TurnBased-ToolKit

Documentation for Unity3D

Version: 3.1.4
Author: K.Song Tan
LastUpdated: 4th August 2021

Forum: http://goo.gl/3hntLv
WebSite: http://songgamedev.com/tbtk
AssetStore: http://u3d.as/5ed

Thanks for using TBTK. This toolkit is a collection of coding framework in C# for Unity3D. This toolkit is designed to cover most of the common turn-based tactics game mechanics. Bear in mind TBTK is not a framework to create a complete game by itself. It does not cover elements such as menu scenes, options, 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 you try to familiarise yourself with the basics of Unity3D.

You can find all the support/contact info you ever needed to reach me under '**Support And Contact Info**' via the top panel. Please leave a review on the AssetStore if possible; your feedback and time are much appreciated.

Important Note:

TBTKv3.x is a new version redesigned and written from scratch therefore it's not backward compatible with earlier versions of TBTK.

If you are updating an existing TBTKv3.x and don't want the new version to overwrite your current data setting (towers, creep, damage-table, etc), Just uncheck '*TBTK/Resources/DB_TDTK'* folder in the import window.

OVERVIEW

How Things Work

A working TBTK scene has several key components to run the basic game logic and user interaction. These are pre-placed in the scene. They have various settings that can be configured, and these settings dictate how the scene will play. For instance, what type of grid will be used, how the unit move order is decided, can unit-A attack unit-B, and so on.

Then there are prefab like units and collectibles, and abstract item abilities and perks. These are the objects needed to form a valid game scene. Each of these have their corresponding control components on them. Their stats and behaviours are configured via those components. These prefabs and items are assigned to a series of databases. Their key components will access their respective databases to allow access to these prefabs. This is so the game knows which units to spawn, which units have access to which abilities, and so on.

Basic Components And Class

These are the bare minimum key components to have in a basic functioning TDTK scene. You will need at least one of these in a working scene. There can be only one instance of these components in the scene.

GameControl and **TurnControl:** Controls the game state and major game logic.

AI: Runs the AI algorithm.

GridManager: Stores the grid information and runs all the logic on the grid.

GridGenerator: Generates the grid.

GridIndicator: Controls all the visual indicators of the game state. It's not technically mandatory but without it the game would certainly be unplayable.

UnitManager: Stores information of all units.

AbilityManager: Runs the logic for all abilities.

CollectibleManager: Runs the logic for all collectibles.

Attack: An abstract class that runs all the logic during an attack (or when an ability is used).

AStar: An abstract class that runs all the logic for pathfinding.

Prefab Components And Class

These are the components that interact with each other in runtime to form the gameplay.

Unit*: The component on each unit.

Collectible*: The component on each collectible.

ShootObject*: The component on objects fired by a unit during an attack to hit their target.

Node: The individual node on the grid.

Ability: The ability item.

Effect: The effect item that can be applied to a unit during runtime as a buff or debuff, from either an attack, an ability or collectible. Can also act as 'Aura' for unit to be applied to friendly units in range.

Perk: The bonus item used to alter other item-based (units, abilities, effects) stats during gameplay.

* These are prefabs.

Optional/Support Components And Class

These are the optional components for a TBTK scene. There can be only one instance of these components in the scene.

PerkManager: Controls the logic for the perks system and contains information about usable perks. It's required only if the perk system will be used in that particular level.

ObjectPoolManager*: The component responsible for all logic pertaining to object-pooling.

CameraControl: The control component for the camera.

* These will be created in runtime automatically if they are required for the scene but are not already present.

Layers

TBTK 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 25: 'Terrain'
- layer 27: 'ObstacleHalf'
- layer 28: 'ObstacleFull'
- layer 29: 'Node'
- layer 30: 'Invisible'
- layer 31: 'Unit'

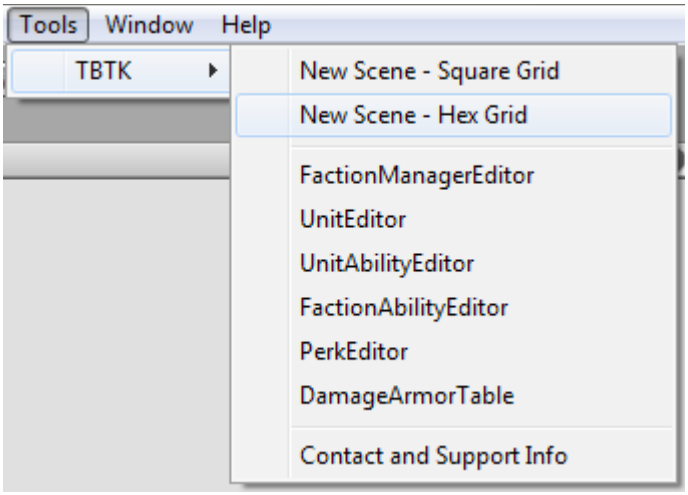
You can find the definition of these layers in *TBTK.cs*.

UI

Although the UI is required for a level to be playable, it's a modular extension of the framework. You can have the base logic running without it. That means you can delete all UI components and replace them with your own custom UI.

HOW TO:

Create A TBTK Scene

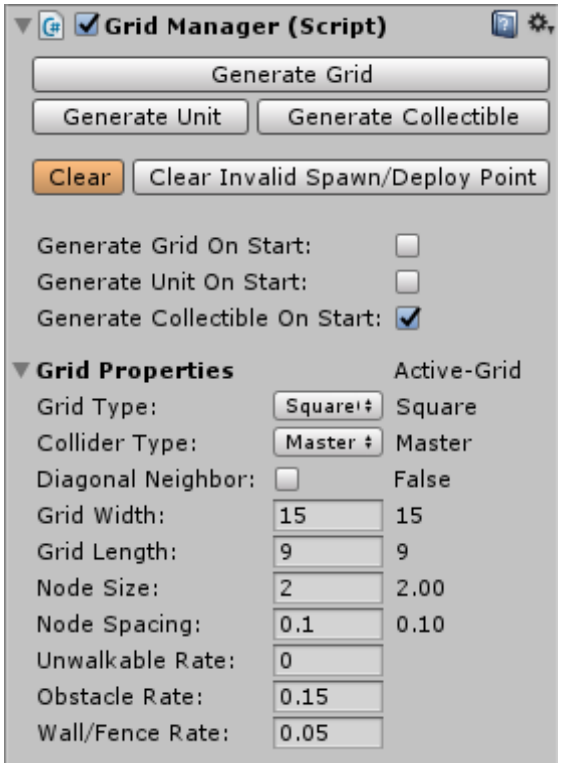


Assuming you have some basic knowledge about Unity, it's very easy to create a functioning game using TBTK. To create a new TBTK scene, simply access the top panel: '***Tools/TBTK/New Scene - Square Grid***' or '***Tools/TBTK/New Scene - Hex Grid***'… depending on whether you want to set up a square-based or hex-based grid.

The scene is set up with 2 factions (1 player and 1 AI) and is ready to be played from the get-go.

It's recommended you start with a default scene and configure it to your liking instead of starting from a blank scene. This way you won't need to worry about getting all the settings correct before the scene can run.

Generating Grid

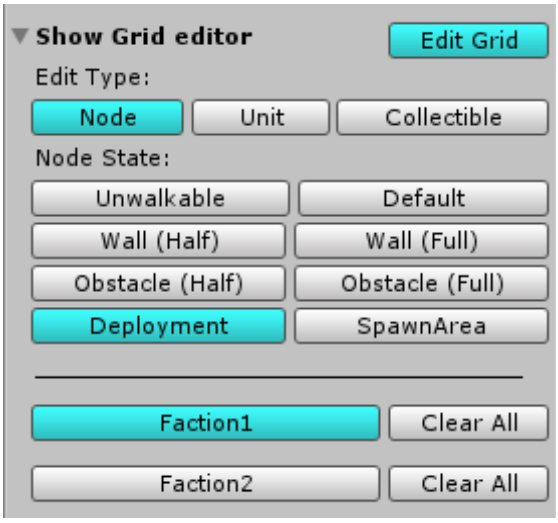


Generating a grid is pretty straight-forward (assuming you are starting with the default scene). Just select the game-object 'GRID_MANAGER_GENERATOR' in the hierarchy.

In the inspector, you can specify the type of grid you want to generate, the dimensions, node size and so on. When you are happy, just click the '**GenerateGrid**' button. Please note that generating a large grid may take some time.

Also note that when generating a grid, the new grid overwrites any existing grid in the scene. Any items on the grid will be removed as well.

Edit The Grid



You can edit the grid directly via the SceneView if you have the GridEditor script component active (specifically, when you have the '*GRID_MANAGER_GENERATOR*' object selected in the Hierarchy).

You will need to make sure you have the button 'Edit Grid' enabled (highlighted blue). From there, simply select what you want to be changed/added to the grid and then click on the grid in SceneView (also ensure *Gizmos* is enabled in Scene View).

As seen in the image, there are 3 main categories which you can edit (Node, Unit and Collectible). Unit and Collectible are self-explanatory; they're for adding/removing units and collectibles. Node is for editing the grid itself. Here's a brief explanation of what each option does.

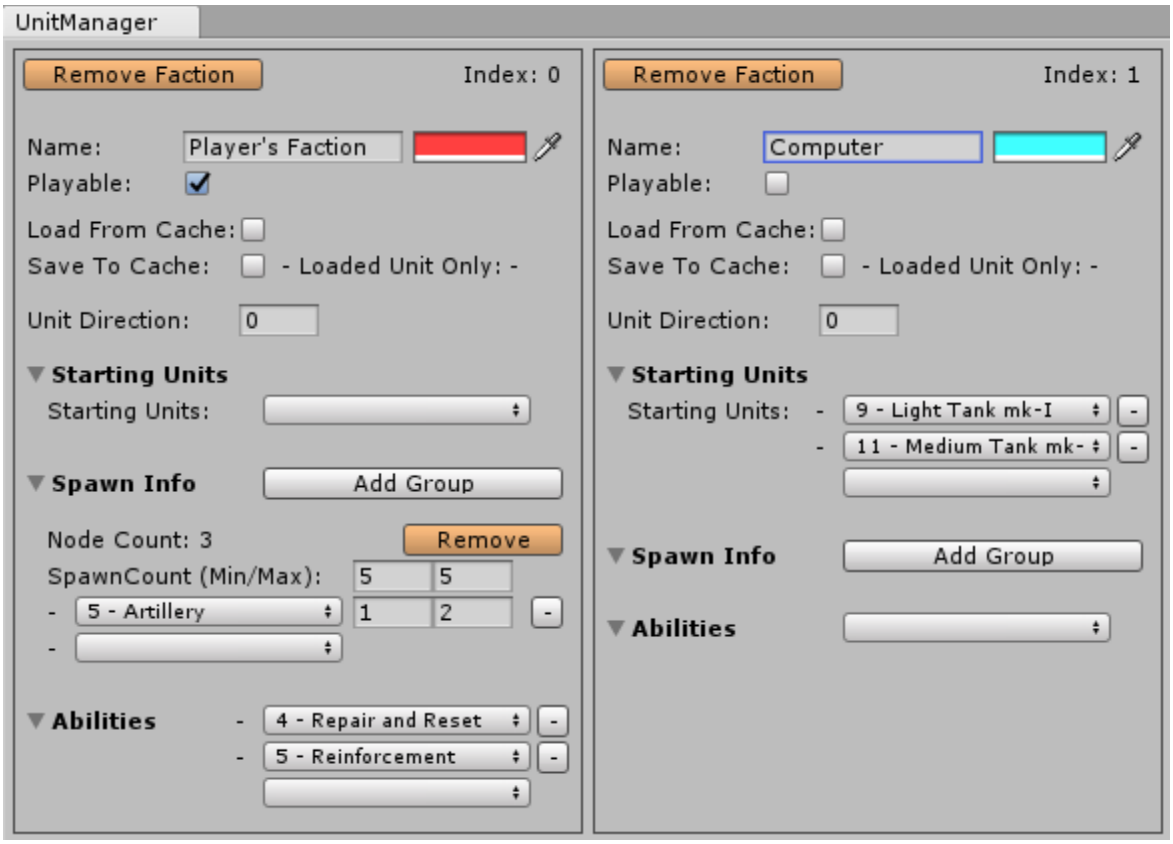
- **Unwalkable**: the node cannot be occupied by a unit and will be set to invisible (by default).
- **Default**: just a normal node
- **Wall (**half/full**)**: an obstacle separating 2 nodes which prevents a unit from moving between those nodes. A full wall will block line-of-sight.
- **Obstacle (**half/full**)**: an obstacle that occupies the entire node, preventing movement from all directions. A full obstacle will block line-of-sight.
- **Deployment**: node which indicates where a faction's unit can be deployed.
- **SpawnArea**: node which indicates where a faction's unit can be deployed when procedurally generated.

While having '*GRID_MANAGER_GENERATOR*' object selected in the Hierarchy with 'Edit Grid' enabled, you can right-click on the grid in SceneView to remove the items (Unit, Collecitble, Obstacle, etc.) or clear a state (deployment and spawn area) corresponding to the edit type tab selected in editor on the node. For instance, right-click on a node when the edit unit tab is selected to remove the unit on the node. Right-click on a node when 'Deployment' tab is selected will clear the node as a deployment node.

***Note**:
- Full obstacles and walls are used for LOS calculations when Fog-of-war is enabled.
- Both half and full obstacles and walls are used to determined cover type when the cover system is enabled. Half Obstacles/walls provide half the cover bonus will full obstacles/walls provide the full cover bonus.
- SpawnArea and Deployment nodes are tied to specific factions specified in UnitManager

Setting And Edit Factions

A functional scene will need at least 2 factions. You can set up factions using UnitManagerEditor, which can be opened via the dropdown menu or using the 'Open Window-Editor' button on UnitManager in Hierarchy. There's no limit to how many factions you can have in a scene. You can also set multiple factions to be player factions, if setting up for a hotseat game. For instance. Any non-player faction will be controlled by AI during runtime.



Each faction can have its own set of starting units, deployed units, spawn/deployment areas on the grid as well as (faction) Abilities.

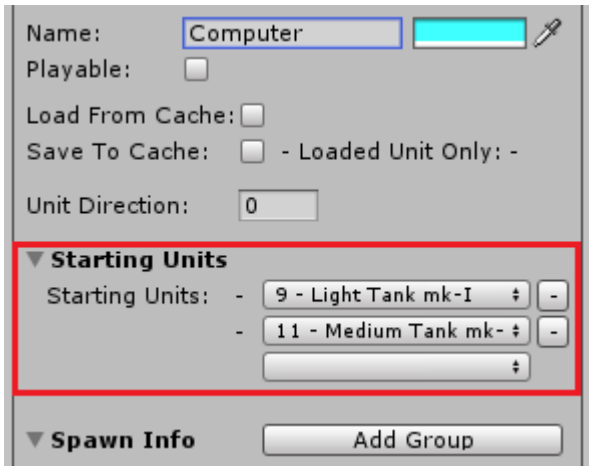
Starting Units determines the fixed set of units the faction possesses at the start of the game. These units will be deployed within their faction's deployment area.

Spawn Info is used for procedural unit generation. These units will be spawned as soon as the grid is generated. You can set the placement of these units by specifying the spawn area on the grid using 'Edit Grid' tool on GridManager . Note that you can create multiple spawn groups to have groups of units that scatter across the grid for each faction in the game.

Adding Units To The Grid

There's a few ways a unit can be added to the grid. In all cases, the unit will need to be associated with a specific faction set in UnitManagerEditor

As 'Starting Unit' under UnitManager



SartingUnit is the unit to be spawned at the start of the game. These unit will subsequently be deployed on faction's deployment area.

Each item in the list will be spawn as a unit instance so you can have more than one instance of the same unit prefab on list.

As you set unit to the StartingUnit list, you will need to setup sufficient deployment area on the grid so that these unit can be placed on grid on runtime.

To setup deployment area for a faction, you will need to use the GridEditor. Make sure you select the right faction in the GridEditor (by matching faction's name). When you click on the node in the SceneView, a cross with the faction's color will show up on the node to mark the node as a deployment node for the faction

As a Deployed unit on the grid via GridEditor under the 'GridManager' component

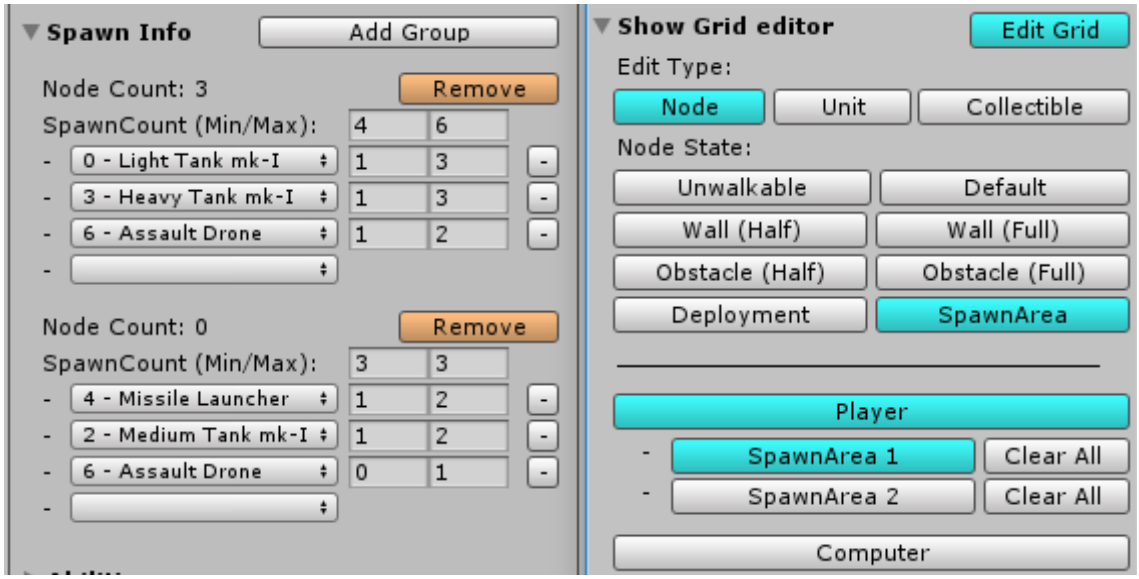
You can directly place a unit onto the grid via the GridEditor. Just set the '*EditType*' on the GridEditor to Unit and you will see all unit prefabs show up in the Editor. You can select any of them and simply click on any node on the grid and then an instance of the selected unit prefab will be spawned on the chosen node. Right-click on a node with a unit on it to remove that unit.

Please note that the faction of the unit being deployed is based on the faction selected in the Unit's Faction Tab. The selection options in there is based on the faction set in UnitManagerEditor.

Tip: when placing units on the grid, the unit will face away from the centre of the node following the mouse position, snapping to face the nearest perpendicular border.

As a Deployed unit on the grid via Procedural generation

The procedural unit generation algorithm spawns units on the grid based on spawn information specified for each faction. So to enable procedural generation, you need to first set up that information for those factions in UnitManager.



SpawnInfo allows you to add spawn groups, where you can assign different sets of units that can be spawned in different locations. Each group has its own spawn limit, where you can assign the min/max available units to be spawned. Each group corresponds to an independent SpawnArea (a group of nodes on the grid where the unit can be spawned) which can be set via 'Grid Manager' by selecting Edit Type: *Node* > *SpawnArea*. Once the information is set up properly, simply use the "*Generate Unit*" button either at the top of GridManager.

Load From Cache…

Finally, the unit can be loaded from a previous scene. To do that, you will need to check the '*Load From Cache*' option for the faction in UnitManager. When checked, instead of loading units listed under 'Starting Units', that faction will instead load any units in the runtime cache as their starting units. You can either assign units to be loaded using code or by checking the '*Save To Cache*' option.

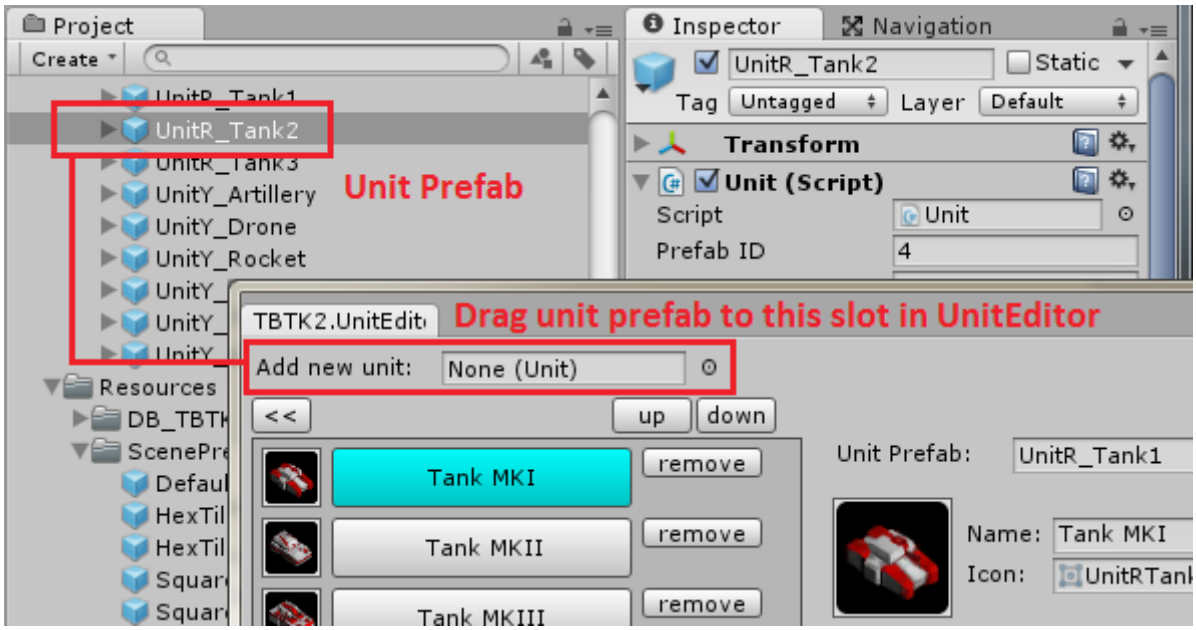
Enabling the '*Save To Cache*' option will save any of that faction's remaining units in the current scene to the cache. So if the next scene has '*Load From Cache*' enabled, those remaining units from the prior scene will be used as the starting units for this new scene.

Refer to the section 'Integrating TBTK To Your Game' for more information.

Adding New (Unit And Collectible) Prefabs To The Game

To add a new unit prefab to the game, first you will need to create the unit prefab (refer to the 'Unit Prefab' section below for how to create a prefab) and then add it to UnitEditor. You can either manually drag the prefab to the UnitEditor window (as shown below) or use the '*Add Prefab to Database*' button on the 'Unit' component of your tower's prefab. Once a unit prefab is in the UnitEditor, the unit should appear in any other editors where that unit prefab applies (ie, Unit Manager)

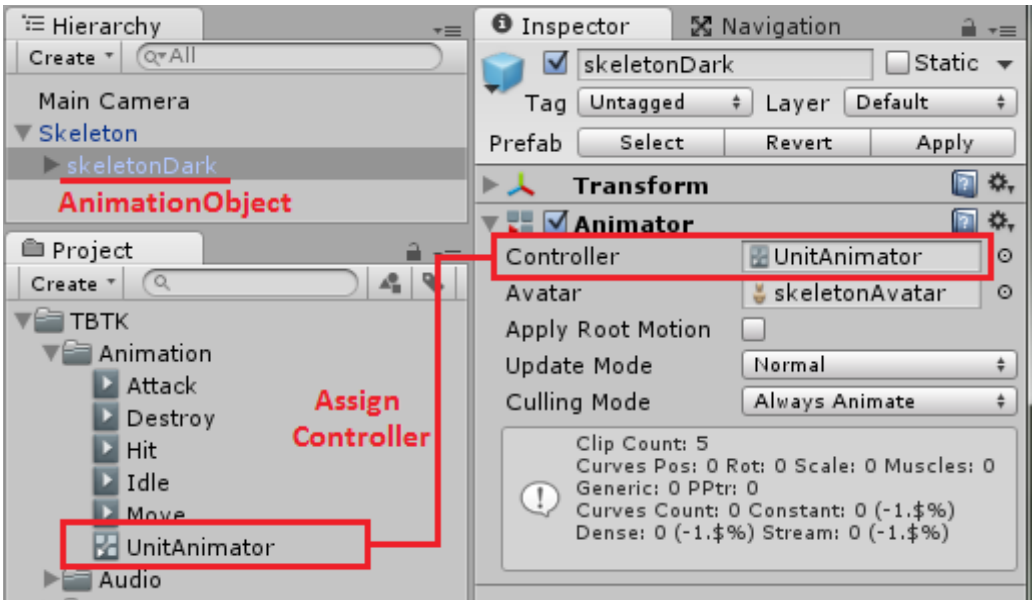
Collectible prefabs can be created and added to CollectibleEditor in the same way.



Adding New Animation To The Unit Prefab

TBTK uses the Mecanim animation system. The various animations to be used for a unit can be assigned in UnitEditor. However, before you can do that you will need to set up the prefab properly. First you will need to make sure there's an Animator component on the prefab; this usually comes with the animated model. You will need to assign the *UnitAnimator* as the animator controller for the *Animator* component as shown in the image below. Once done, you should be able to assign the animation clip you want to use in the UnitEditor.

*The system also works with legacy animation clips.
*Make sure 'Apply Root Motion' is turned off in the animator or the animation may drive the model position, offsetting the model position as it plays.



Working With Collectibles

The workflow for creating collectible prefabs and adding them to the game is very similar to the workflow for unit prefabs. The only difference is that the settings for collectible generation on the grid is done through CollectibleManager.

To create a new collectible item, simply create a new object and attach the *Collectible.cs* script to it. Like units, any visible mesh/effect on the prefab is optional. Once it's made into a prefab, it's ready to be added to the grid.

There's two ways to add a collectible to the grid. Manually, you can use GridManager to plop individual items on the grid. Alternatively, you can let procedural generation do the work. The parameters for procedural generation can be found on CollectibleManager.

HOW THINGS WORK:

Item DB (DataBase)

TBTK uses a centralized database to store all the information for prefabs (units and collectibles) and in-game items (abilities, perks, damage and armor types, etc.). 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 (see 'Add New Prefabs to The Editor' for more information). Once added, they can be accessed and edited via their editor. Prefabs that aren't added to the database will not be appear in the game.

Abstract items like abilities and effects can be created in their corresponding editor. Once created, they can be accessed in all other relevant editors. Unit abilities will show up in UnitEditor and so on.

Perks can be individually enabled/disabled in PerkManager via the Inspector. Disabled perks won't be available in the scene.

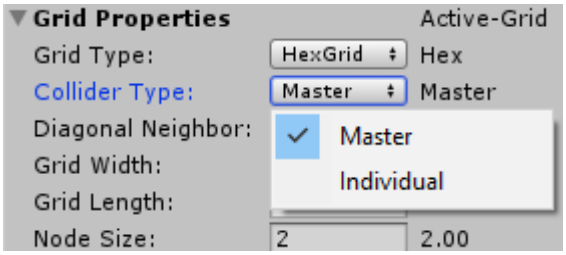
Global Setting & GameControl

You may want to have all the scenes in your project using a similar ruleset. In that case, check the '*UseGlobalSetting*' box on GameControl. When checked, instead of configuring the local variables on GameControl, you will be configuring the global setting, which is applicable to any scene that also has its '*UseGlobalSetting*' box checked.

Grid

The grid is made out of a series of game-objects. Each node and item on the grid is a prefab. These prefabs are assigned in GridGenerator. You can change the appearance of the node by switching the default prefab to a custom prefab. Note that you can assign multiple prefabs for the obstacles and walls.

Collider on Grid

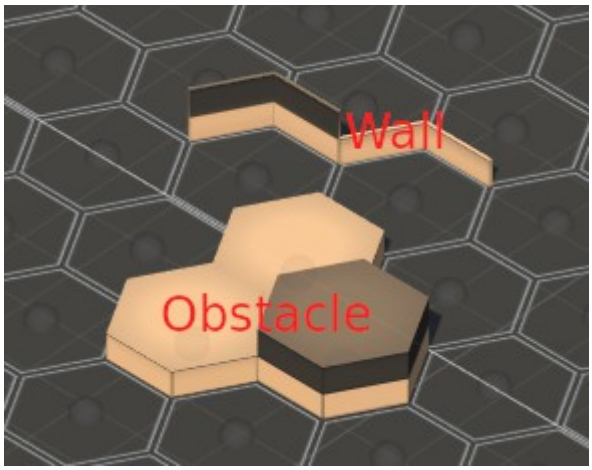


To enable detection of the cursor on a node in runtime, the grid will need to have collider(s) on it. There are two modes which can be set on GridManager – *ColliderType*. The first one is for the whole grid to use a single master collider (thus using the node prefab with *no* collider). The second one requires each individual node to have a collider (thus using the node prefab *with* collider).

Using master collider would allow a significant performance boost and a much larger grid. However, the node position cannot be adjusted in any way. On the other hand, while individual colliders have a much higher performance cost, the position of individual nodes can be adjusted manually. In short, individual collider mode is used when the position of the individual node needs to be customized (ie. height adjust, position shift, etc.).

Obstacle And Wall Prefabs

Obstacles and walls are objects placed on the grid to block unit movement, block line-of-sight (if fog-of-war is enabled) and provide cover bonus (if cover system is enabled). There are two types of obstacles: half cover and full cover. Only a full cover obstacle would block line-of-sight.



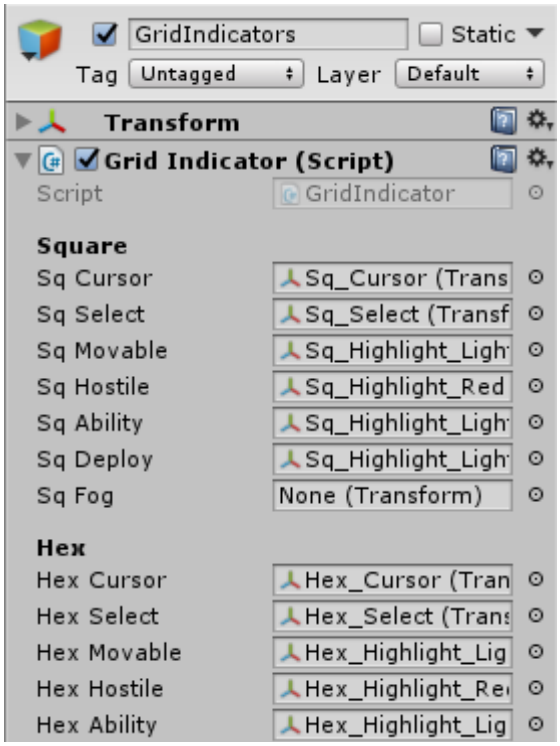
Obstacles are placed on top of nodes to stop that node from being occupied by any unit. Walls are placed between two nodes to block access between them.

Both wall and obstacle prefabs can be placed on the grid via GridEditor. The prefab requires:

- a collider
- layer assigned to CoverHalf/CoverFull (27/28 by default)

The type of cover will depend on the layer assigned to the prefab.

Visualization & Indicators

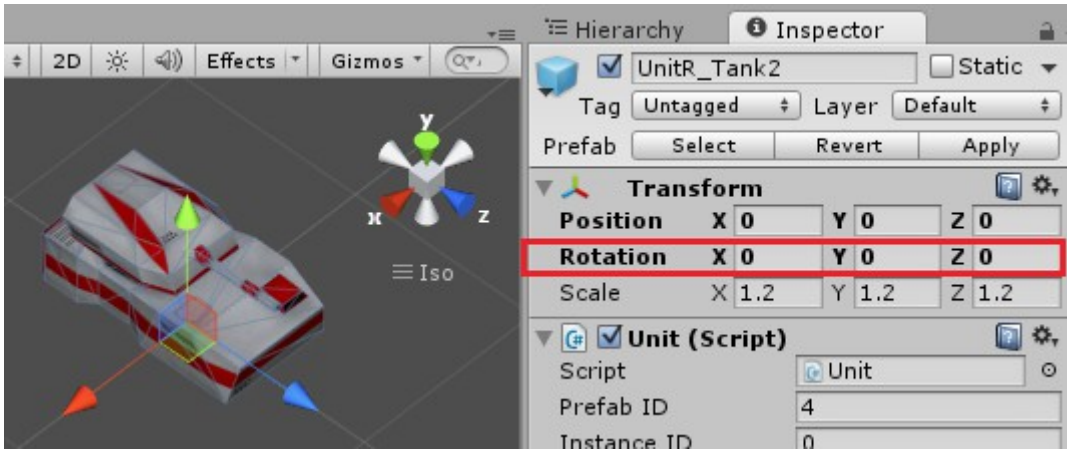


The appearance of the cursor/target indicators, as well as the appearance of nodes indicating various statuses, can all be customized. These indicators are basically gameObject prefabs of various appearances. They are assigned to GridIndicators (object in Hierarchy). You can change the appearance of a certain state of the node or an indicator by changing the prefabs.

Unit Prefabs

A primitive unit prefab can be an empty game-object with just the script *Unit.cs* attached on it. Any mesh/modal is entirely optional (an invisible unit can function just fine).

It's recommended that any mesh is place as the child transform of the unit prefab. When placing the mesh, make sure it's facing the +ve z-axis when the unit rotation is at (0, 0, 0) otherwise the unit may not be facing the correct way in runtime.



To configure a unit, it's recommended that you do so via the UnitEditor. Which can be access via the top panel. Note that to assign abilities to unit, you must first create the abilities in UnitAbilityEditor. Once an ability is created, it will show up in the Ability tab in UnitEdtitor.

A unit prefab has to be add to the UnitEditor before it can be accessed in editors.

Hybrid Units (Range + Melee Attack)

It's possible to have units that switch between range and melee attacks, depending on their target's distance. To do that, simply check the "Use Melee Attack" option in UnitEditor. This will open up another set of parameters attributed to the unit's melee attack. You can then set the melee range for the unit. In runtime the unit will switch between melee and range attack depending on target range. For this reason it's also recommended that you make sure the melee-attack range is less than the unit range-attack range.

A unit that use both range and melee will have 2 sets of stats (damage, hit chance, etc.) for normal attack; one for range and one for melee. Each attack type will have their respective shootObject, audio and animation.

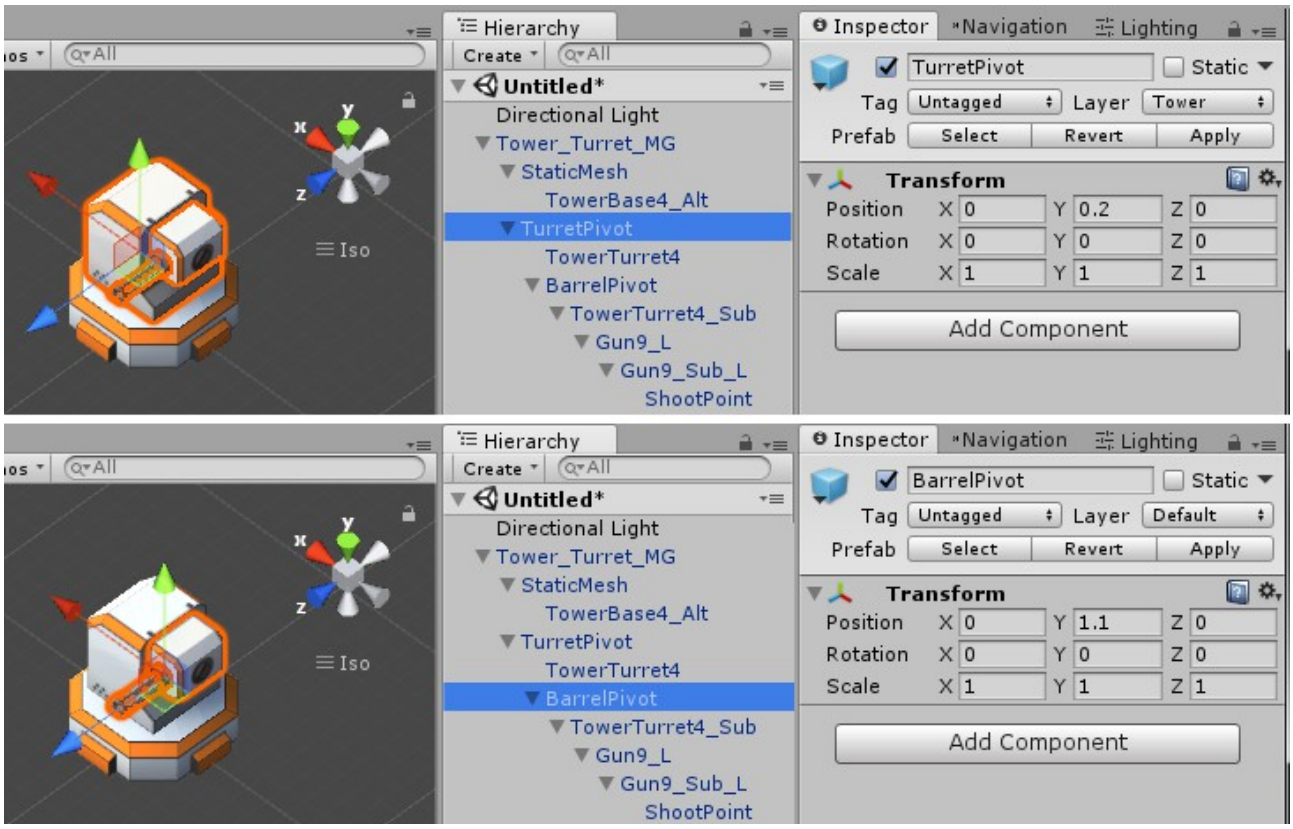
If you need a unit that use melee attack exclusively, you don't need this option. Using an empty 'Effect' type shoot-object and an attack animation would suffice. Stats calculation wise, both melee and range attack are identical.

Get Unit To Aim And Fires At Targets

Turret-type units attack their target directly by firing shoot-objects at them. In many cases, you might want the unit to aim towards the target and have the shoot-object fired from the correct position in relation to the model/mesh.

For aiming, the code will rotate the assigned transforms '*Turret-Pivot*' and '*Barrel Pivot*' in the unit's transform hierarchy to face the unit's current active target. The way the model works, 'Turret-Pivot' is the main pivot that can be rotated in both x-axis and y-axis. '*Barrel-Pivot*', on the other hand, is optional and expected to be anchored as a child to '*Turret-Pivot*'. '*Barrel-Pivot*' will only be rotated along the x-axis.

To make sure the turret aims in the right direction, you need to make sure the rotation of both '*Turret-Pivot*' and '*Barrel-Pivot*' is at (0, 0, 0) when aiming along the + z-axis as shown as the image below. If you are not sure, please refer to the default tower prefab. You can also follow this step by step instruction at the end of this section. To understand why this is set up this way, it's strongly recommended you go through [this tutorial video] to get the basic understanding of hierarchy and parent-child relationships.



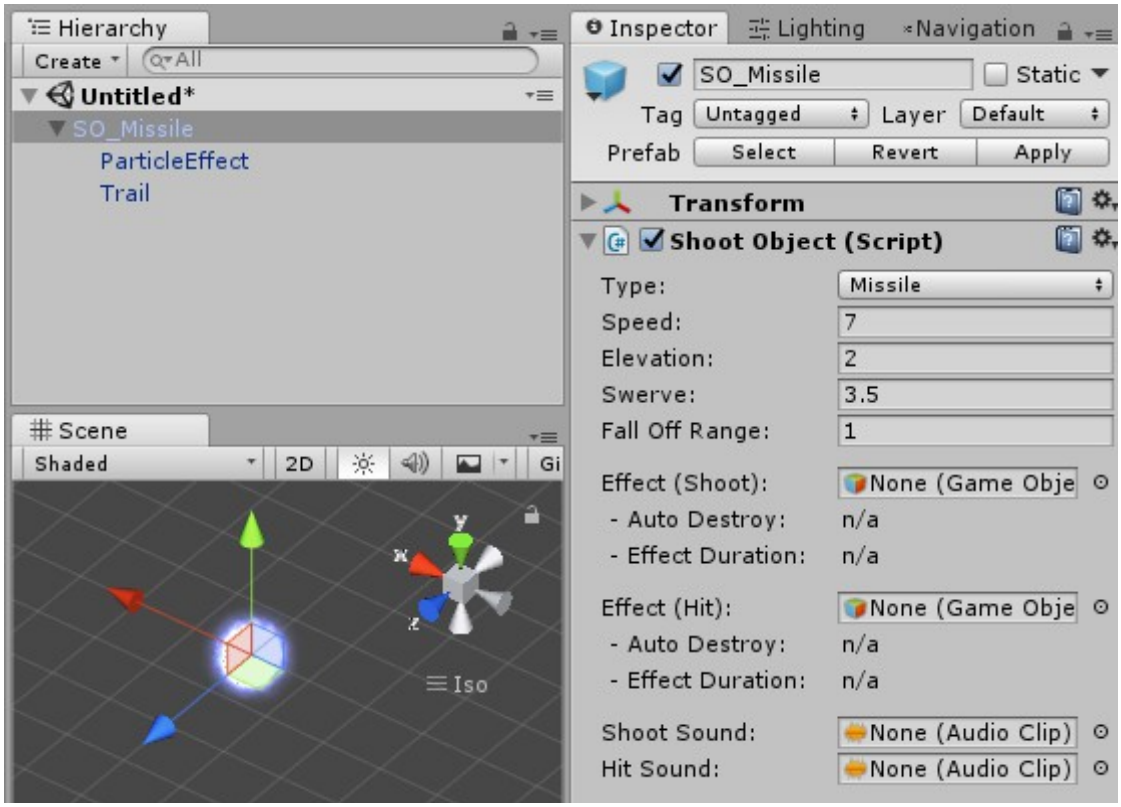
Shoot-points are the reference (empty) transforms in the hierarchy of a unit to indicate the position where shoot-objects should be fired from. To have it work with aiming, it's recommended that you anchor it as a child object of 'Turret-Pivot', or 'Barrel-Pivot' (if there is one). Again, please refer to the default tower prefab. You will find that they're all positioned at the tip of the model's barrel.

Step by step to set up basic unit with turret that aims in the right direction.

1. Create an empty game-object, add a Unit component (either '*UnitTower,cs*' or '*UnitCreep.cs*') on it. This game-object is the root object of this new prefab.
2. Drag your model into the empty game-object, set the position to (0, 0, 0). This will make sure the model is positioned at the root object center.
3. Now rotate the model's transform so that it faces the z-axis.
4. Expand the hierarchy, try to find the game-object which serves as the pivot for the turret. For reference, we'll call this *Turret*.
5. Create a new empty game-object, make sure the rotation is (0, 0, 0) and name it 'TurretPivot'
6. Drag TurretPivot into '*Turret*', set the position to (0, 0, 0). Now the TurretPivot should be in a similar position with *Turret*.
7. Reverse the child parent relationship between TurretPivot and *Turret*. Make TurretPivot the parent and *Turret* the child.
8. Now TurretPivot is ready to be set as '*Turret-Pivot*' of the unit in the editor.
9. You can repeat step-3 to step-7 to set up the barrel object, if the model has one.

ShootObjects

ShootObject refers to the 'bullet' object that is fired from a unit in an attack. In TBTK, an attack event from unit to unit will always involve a shoot-object. Even in an event of a melee attack.



A primitive shootObject can be an empty gameObject with just the *ShootObject.cs* script attached to it. This would enable the shoot-object to be assigned to unit and be fired upon attack despite it's not visible. Any mesh/model/visual-effect on the shoot-object is optional.

There are three types of shoot-objects and each of them serve a different purpose.

Projectile: A typical object that travels from the firing point toward the target before it hits. A projectile shoot-object can be configured to simulate a curved trajectory.

Missile: Similar to a 'projectile'-type shoot-object, except it also curves along the y-axis.

Beam: Used for a beam effect where LineRenderer is used to render a visible line from the shoot-point to the target. A timer is used to determine how long until the target is hit after the shoot-object is fired.

Effect: For attacks that don't require the shoot-object to travel from the shoot-point to target. A timer is used to determine how long until the target is hit after the shoot-object is fired. Also used for melee attacks.

Unit And Faction Abilities

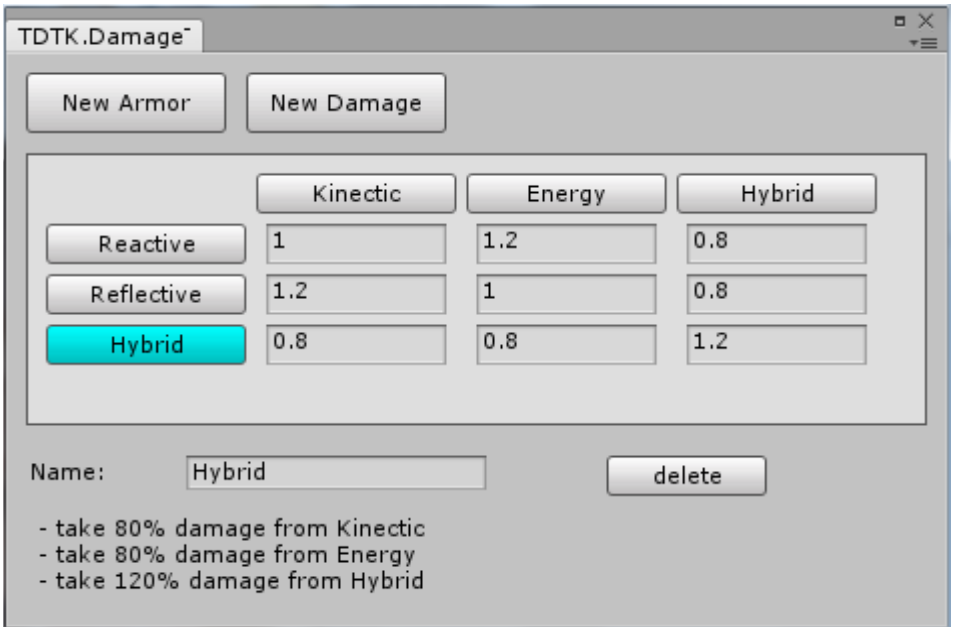
UnitAbility refers to abilities that are tied to a unit. They can be customized to do a variety of things. You can create *unit-abilities* in *AbilityEditor* and assign them to each unit in *UnitEditor* (both can be accessed via the top panel). Once created, a *UnitAbility* can be shared across multiple units.

FactionAbility is similar to *UnitAbility* except they refer to abilities that are tied to an entire faction. They can be created and edited via AbilityEditor (you will need to switch the editing target to faction). Like *UnitAbility*, once created, a *FactionAbility* can be assigned to any faction, and each faction can have their own set of *FactionAbilities*. Ability assignments to each faction can be done in UnitManagerEditor.

Damage Table

DamageTable is a multiplier table used to create a rock-paper-scissors dynamic between units. You can set up various damage and armor types using DamageTableEditor (accessed from the drop-down menu). Each damage type can act differently to each armor type. (ie. damage type1 would deal 50% damage to armor1 but 150% damage to armor2).

Each unit's attack, ability and effect that could cause damage to their target can be assigned a damage type.



Perks And PerkManager

The Perk system is an (optional) extra tool for the framework. Perks are upgrade items that can be purchased during runtime to give players a boost in various ways. This includes modifying a unit's stats, adding new *UnitAbilities* to a specific unit, modifying stats of existing abilities, etc.

You can create a perk much like how you would create an ability, via PerkEditor (which again can be accessed from the drop-down menu). Once you have created a perk, it should show up in the PerkManager, allowing you to set its status as enabled/disabled/pre-purchased in the game.

You can enable persistent progress of the perk in a game session by checking the option '*Save To Cache On End*' on PerkManager. This will cache any progress made with the perk (ie, unlocking a perk). If any subsequent level has a PerkManager with '*Load From Cache On Start*' enabled, the PerkManager will retrieve the cached progress so any perks unlocked in previous levels will be unlocked at the start of this new level. Note that this is not the same as saving; the progress only lasts through the current game session. Once the player exits the game, the progress is lost.

Please note that PerkManager only supports playable factions and the perk bonuses will be applied to all playable factions.

Fog-of-War

Fog-of-war is a built in system in TBTK to hide any hostile unit that is out of unit's sight. There are two ways in which a unit can be concealed from hostile units.

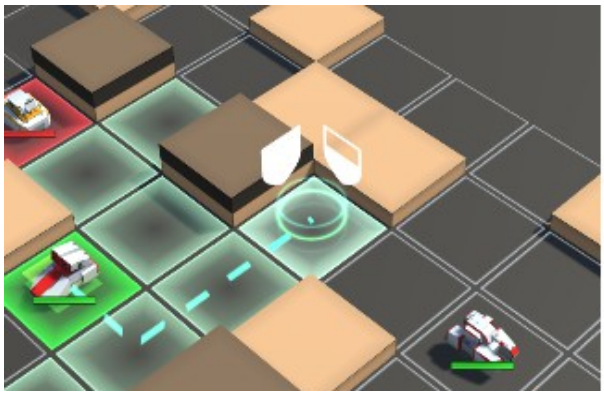
- Being out of all hostile units sight.
- Has no direct Line-of-Sight with any hostile unit (blocked by a obstacle with full cover)

A unit cannot attack a hostile unit that is concealed.

CoverSystem & Obstacle

CoverSystem is a built-in system in TBTK to emulate the cover system seen in games like XCOM. Basically, it tries to simulate a real life situation where a combatant is harder to hit when he is hiding behind cover. A unit is considered “in cover” when it is behind--and adjacent to--an obstacle with respect to its attacker. A unit in cover gains a cover bonus which gives them a bonus to dodging incoming attacks. Any unit attacking a target not in cover will gain a critical hit bonus.

There are two tiers of cover bonuses: half and full. Each tier of cover provides different bonus values which can be set in GameControl. A cover is determined by raycasting against Obstacle or Wall objects. The layer assigned to the obstacle or wall object determine if the cover bonus is half or full. A shield icon overlay will show up when a unit is about to move into a node with cover.

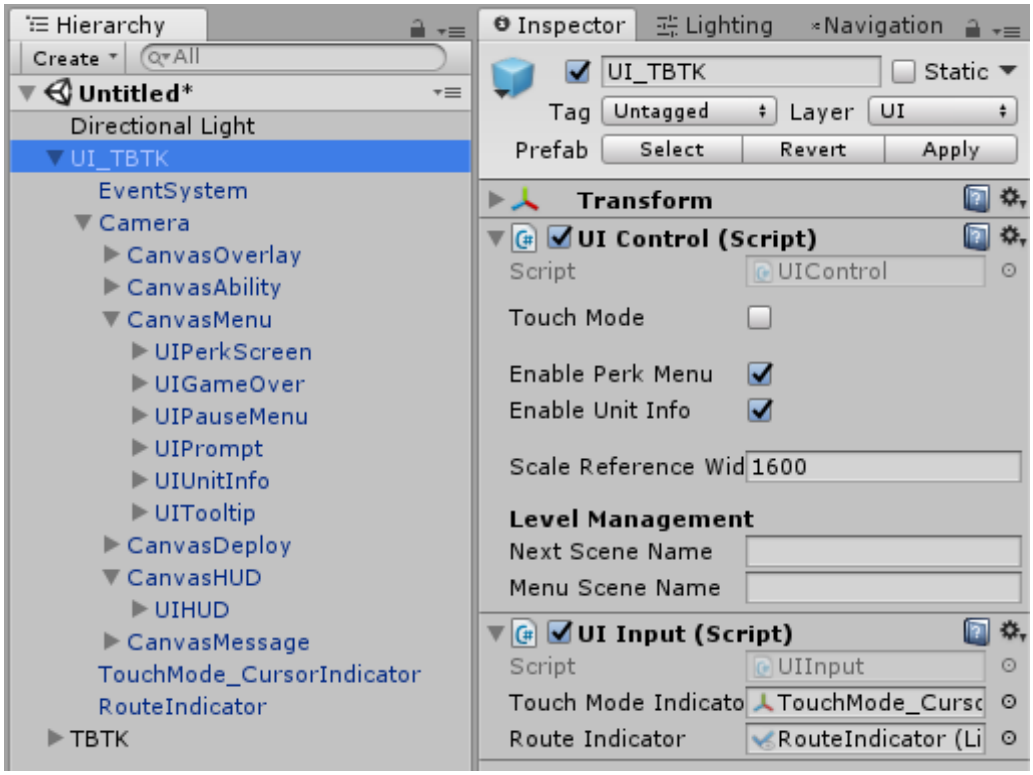


ABOUT USER INTERFACE:

Configure The Default UI

User Interface is very much an integral component of the package; although it's not a core component. It's built to be modular and thus can be replaced with any custom solution. You can simply adjust the various elements in the default UI to change its appearance. However, it's recommended you get yourself familiar with Unity GUI and understand how it works before you start tinkering with it. There's a prefab that acts as a UI template, located in '*TBTK/Resources/NewScenePrefab/UI_TBTK*'.

Certain parameters in the default UI prefab are made to be configurable. They are immediately visible on the UIControl component when you select the UI_TBTK prefab (as shown in the image below). In most cases you won't need to bother with other settings of other UI components unless you want to modify the UI.

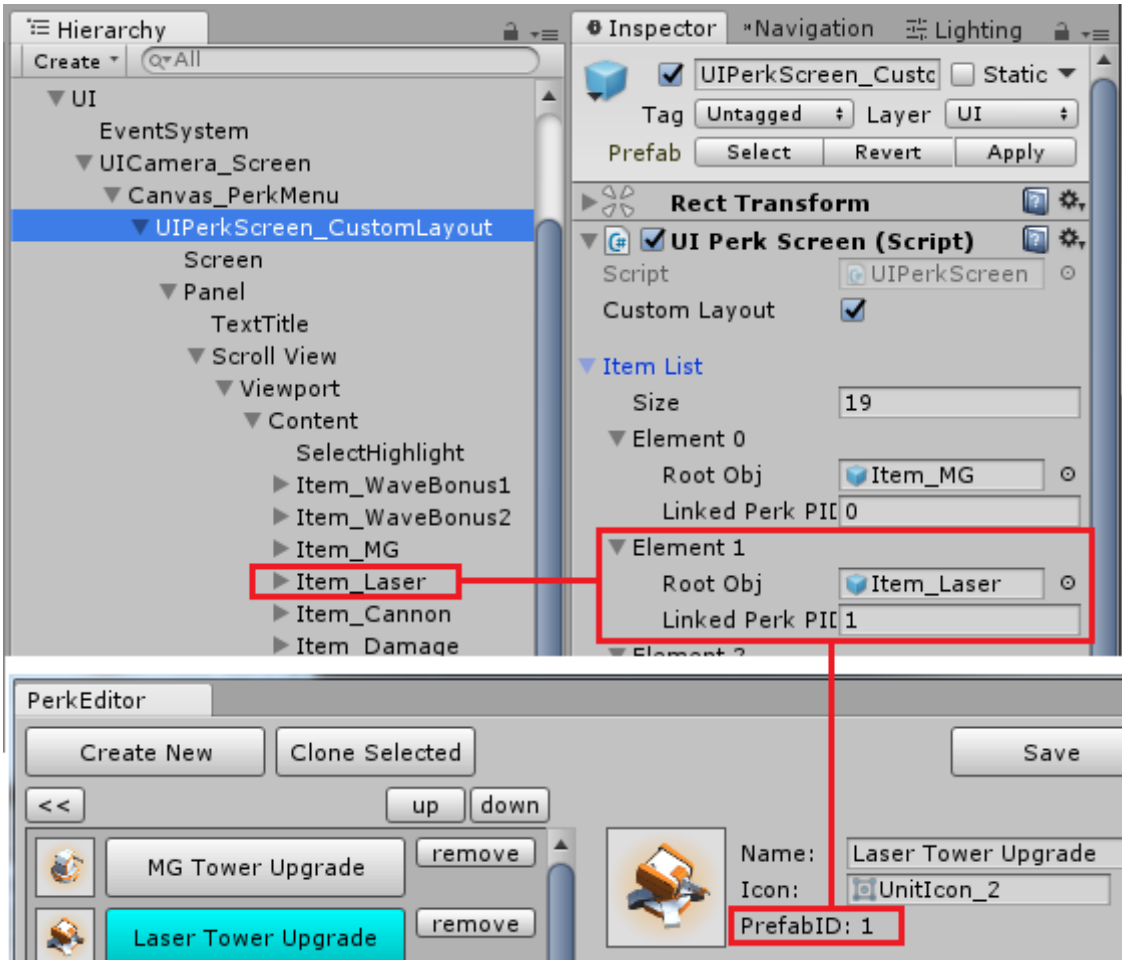


The UI is made to be touch control compatible; for touch interface on mobile devices. It's recommended you have the 'Touch Mode' enabled. Touch Mode is a special mode that, when enabled, requires double tap on buttons to trigger build/ability buttons (first tap to show the tooltip, second to confirm). This is specifically made to solve the issue where on touch devices, it's impossible to hover over a button with a mouse cursor. The UI should scale automatically in most cases. However, you might want to disable '*Limit Scale*' in case the UI elements get too small on devices with smaller screens.

Perk Menu

By default, the perk menu will simply list all available perks from PerkManager. However, you can override that and arrange your own custom layout, just like the one shown in the demo scene (*TBTK/DemoNScenes/Demo_Persistent_PreGame).*

To do that, first check the '*Custom Layout*' option in UIPerkScreen. Then you can add your own button(s) in the ScrollView and assign them to the '*Item List*' in UIPerkScreen. Note that each item will have a 'Linked Perk PID (Prefab ID)', the PrefabID of the perk the item corresponds to. You will need to manually match that to the PrefabID of the perk in PerkEditor, as shown as the image below. You can automate this process by using the '*GeneratePerkButtons*' button in UIPerkScreen.



Note that under each 'Item' game object, there are 'Connector' and 'ConnectorBase' 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 'ConnectorBase'. 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 ConnectorBase objects are the (grey) lines indicating the upgrade perk at the end of the path is unavailable, while the Connector objects are the (orange) lines indicating the next connected upgrade(s) is available. When an item is purchased, the ConnectorBase object will be disabled, revealing the Connector object behind it. For the player, it will seem like the line has turned from grey to orange, indicating that the link between two items has been activated and is thus available for purchase.

INTEGRATING TBTK TO YOUR GAME:

TBTK is made to support custom integration; that is to allow you to easily transfer your own custom units into a TBTK scene and provide you the unit information when the battle is done. An example for how to do this has been set up in the scene Demo_Persistent (PreGame and Game). You can refer to the script *Demo_Persistent_PreGame.cs* for example code. The script is fully commented to help you understand it.

Before diving into the code, you will need to enable some settings on UnitManagerEditor. For each faction, there's an option for "*LoadFromCache*" and "*SaveToCache*". Enable "*LoadFromCache*" to indicate the faction will load the starting unit list from cache and enable "*SaveToCache*" to have the UnitManager cache the remaining units on the grid to be loaded into the subsequent scene.

To cache the starting unit for a particular faction in the next game scene, simply call the function:

UnitManager.CacheFaction ()
public static void CacheFaction(int factionIndex, List<Unit> unitList)

The factionIndex is the index of the faction in the intended scene. The value is displayed on UnitManagerEditor. The unitList is simply the list of unit prefabs to be used. The units specified will be used to override the starting units on the faction in question.

To retrieve the units from the cache, simply call the function:

UnitManager.*GetCachedUnitList* ()
public static List<Unit> GetCachedUnitList(int factionIndex)

The function will return a list of units being cached.

Please note that the cache system will always refer to the unit prefab instead of the unit instance.

THANK-YOU NOTE & CONTACT INFO

Thanks for purchasing and using TBTK. 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/TBTK/Contact&SupportInfo***". Just in case, you can reach me at k.songtan@gmail.com or TBTK support thread at the Unity forum.

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