectSpawner. Each Collectible instance can then be handled locally on the client. The relationship between Collectibles and their spawner is set locally too, directly via the ObjectSpawner that instantiates the powerup.
Syncing of Collectible states for joining clients or a new master client is covered via Photon's callbacks:
OnPlayerEnteredRoom: the master client sends whether the object should be instantiated or spawned later
OnMasterClientSwitched: the new master client decides whether the spawn routine needs to be run

5.4 Game Modes

Each game mode has a separate matchmaking queue, as described in the Game Settings section. Within the specified game mode, the map loaded for a new room is random. This works by **defining** game modes and **assigning** scenes.

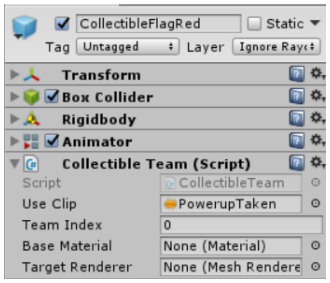
```
public enum GameMode
{
    TDM,
    CTF
}
```

If you would like to create a completely new game mode, first you have to **define** it in the GameManager enumeration named GameMode.

In order to **assign** scenes to that game mode for the matchmaking service, your scenes then have to follow the GameMode naming convention. Meaning for a Team Deathmatch scene, the scene should start with "TDM".

In addition, only scenes are chosen which are added and active in Unity's Build Settings. This allows for testing new maps without affecting live builds. Finally, in the scene's GameManager inspector, select the GameMode the map was designed for.

Since Team Deathmatch is the default mode, the following section only describes the differences to **Capture The Flag (CTF)**. CTF is based on Collectibles. Powerups mentioned previously are consumed directly on collision, but Collectibles can actually be picked up, dropped and returned too! All of this is handled by the managing ObjectSpawner as well.



The flags in CTF have the CollectibleTeam script attached to them, which is responsible for checking the player's team before executing the desired logic: pick up enemy flag, return own flag if dropped.

In this regard, the "Team Index" variable assigned in the inspector is the team that owns this Collectible.

For the sake of covering several different network approaches in this asset, a new type of synchronization is used for the pickup/drop/return of Collectibles.

MLAPI

The CollectibleState struct defines the Collectible's NetworkObjectId, its parent Id and position. In order to synchronize it across the network, the struct needs derive from INetworkSerializable. The actual list implementation is defined in the GameManager, in a NetworkList, which is automatically synced to new players.

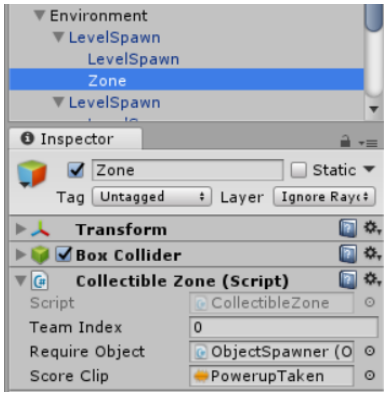
Mirror

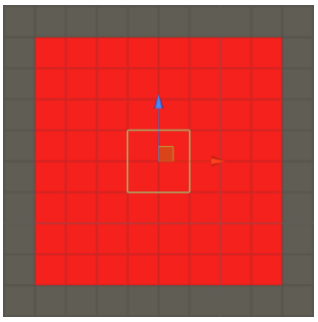
We are using another type of SyncList: SyncListStruct. The inheriting class and struct are defined in the ObjectSpawner (SyncListCollectible, CollectibleState), whereas the actual list implementation is again defined in the GameManager. Just like SyncLists, SyncListStruct is automatically synced to new players.

Photon

Buffered RPCs are like regular RPCs, but remembered on the server and executed for newly joining players. We make use of this concept for collectibles by just calling a regular RPC method on the target ObjectSpawner and passing in RpcTarget.AllBuffered on the call. Depending on the action executed (pick up or drop), an additional variable (carrier or position) is then provided to all clients as well. An exception is the RPC to return a dropped Collectible, which is not buffered: having the Collectible at its original position is the default, so we don't have to explicitly save that for connecting players – the Collectible will spawn there either way. Every time a new buffered RPC is being sent, we remove all prior RPCs related to that PhotonView.

With our implementation for Collectibles, we wouldn't want to store an infinite amount of entries every time a player picks up or drops a Collectible though – just the most recent action. For this, we are searching for an existing entry about it first and then modify that. Note that the buffered RPCs for Photon are a sample for demonstrating a different networked approach. In this case, since we only store the most recent RPC, the same functionality could also be achieved easily by storing the necessary variables on the Collectible itself, and letting the clients request them from the server via a single RPC call.





With the networked functionality for pick ups done, we still need a way to detect if a player has scored by getting the enemy flag to its own base. In the CTF mode, we have added the CollectibleZone component for this reason.

The CollectibleZone component, including a BoxCollider for detecting player or collectible collisions, is attached to a gameobject in the scene, defining the home base for one team. If the variable "Require Object" is assigned with an ObjectSpawner, that specific object needs to be present at its home base before the player can score. In the CTF scene and the screenshot above, this means that Team=0 (Red Team) can only score by getting the blue flag, if the required object (their own - red flag) is at their base. With the "Require Object" unassigned, players could also score when someone else has taken their own flag.

6 Unity Services

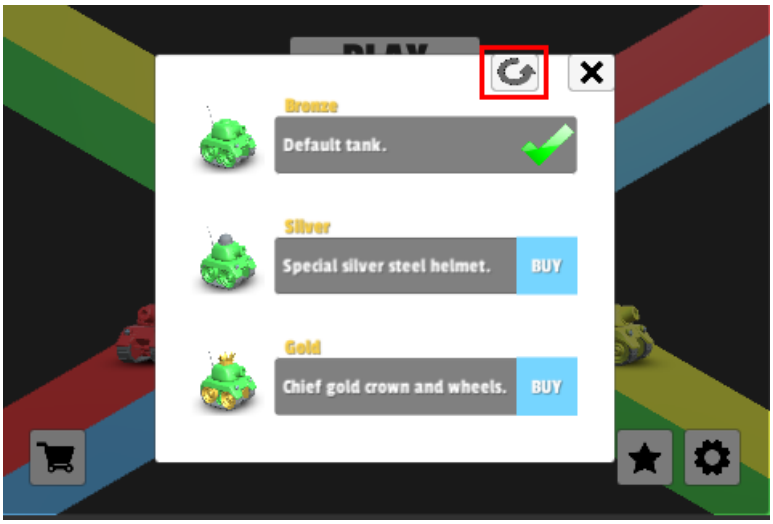
According to their own landing page, Unity offers a **range of services** to help game developers engage with their players, understand how they are playing your app and monetize them in the most efficient way. In this asset, we've tried to integrate as many services as possible, in the areas it made 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.

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. If you are using shop entries, like with the IAPProducts showcased in this asset, you already have a buy

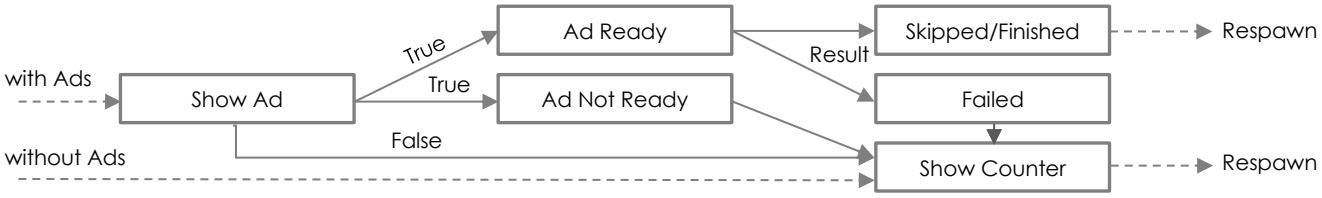
button linked to them in the UI. If you would like to initiate a purchase directly via code instead, you can do so by calling UnityIAPManager.PurchaseProduct. In this case you will have to add the product identifier to the Purchasing builder in the Start() method manually though.

6.2 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:



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 me. You will find all important links in the 'About' window, located under *Window > Tanks Multiplayer*.

If you would like to support me 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!

FLOBUK