TopDownShooter-ToolKit

Documentation for Unity3D

Version: 1.3.2 f7
Author: K.Song Tan
LastUpdated: 5th January 2021

Forum: http://goo.gl/HVsZQ9
WebSite: http://songgamedev.com/tdstk
AssetStore: http://u3d.as/mqT

Thanks for using TDSTK. This toolkit is a collection of coding framework in C# for Unity3D, designed to cover most, if not all, of the common top down shooter ("TDS") game mechanics. Bear in mind TDSTK is not a framework to create a complete game by itself. It does not cover elements such as menu scenes, options, progression saving etc.

The toolkit is designed with the integration of custom assets in mind. The existing assets included with the package are for demonstration. However, you are free to use them in your own game.

If you are new to Unity3D, it's strongly recommended that you try to familiarise yourself with the basics of Unity3D. Once you grasp the basics, the rest should be pretty intuitive. Almost all of the editable elements of the toolkit come with tooltips in the editor. You can just hover over the label to get an idea of what a particular setting is about. For that reason, most of this documentation only explains the general concept of the toolkit and things that are either less obvious or aren't included in the tooltip.

You can find all the support/contact info you ever needed to reach me on '***Tools/TDSTK/Support And Contact Info***' via drop down menu from the Unity Editor. Please leave a review on the Asset Store if possible; your feedback and time are much appreciated.

Important Note:

If you are updating an existing TDSTK and don't want the new version to overwrite your current data setting (units, weapons, collectibles, etc), Just uncheck '*TDSTK/Resources/DB_TDSTK*' folder in the import window.

OVERVIEW

General

Basic components. All component are optional with the exception of GameControl and UnitPlayer. Special case being AbilityManager, which is required if the player uses any ability.

- **GameControl/DamageTable* -** control all the general game logic (life, hit logic, etc.)
- **ObjectiveTracker*** – a component to track and the objective in the level
- **UnitPlayer*** - the component on player, handle everything the player do
- **PlayerProgression** – the component that hold and process the level information of player unit
- **PlayerPerk** – the component that hold and process the perk information of player unit
- **AbilityManager*** – component that govern player's ability
- **DropManager*** – handle the spawning of collectible(bonus) item when hostile unit is destroyed
- **UnitSpawner** – the component that handle procedural spawning of additional AI unit
- **CollectibleSpawner** – the component that handle procedural spawning of collectible item
- **Trigger** – component that is used to trigger various event when activate by player unit
- **TDSArea** - a component that is used to define effective area for various spawner/trigger

*There cannot be more than one of these component in the scene at any given time

Runtime component: - These are in game object that can be pre-placed or spawned during runtime

- **UnitAI** – component that drive each individual AI unit
- **Weapon** – the weapon of the player, spawned when the level is loaded
- **ShootObject** – the 'bullet' object fired by all units
- **Collectible** – item that can be collect by player for various bonus effect

Support components: - These secondary supports the mechanic of the game so that it's playable

- **CameraControl** – handle the in game camera
- **AudioManager** – component that handle the audio aspect of the game
- **UI** – a series of component that handle the User interface

Layers

TDSTK uses Unity layer system to identify various objects in the scene. Unfortunately the layer setting doesn't get imported to your project automatically. The system will still work but to avoid confusion it's recommended that you name the layer accordingly.

The layer used by default are:

- layer 8: 'Unit_Player'
- layer 9: 'Unit_AI'
- layer 10: 'ShootObject'
- layer 11: 'Collectible'
- layer 12: 'Trigger'
- layer 13: 'Terrain'

You can find the definition of these layers in TDS.cs.

Gizmos

There are gizmos as visual aid when placing various spawners and triggers in a scene. However to enable this, you will need to manually move the folder '***TDSTK/Gizmos***' to the root directory of the project (just move it out of *TDSTK*).

Demo And Examples

The framework comes with sets of example scene and prefabs that show case how the toolkit works. These scenes are what you would have seen in the playable web demo. The scenes can be located in '***TDSTK/Scenes/***'. There's another sets of similar scenes with the UI and input set to be mobile ready located in '***TDSTK/Scenes/Mobile***'.

BASIC CONCEPT:

Unit Player: In-Game Object

The unit controlled by the player. You can assign multiple weapons and abilities to a player unit, which can be switched during runtime. The player's primary attack will depend on the weapon selected.

Unit AI: In-Game Object

Unit AI is pretty much any unit (not necessarily hostile) that can be attacked by the player in the game. They can be setup with various behaviors and attributes to accommodate various roles (a stationary neutral target, an aggressive hostile that shoots, a proximity mine, etc...)

Weapon: In-Game Object

Weapon defines the attack of the player unit. Each weapon has its own stats and shoot-object component. Each weapon can also be assigned a unique alt-attack mode, in the form of an ability.

ShootObject: In-Game Object

ShootObject refers to any form of 'bullet' object that is fired by a player's weapon/ability or AI unit. There are different kinds of shoot-objects, namely simple, homing, beam and point. Please refer to the later section for how each shoot-object works.

Collectible: In-Game Object

Collectibles are bonus items that can be collected by the player unit in the game for various effects.

Effect

Effects are buffs/debuffs that can be applied to units. These effects can be applied through a normal weapon attack, an ability, or collectible.

Ability

Abilities are special actions available for the player. They can be used for various effects like a direct attack, a game-wide debuff, a self-buff and so on.

UnitSpawner

Spawner that procedurally spawns AI units during runtime.

CollectibleSpawner

Spawner that procedurally spawns collectible items during runtime.

DamageTable

DamageTable is a multiplier table used to create a rock-paper-scissors dynamics between attacks. You can set up various damage and armor types using DamageTableEditor (accessed from top panel). Each damage can act differently to each armor (ie. damage type1 would deal 50% damage to armor1 but 150% damage to armor2). Each attack (from weapons, abilities, etc.) can be assigned to use a specific damage type and each unit can be assigned a specific armor type.

ShootPoint (for Weapon And AI Unit)

Shoot-points are reference transforms assigned to Weapon and AI Unit to indicate the position where the shoot-object should be fired from and the direction they are facing. For a typical weapon which shoots straight, it's important that the z-axis direction of the shoot-point transform is aligned with the barrel of the weapon.

Triggers: In-Game Object

'Triggers' are objects used to trigger various events. Essentially, they are just invisible areas/switches in the scene. When triggered by either the Player or AI unit (by moving into the area), they perform a specific action, depending on the type of trigger used. Please refer to the 'Making & Using Triggers' section for more information.

Objective & ObjectiveTracker

Each level in TDSTK can be set to have multiple objectives. An objective can include destroying a certain group of units, fight until the spawner has exhausted all available spawns, get to a series of check points, timed survival and so on.

Most of the objective parameters can be set in the ObjectiveTracker component with the exception of timer (which can be set using GameControl) and win trigger (which is a trigger placed manually into the scene)

Level Up And Perks (PlayerProgression & PlayerPerk)

Players can gain experience and level up. As players level up, they gain various stat bonuses. The experience cap required to level up and the bonus stats gained can be configured in the ProgressionStatsEditor window.

Along with that, there's the perk system. Perks are unlockable and purchasable items that grant players various things like stat bonuses, new abilities /weapons or modify existing abilities/weapons. The perk system can be tied to the levelling system where players can gain a new perk whenever they level up.

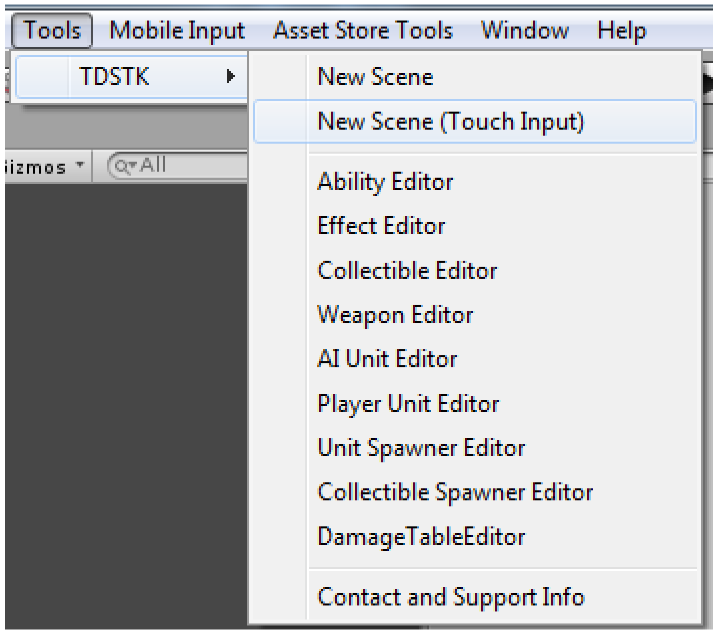
Both systems can be configured to use in conjunction with each other to create different synergies in a game. Alternatively, they can be used independently of each other. Both are optional.

Database

TDSTK uses a centralized database to store a reference to some of the prefabs (units, weapons, collectibles) and in-game items (effects and abilities). The in-game items can be created with just a simple click of a button in their associated editor. The prefabs, however, need to be created manually before they can be added to the database. Once added, they can be accessed and edited via the editor.

HOW TO:

Access The Editors



The first thing in TDSTK that you should be aware of is the drop-down menu in the top panel.

From this menu you can create a simple starter scene, access various editor menu, as well as find the contact info for further support.

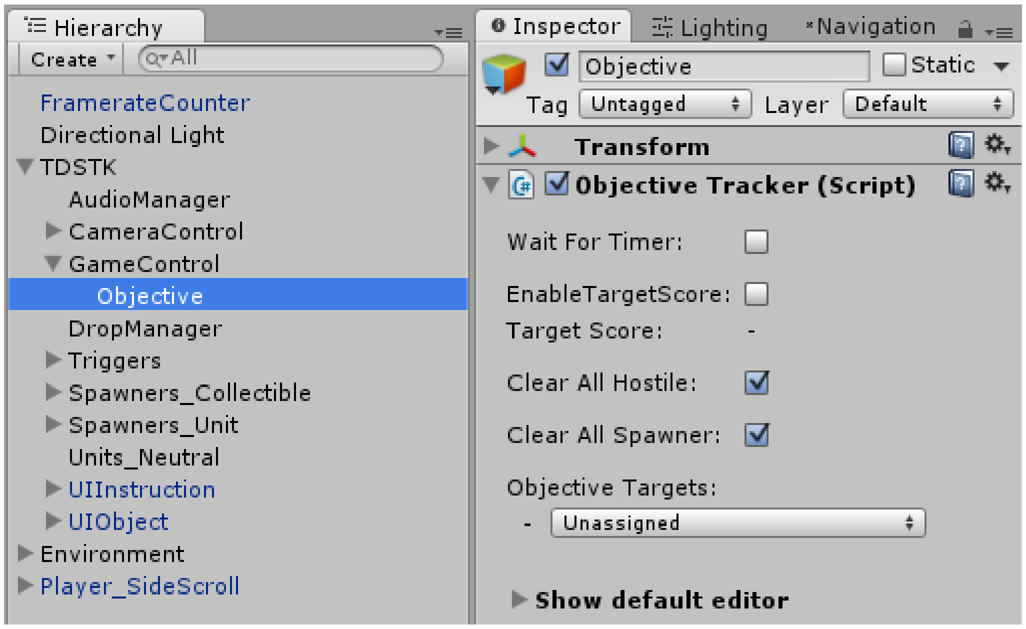
Getting Started - Create A TDSTK Scene And Modify It

To create a new TDSTK scene, simply access the top panel: '***Tools/TDSTK/New Scene***'. This should set up a game-ready scene. '***Tools/TDSTK/New Scene (Touch Input)***' would set up a game-ready scene with user interface for touch input.

The scene contains all the elements you can potentially use. It's recommended that you always start with this template and then further customize the level to your liking. Change the player prefab, modify the objectives, add triggers to spawn more units, and so on.

Setting Up Objective And Level Fail/Complete State

There are several criteria that could trigger a level complete or level fail state when using TDSTK. The first thing you should be aware of is the ObjectiveTracker component. You can assign one to the GameControl, and the setting on the component would be used as the condition for level completion. You can enable as many conditions on the objective component as you want and players would need to fulfil all of those conditions to complete the level. Each of the settings available on the component has a tooltip explaining what they are, so please mouse over the settings in the Inspector for more information.



Objective
Tag Untagged Layer Default
Objective Tracker (Script)
Wait For Timer: ☐
EnableTargetScore: ☐
Target Score: -
Clear All Hostile: ✓
Clear All Spawner: ✓
Objective Targets:
- Unassigned

In addition to the ObjectiveTracker, there's a game timer on GameControl. When enabled, this timer limits the length of the level with the time specified. When the timer runs out, the level is considered completed if all the objectives in the ObjectiveTracker have been completed. Otherwise, the level is considered failed. If the wait-for-timer box in ObjectiveTracker is disabled, the level can be completed before time runs out, as soon the objectives are completed.

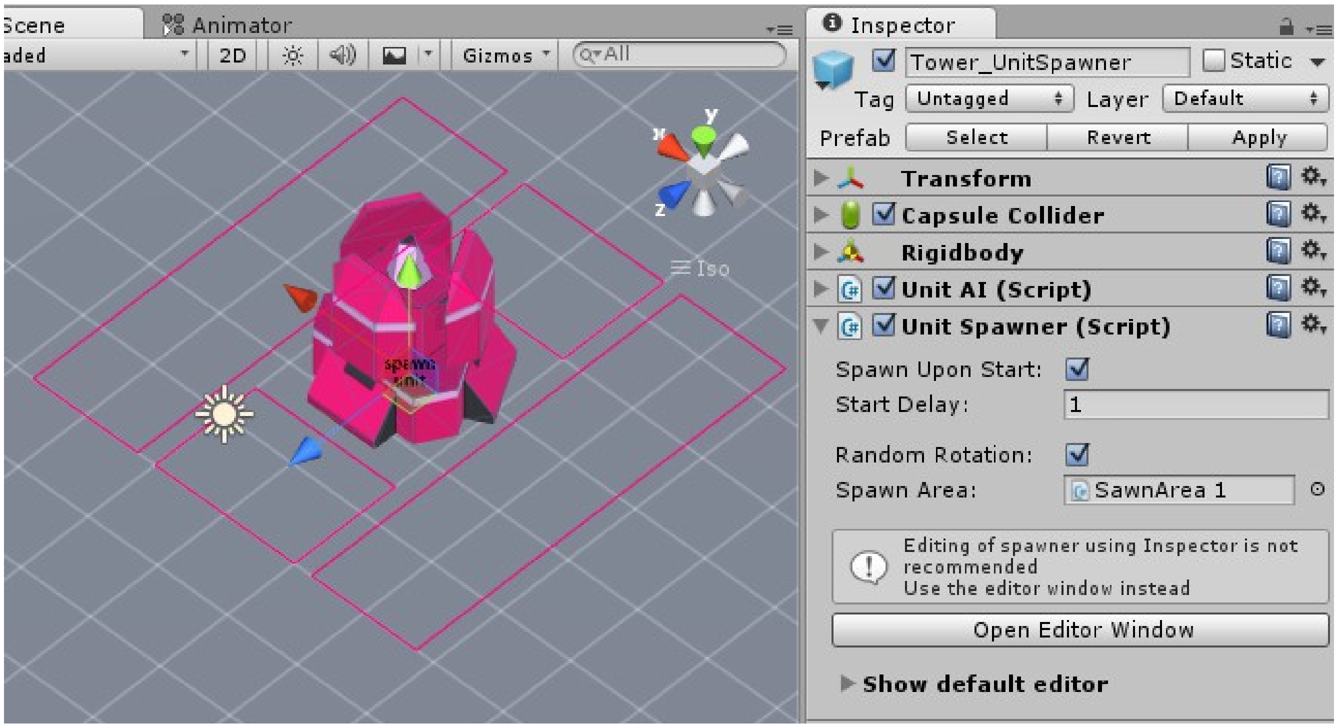
To fail a level, the player either has to be destroyed having no respawns remaining, or fail to complete all the specified objectives before the timer runs out.

Working With Unit Spawner

When designing levels, you can place specific AI units into the scene, or use UnitSpawner to spawn units procedurally during runtime. You can have multiple instances of UnitSpawner in a single scene, each with their own spawn criteria and information. For instance, spawn 10 unit-A's when player gets to room1, then spawn another 15 unit-B's when player gets to room2. The exact spawn information can be configured using SpawnEditorWindow, where the settings are pretty much self-explanatory. Furthermore, there's a tooltip for each of these settings.

It's worth mentioning you can either set the spawner to start spawning automatically (with option to time it), or use triggers. When using triggers, the spawner will remain in a dormant state until the trigger is 'triggered' by the player. For instance, place a trigger at the entrance of a room, player enters room, hits trigger and the spawner in the room starts spawning units.

You can place a UnitSpawner either as an empty game-object in the scene or as a Unit. The latter of course can be destroyed by the player. The example prefab, Tower_UnitSpawner, does just that.



Tower_UnitSpawner
Tag: Untagged
Layer: Default

Spawn Upon Start: ✓
Start Delay: 1
Random Rotation: ✓
Spawn Area: SawnArea 1

Editing of spawner using Inspector is not recommended
Use the editor window instead

An example of UnitAI being used as a UnitSpawner

Work With Collectible Spawner

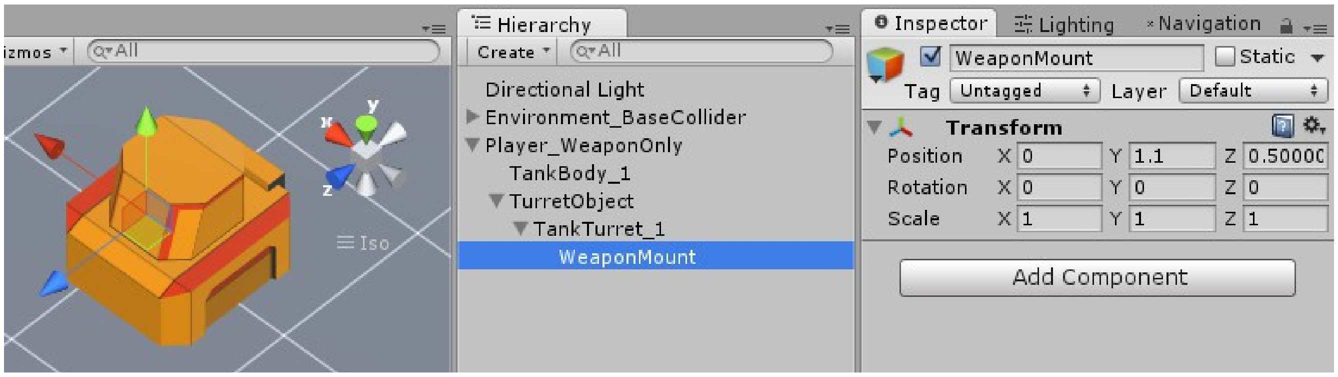
CollectibleSpawner is the collectible counterpart for UnitSpawner. It works pretty much like UnitSpawner in every way except it doesn't support a wave-based spawn, for obvious reasons.

Make A UnitPlayer Prefab

Making a player unit can be as simple as attaching a "*UnitPlayer.cs*" script to a game-object. The script will automatically add all the required components (Rigidbody and Collider) to the game-object.

Of course, to have the unit look good in-game or at least have it aim towards where it's firing, you will have to give it a turret object. TurretObject is typically a child object within the player unit transform hierarchy which always aims toward where the cursor is pointing. When setting up turret objects, it's important to make sure the transform aims toward the + z-axis when the rotation is (0, 0, 0). You can refer to the default UnitPlayer prefabs.

If you want the player to be able to switch weapons during runtime and have the switched weapon show up in the right place, you will need to assign a WeaponMount point (as a transform). This will determine where all the assigned weapon transforms will be anchored . Typically, you will want the WeaponMount point to be a child of the turret object so that it follows the aim direction of the turret. And again, make sure that transform aims toward the + z-axis when the rotation is (0, 0, 0). You can refer to the image below; note that the rotation of the WeaponMount point is (0, 0, 0) and it's pointing towards the + z-axis.



Directional Light
Environment_BaseCollider
Player_WeaponOnly
  TankBody_1
  TurretObject
    TankTurret_1
      WeaponMount

WeaponMount Static
Tag Untagged Layer Default

Transform

	X	Y	Z
Position	0	1.1	0.50000
Rotation	0	0	0
Scale	1	1	1

It's recommended you keep all child transforms of the unit free of colliders to avoid any errors in hit detection of ShootObjects or Collectibles. If you must have other colliders on the unit for whatever reason, try make them smaller than the base collider so they don't trigger inappropriately or impede the unit movement. If you want to take it a step future, you can assign it with a custom layer and have the physics to ignore all collision with that layer.

Finally, it's recommended you keep the root transform of the unit free of any mesh renderers. Instead, make any mesh a child object like the example prefab. That way you have the freedom to adjust the position of the mesh.

Make A UnitAI Prefab

Making a AI unit is very similar to making a player unit. You simply attach “*UnitAI.cs*” script to a game-object and the script will do the rest.

Of course the same rules applies for other setting such as turret and mesh. Please refer to previous section - 'Make a UnitPlayer Prefab'.

Make A ShootObject Prefab

A shoot-object is any game-object with the “*ShootObject.cs*” script attached to it. There are four distinctive types of shoot-objects, which all work in very different manners and may require different setups.

Simple: a simple projectile shoot-object which shoots forward from its shoot direction. A simple shoot-object will require a Collider and a Rigidbody attached to it. Though these components are optional, the ShootObject component will automatically add them if there aren’t any already.

Homing: a homing shoot-object which tracks a specific unit, or a point if no unit is specified. In terms of setup, a homing projectile is similar to a simple shoot-object.

Beam: a special heat-scan based shoot-object which instantly hits targets in the line of fire based on its shoot direction. In terms of setup, a beam requires a LineRenderer component to render the line of fire.

Point: a special shoot-object which uses cursor Raycasts and hits directly (and instantly) any object at the cursor’s position; meaning, it shoots passed any obstacles and hits a specific spot. This type of shoot-object has no prior requirement in order for it to work correctly. However, it's recommended you at least assign it a hit effect so the impact is visible.

Make A Collectible Item

A Collectible is any game-object with the Collectible.cs script attached on it. Simply attaching a Collectible.cs to any game-object will enable it to start working as a collectible item. To avoid any errors in hit-detection, avoid having any colliders in any of the child transforms of the collectible item, except for the collider which are required on the root object.

Making And Using Trigger

A Trigger is simply an empty game-object with the "*Trigger*" script attached on it. The trigger area of a Trigger will depend on the scale of the game-object. You can adjust them as you see fit. The gizmo should show the area being covered as you adjust the scale.

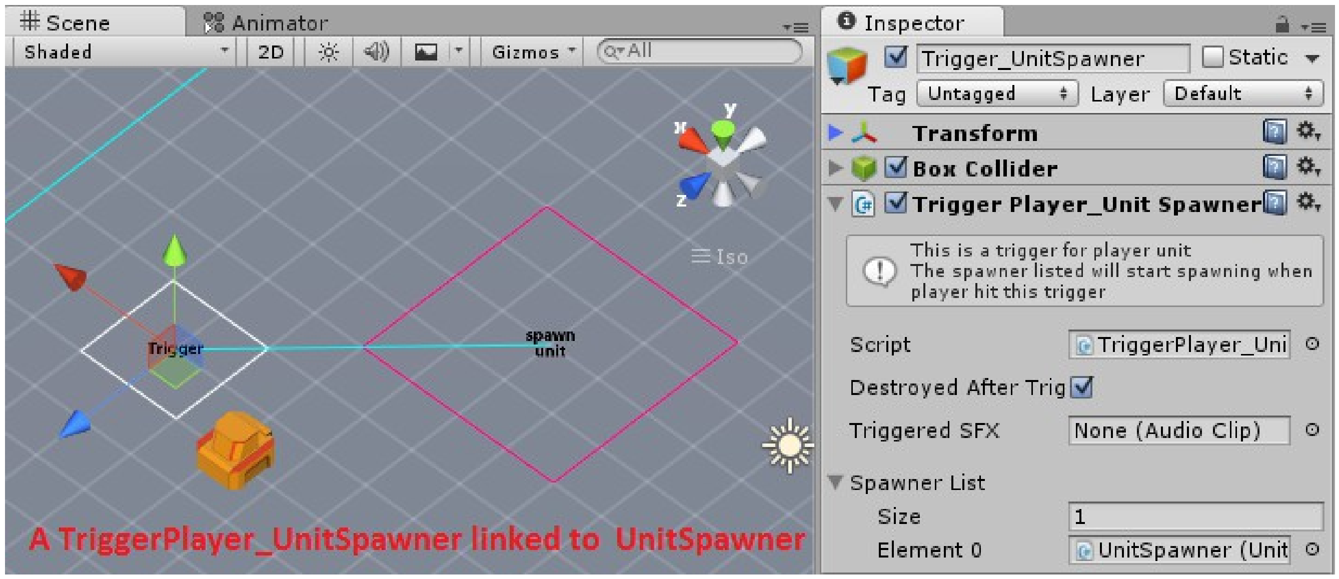
Each type of trigger serves a different purpose. The types of triggers available in the current version are:

For hostile units (to be triggered by hostile unit)

- TriggerHostile_DamagePlayer – damages player unit
- TriggerHostile_Kill - destroys the hostile unit that triggers it

For player units (to be triggered by player)

- TriggerPlayer_ActivateUnit – activates designated inactive units
- TriggerPlayer_CollectibleSpawner – prompts designated collectible spawners to start spawning
- TriggerPlayer_Damage – damages player unit
- TriggerPlayer_Objective – completes certain objective
- TriggerPlayer_PlayerSwitch – switches current active player to another player prefab
- TriggerPlayer_RepawnPoint – activates position as player's new respawn point
- TriggerPlayer_SaveProgress – save player's current progress
- TriggerPlayer_Teleport – teleports player to a designated position
- TriggerPlayer_UnitSpawner – prompts designated unit spawners to start spawning
- TriggerPlayer_Win – wins the level instantly
-



Trigger_UnitSpawner ✓ | Static ☐
Tag: Untagged | Layer: Default

Trigger Player_Unit Spawner ✓

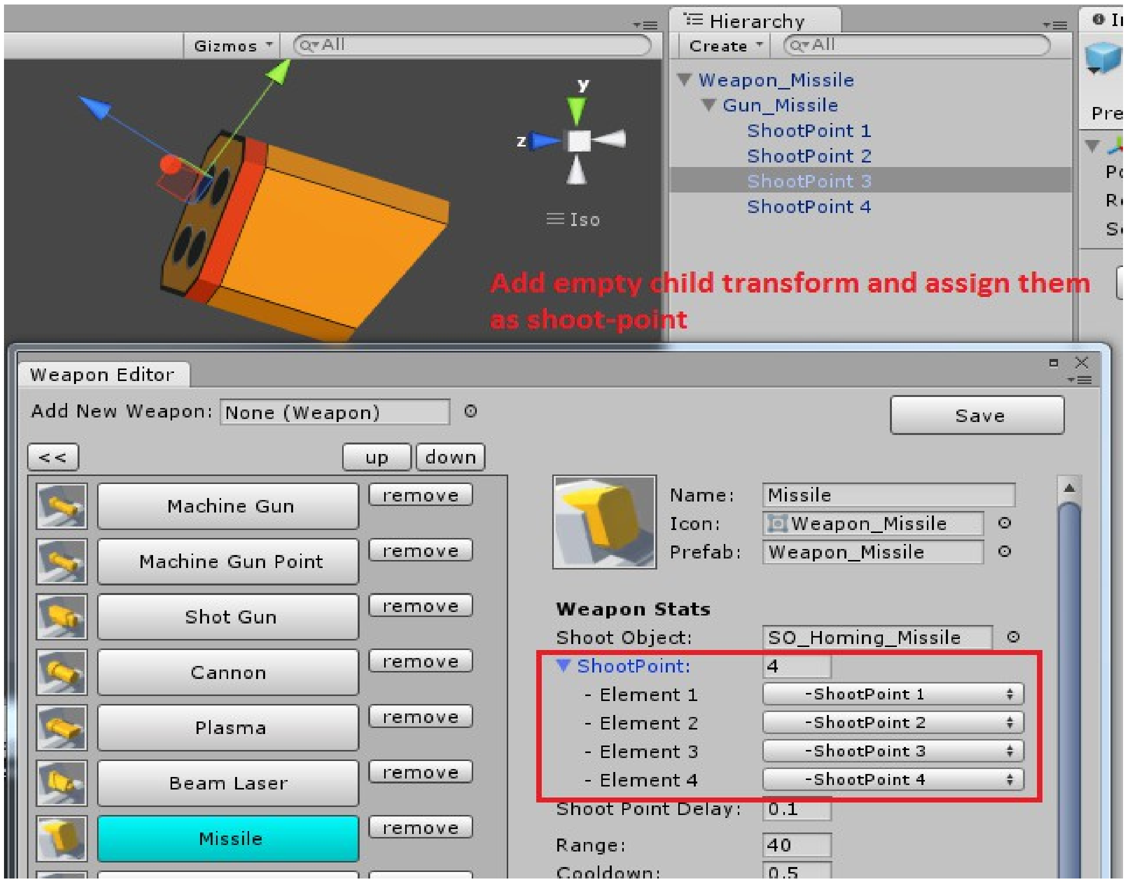
This is a trigger for player unit
The spawner listed will start spawning when player hit this trigger

Script	TriggerPlayer_Uni
Destroyed After Trig	✓
Triggered SFX	None (Audio Clip)
Spawner List	
Size	1
Element 0	UnitSpawner (Unit

A TriggerPlayer_UnitSpawner linked to UnitSpawner

Make A Weapon

A basic weapon prefab can be an empty game-oject with the “*Weapon.cs*” script attached to it. In most cases, you can give it a unique appearance by adding a custom mesh as one of the child objects. Should you do that, you may want the shoot-object (the barrel, for instance) to be fired from a specific position toward a specific direction. To do that you will want to assign a specific shoot-point (or multiple shoot-points, depending on how many objects are being fired) to the weapon in the form of an empty child transform. This empty transform will simply be used as a dummy so the weapon script knows where to spawn the shoot object and in which direction. You will need to assign this shoot-point to the weapon component. You can refer to the example weapon prefab for how the shoot-point is arranged. In most case you will need to position the shoot-point at the tip of the weapon's barrel.



Weapon_Missile
Gun_Missile
ShootPoint 1
ShootPoint 2
ShootPoint 3
ShootPoint 4

Add empty child transform and assign them as shoot-point

Add New Weapon: None (Weapon)

Name: Missile
Icon: Weapon_Missile
Prefab: Weapon_Missile

Weapon Stats
Shoot Object: SO_Homing_Missile
ShootPoint: 4
- Element 1: -ShootPoint 1
- Element 2: -ShootPoint 2
- Element 3: -ShootPoint 3
- Element 4: -ShootPoint 4
Shoot Point Delay: 0.1
Range: 40
Cooldown: 0.5

Firing multiple shoot object per shot
There are two ways to do this. First, you can increase the spread value on the weapon setting. However, with spread you are restricted in the sense that all the shoot-objects are going to be fired from the same shoot-point and subject to a pre-defined, uniformly scattered angle. Alternatively, you can assign multiple shoot-points to the weapon. A shoot-object will be fired from each shoot-point, in accordance with their position and direction. To get a clear example of these differences, please refer to the default shotgun and missile weapon prefabs. Please note that assigning multiple shoot-points can work in conjunction with spread value; so if you have 3 shoot-points assigned and have a spread value of 3. A single shot from the weapon will fire 9 shoot-objects (3 from each shoot-point).

Since a weapon prefab is eventually going to be attached to a player unit as a child object, it’s vital that it doesn't have any collider component on it to avoid hit-detection error.

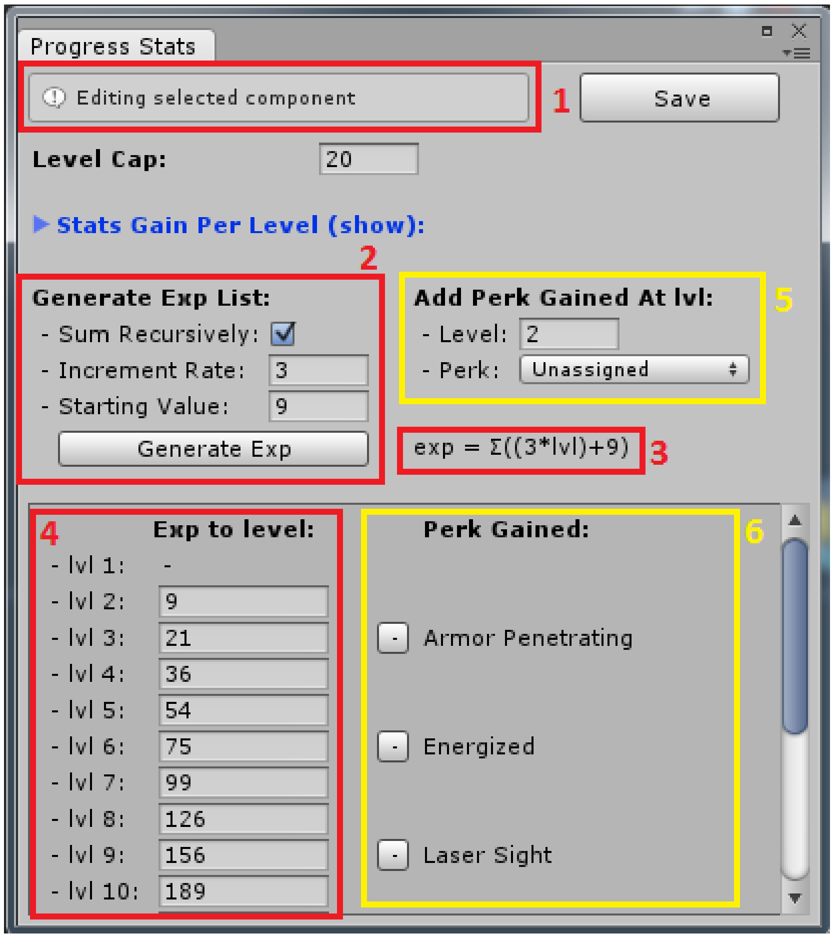
Leveling And Perk System

LEVELLING SYETEM are optional extras where player can gain experience and level up. Levelling-up grants players various stat bonuses. To enable player-levelling, you will need to attach the script "*PlayerProgression.cs*" to the player unit in the game.

There are a set of global progression stats which can be accessed and shared by all PlayerProgression components in the game. However, each PlayerProgression component also carries their own set of value. You can determine if a PlayerProgression component should use the global value or its local value by turning the '*UseGlobalStats*' option in the component on/off.

This is to facilitate the option of using a common game wide progression in typical rpg fashion, or level-oriented gameplay where you have different progression parameter in every level. For instance, if your player needs to level up through out the game, you only need to configure the global value once, then can leave '*UseGlobalStats*' on for every PlayerProgression component. But if you building a game where there are multiple levels, each with their own setup and thus require different progression/perk, you can use the local value on each PlayerProgression component

To configure various stats in regard to levelling, you can use the ProgressStatsEditor window from the drop down menu on the top panel. The ProgressStatsEditor edits the global value by default. To edit the local stats on the PlayerProgression instead of the global stats, just select the game-object with the target PlayerProgression. This is similar to the idea of editing a unit prefab or it's instance in a scene.



Level Cap: 20

Generate Exp List:
- Sum Recursively: ✓
- Increment Rate: 3
- Starting Value: 9

Add Perk Gained At lvl:
- Level: 2
- Perk: Unassigned

exp = Σ((3*lvl)+9)

Exp to level:	
- lvl 1:	-
- lvl 2:	9
- lvl 3:	21
- lvl 4:	36
- lvl 5:	54
- lvl 6:	75
- lvl 7:	99
- lvl 8:	126
- lvl 9:	156
- lvl 10:	189

Perk Gained:
- Armor Penetrating
- Energized
- Laser Sight

1. Indicates if the editor is editing the global or local stats.
2. Generates the experience list procedurally using a linear equation.
3. The equation used to generate currently selected experience list.
4. The experience list indicates the experience cap required for each level. You can edit each field directly, whether or not you first generated a formulaic list.
5. Add 1 or more perks to be gained at specific levels. The added item(s) will show up in (6).
6. Indicates any perks the player gains at each level. Each item can be removed by clicking on the '-' button.

PERK SYSTEM is an extra optional component that can be added to the framework. Perks are upgrade items that can be purchased during runtime to give players a boost in various means. This includes modifying existing stats, adding new weapons or abilities, modifying stats of an ability, etc. It's also possible to add your own custom effects.

To use the Perk System, you will need to attach the script PlayerPerk to the player unit in the game. You can then create a perk much the same way you would create an ability, via PerkEditor; which again, can be accessed from the top panel. Once you've created a perk, it should show up in the PlayerPerk component, allowing you to set its status as either enabled/disabled/pre-purchased in the game. There are several ways to use the Perk System.

These are a few examples supported by the default UI:

- **Stand Alone** – Player gains perk currency through various in-game events like defeating a boss or reaching a checkpoint. The perk currency is then used to purchase various perks.
- **Passive** – Ties to levelling system: Player gains pre-determined perk automatically by reaching a certain level. These depend on the setting in ProgressStatsEditor (refer to last section: 'Levelling System').
- **Skill/Tech-Tree** - Ties to levelling system: Player gains perk currency through levelling, which can then be spent on any perks they choose. An example would be the talent tree system in MMOS's like *World of Warcraft.*
- **Attribute** – Ties to levelling system: Perks are treated like attributes and can be repeatably purchased. Each level grants perk point(s) where a player can choose which attributes to upgrade. A similar example would be the attribute system in *Diablo*.

Obviously, some of the examples mentioned are not mutually exclusive. You can in fact use all three 'passive', 'attribute' and 'skill-tree' setups together. However, to reduce the complexity, the default UI only supports one of them at any given time. Due to the open nature and the synergy possibilities between the levelling and perk system, the potential of the levelling and perk system can only be fully utilised if you customise your UI.

For more information about how to set up the UI for the levelling and perk system, please refer to the UI section: 'Level and Perk Menu'.

Saving and Loading Perk Progress

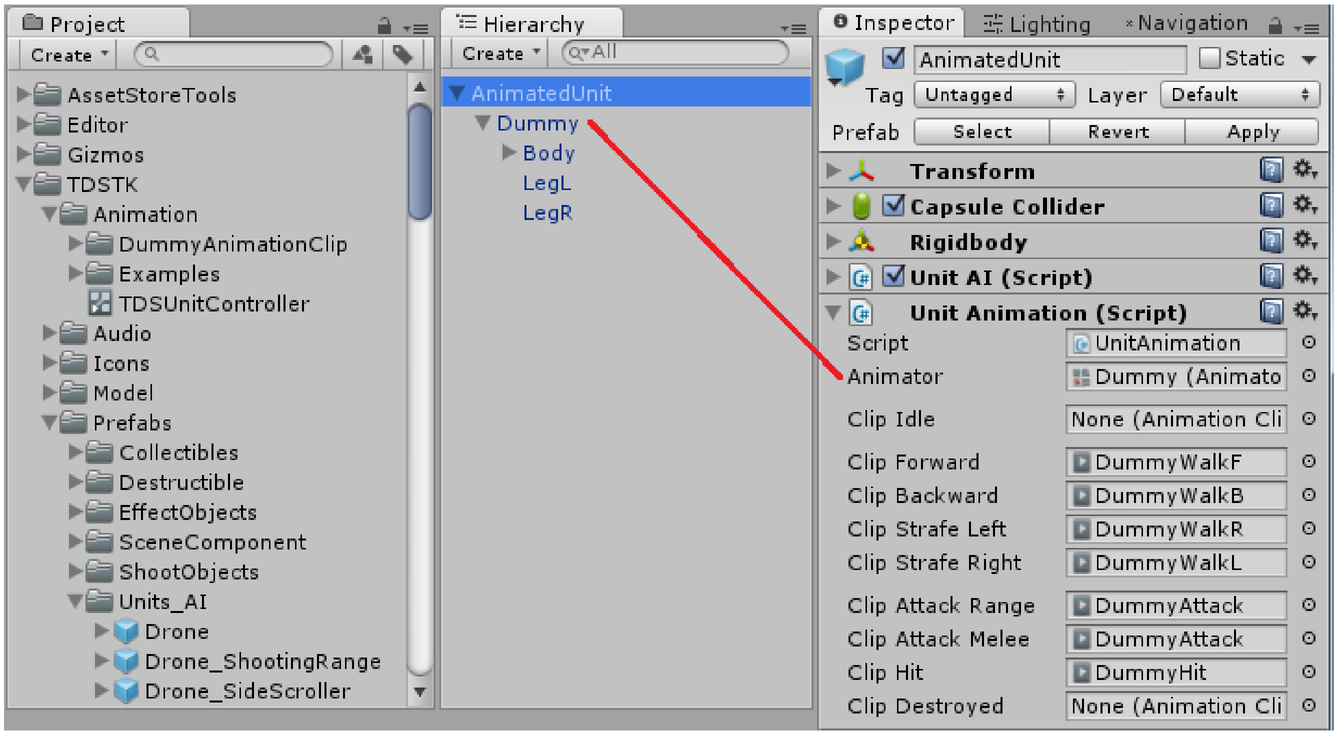
It's possible to save the progress of player levels and perks, and then load them in a future game session. The progress can be loaded when starting a new level or when player starts another game session. To do that, you simply have to check the '*LoadProgress*' and '*SaveProgress*' options on the UnitPlayer.

When save is enabled, there are a few ways which the progress can be saved. You can either check the '*SaveUponChanged*' option on UnitPlayer, which would prompt a save whenever something changes in PlayerPerk or PlayerProgression. Alternatively, you can use 'Trigger' so progress is only saved when the player reaches a certain checkpoint in game.

Animation

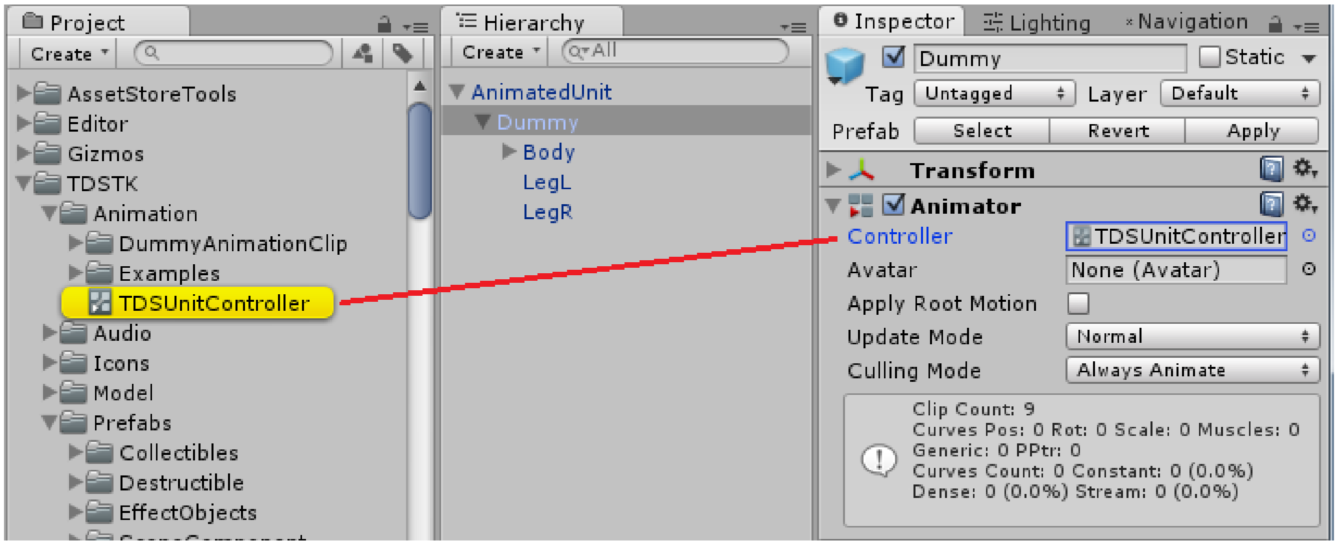
TDSTK has a system that supports animation on both player and AI units. To Add animation to a unit, you will need to:

- add UnitAnimation.cs to the root unit prefab, alongside either “*UnitAI.cs*” or “*UnitPlayer.cs*”.
- assign the animator component from the unit model to the Animator Slot of the root (Player or AI) UnitComponent
- assign '*TDSTK/Animation/TDSUnitController*' as the controller of the animator component.



Script	UnitAnimation
Animator	Dummy (Animato
Clip Idle	None (Animation Cli
Clip Forward	DummyWalkF
Clip Backward	DummyWalkB
Clip Strafe Left	DummyWalkR
Clip Strafe Right	DummyWalkL
Clip Attack Range	DummyAttack
Clip Attack Melee	DummyAttack
Clip Hit	DummyHit
Clip Destroyed	None (Animation Cli

Unit Animation component is added to the unit prefab. Animator component on the '*Dummy*' game-object is being assigned to the '*Animator*' slot of UnitAnimation.



Dummy
Tag: Untagged
Layer: Default
Controller: TDSUnitController
Avatar: None (Avatar)
Apply Root Motion: ☐
Update Mode: Normal
Culling Mode: Always Animate

```
Clip Count: 9
Curves Pos: 0 Rot: 0 Scale: 0 Muscles: 0
Generic: 0 PPtr: 0
Curves Count: 0 Constant: 0 (0.0%)
Dense: 0 (0.0%) Stream: 0 (0.0%)
```

TDSUnitController has been assign as the controller of the Animator

Once that is done, you can assign the animation clip wanted to each corresponding animation slot in the UnitAnimation component. They are all optional so you can leave any slots empty where there's no animation clip to play.

You can refer to the two example prefabs about how this is done. There's one for player unit and one for AI unit. They are both located in '*TDSTK/Animation/Examples*'.

PLAYER CONTROL & USER INTERFACE:

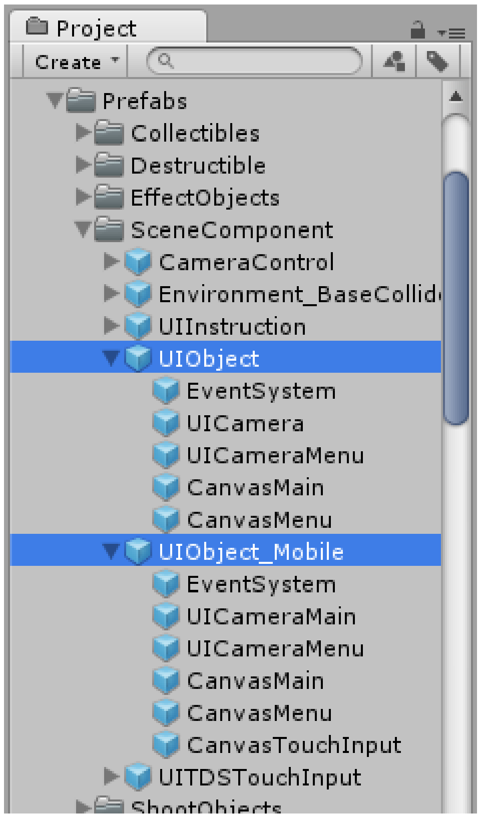
Player Control

There are several control options that might be enabled/disabled based on the settings you use.

First, movement: There's two types of movement, rigid and freeform. Rigid movement is very much a binary movement where player either stays stationary or moves at max-speed. Freeform is a more natural movement mode with acceleration and deceleration. With freeform movement mode, the player unit will have movement momentum and may carry on moving even when there's no move input. In this case, 'Space' key can be used to apply braking to the player unit. In either mode, 'Left-Shift' can be used to boost player speed at the expense of energy.

Abilities and weapon alt-fire mode both shares the same input: right-mouse-button. However, that is only true <u>when only one of them is enabled </u>in the game. When both ‘Ability’ and weapon ‘Alt-Fire’ mode is enabled, the activation of Abilities is switched to the middle-mouse button. You can enable/disable ‘Ability’ and/or ‘Alt-Fire’ mode in GameControl.

User Interface



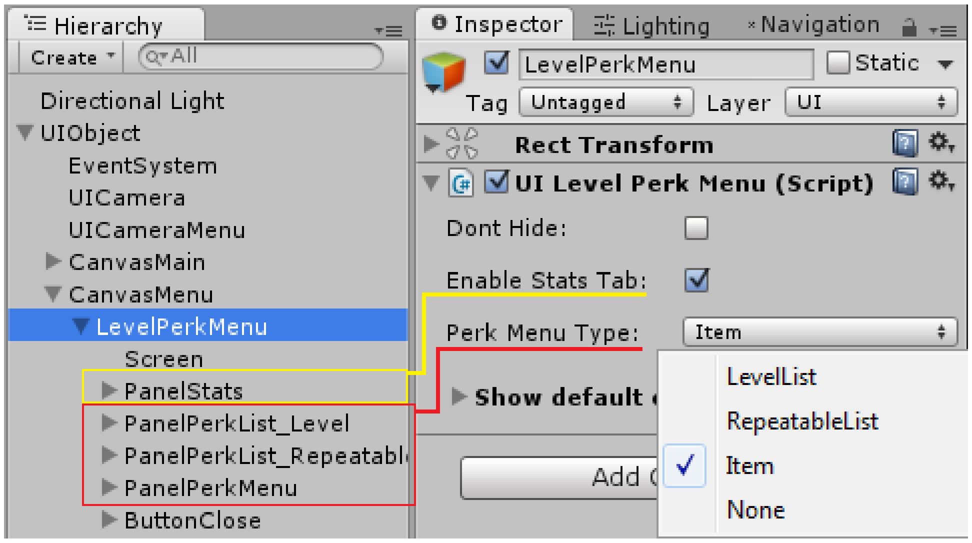
User Interface is very much an integral component of the package although it is not a core component. It's built to be modular and thus can be replaced with any custom solution.

There are two pre-built prefabs that act as a UI template which you can drag into any custom scene; one for mouse and keyboard input and one for touch input (mobile platform). They are both located in '*TDSTK/Prefabs/SceneComponent/*', named respectively as *UIObject* and *UIObject_Mobile*. If you create a new scene using the top panel. The new scene will take consideration of the type of scene you want to create and will use the appropriate UI prefab automatically.

Certain parameters in the default UI prefabs that are made to be configurable for ease of use (ie. Enable-overlay, list all abilities, etc) but for the most part, you will need to thinker with it to get the setting/appearance you want. You can also adjust the various elements within it to change its appearance. You can also deactivate certain UI elements if they are not needed in the scene (like how it's done in the demo scene). However, it's recommended that you get yourself familiar with Unity GUI system; understand how it works before you start tinkering with it.

Level And Perk Menu

LevelPerkMenu is the special UI object that supports customization, mainly to provide a means to build custom tech-trees in conjunction with the perk system.



LevelPerkMenu
Static
Tag Untagged
Layer UI
UI Level Perk Menu (Script)
Dont Hide: ☐
Enable Stats Tab: ✓
Perk Menu Type: Item

In a nutshell, LevelPerkMenu is the main controller for all the UI regarding level stats and perks. As shown in the image, the option '*EnableStatsTab*' is related to PanelStats (which shows player level information and its non-interactable) while '*PerkMenuType*' is related to PerkMenu. You can only choose either to show or not show the level stats, but you can choose from which type of perk menu to use.

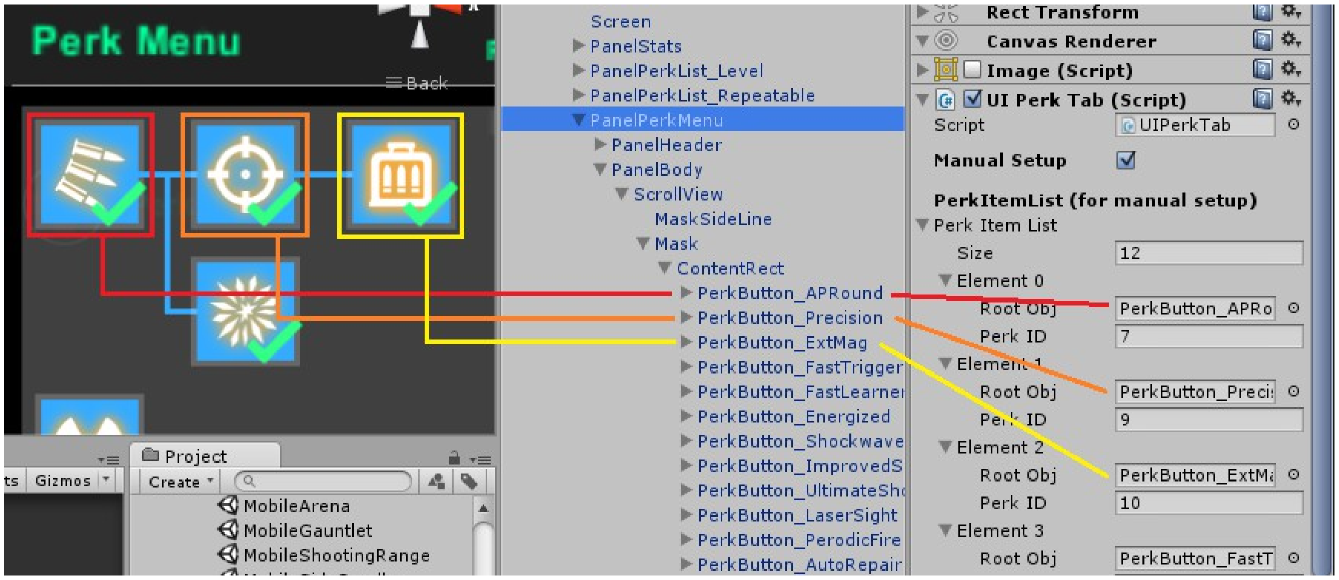
The options of perk menu type are as follow:

- **LevelList** – have perk UI show a list of perks to be unlocked upon reaching certain levels, as seen in demo scene 'DemoShootingRange' (corresponds to child object: PanelPerkList_Level).
- **RepeatableList** – have UI show a list of perks that would be presented as attributes and can be repeatably purchased, as shown in demo scene 'DemoGaunlet' (corresponds to child object: PanelPerkList_Repeatable).
- **Item** – have UI show a customizable skill-tree (or just a grid of items), as shown in demo scene 'DemoArena' (corresponds to child object: PanelPerkMenu).
- **None** – don't have any perk menu.

***PanelPerkList_Repeatable -** you can choose which perks will show up in this menu by selecting it just like you would with the available perk selection in PlayerPerk.

***PanelPerkmenu**

When the box '*ManualSetup*' is checked, you can arrange the perk items any way you want. You'll need to assign them as items to the UIPerkTab component (as shown in the image below), as well as specify the ID of the perk the button is associated to. When the flag '*ManualSetup*' is off, all the available perks enabled in PlayerPerk will be assigned a button automatically; you can specify in the future if the perk should show up in the UI.



```
Screen
▶ PanelStats
▶ PanelPerkList_Level
▶ PanelPerkList_Repeatable
▼ PanelPerkMenu
  ▶ PanelHeader
  ▼ PanelBody
    ▼ ScrollView
        MaskSideLine
      ▼ Mask
        ▼ ContentRect
          ▶ PerkButton_APRound
          ▶ PerkButton_Precision
          ▶ PerkButton_ExtMag
          ▶ PerkButton_FastTrigger
          ▶ PerkButton_FastLearner
          ▶ PerkButton_Energized
          ▶ PerkButton_Shockwave
          ▶ PerkButton_ImprovedS
          ▶ PerkButton_UltimateSh
          ▶ PerkButton_LaserSight
          ▶ PerkButton_PerodicFire
          ▶ PerkButton_AutoRepair
```

```
▶ Rect Transform
▼ Canvas Renderer
▶ ☐ Image (Script)
▼ ☑ UI Perk Tab (Script)
  Script              UIPerkTab
  Manual Setup        ☑

  PerkItemList (for manual setup)
▼ Perk Item List
    Size              12
  ▼ Element 0
      Root Obj        PerkButton_APRo
      Perk ID         7
  ▼ Element 1
      Root Obj        PerkButton_Preci
      Perk ID         9
  ▼ Element 2
      Root Obj        PerkButton_ExtMa
      Perk ID         10
  ▼ Element 3
      Root Obj        PerkButton_FastT
```

You can find an example of this UI (as shown in the image above) in the example scene Level&PerkManu. Note that under each 'PerkButton' game object, there are 'Connector' and 'ConnectorBG' objects. These are simply stencil line Images indicating the upgrade path from one perk to another (if there is one). The Connector objects must be child objects under the hierarchy of the perk item and be named 'Connector' and 'ConnectorBG'. The 'Connector' game object will be set to inactive if the perk has not yet been purchased and active if the perk has been purchased.

You can refer to the perk menu in the demo scene to find out how all this setup comes together. In the demo, the ConnectorBG objects are the (grey) lines indicating the upgrade perk at the end of the path is unavailable, while the Connector objects are the (blue) lines indicating the next connected upgrade(s) is available. When an item is purchased, the Connector object will be enabled, blocking the ConnectorBG object behind it. For the player, it will seem like the line has turned from grey to blue, indicating that the link between two items has been activated and is thus available for purchase.

Touch Input And Mobile Support

The framework is mobile-ready and comes with a pre-set touch input for that purpose. Just like the default UI and input for mouse and keyboard, the touch input is independent from the core framework. You can always implement your own touch input should you wish.

By default, the touch input is integrated with the UI prefab for mobile platforms. For all intents and purpose, it’s the same as its mouse and keyboard counterpart, apart from the additional UI elements for touch input (on-screen buttons and joystick).

Please note that the default visual effects used for the demo are not exactly optimized for mobile; same goes for the level design of the demo scene (some spawn too many units). Therefore, you might experience some frame-drop when using some of the demo scene as is.

THANK-YOU NOTE & CONTACT INFO

Thanks for purchasing and using TDSTK. I hope you enjoy your purchase. If you have any feedback or questions, please don't hesitate to contact me. You will find all the contact and support information you need via the top panel "***Tools/TDSTK/Contact&SupportInfo***". Just in case, you can reach me at k.songtan@gmail.com or TDSTK support thread at Unity forum.

Finally, I would appreciate if you take time to leave a review at AssetStore page. Once again, thank you!