

# 3D Tower Defense Starter Kit

by Rebound Games

v1.3

---

1.	Introduction.....	2
2.	First Steps.....	3
3.	Scene Setup .....	5
4.	Manager Objects.....	9
4.1.	Pool Manager .....	10
4.2.	Audio Manager.....	13
4.3.	Grid Manager .....	15
4.4.	Tower Manager .....	17
4.5.	Waypoint Manager.....	18
4.6.	Wave Manager .....	20
4.7.	Game Manager.....	23
4.8.	GUI.....	24
5.	Setting up Towers .....	31
5.1.	Tower Base.....	33
5.2.	Upgrade .....	34
5.3.	Projectiles .....	35
6.	Setting up Enemies .....	38
6.1.	Movement .....	40
6.2.	Properties .....	41
7.	Interactive Tower Control .....	42
8.	Mobile Setup .....	43
9.	Advanced Example Scene .....	45
10.	Where to go from here .....	48
11.	Contact .....	49
12.	Version History.....	50

*Thanks for your purchase!*

*Your support is greatly appreciated!*

## 1. Introduction

---

3D Tower Defense Starter Kit is designed for all users of Unity3d, who ever wanted to create a tower defense game. The goal of this project is to deliver you a game-ready prototype with all resources, so you are able to integrate your own assets and adjustments easily, on your way to your own game - this project is all you need to do so.

Included features are:

- All game files, free for commercial use (re-selling prohibited)
- Simple Enemy Logic (movement using [Simple Waypoint System](#))
- Advanced Tower Logic
- Interactive Tower Control
- Multiple Projectile Variations
- Custom Editor Scripts
  - Wave Manager
  - Pool Manager
  - Waypoint Manager
  - Audio Manager
  - Grid Manager
  - Tower Manager
  - Enemy, Tower & Projectile Setup
- NGUI Integration
  - (The contents of the NGUI folder is the work of Tasharen Entertainment)
- Smartphone Support ( tested on Android & iOS )
- Full Source Code in C# - every line documented

These features should cover the most requirements for a tower defense game. However, please note that a Starter Kit can't suit all game cases. You likely want to modify it to fit your needs and implement your own unique game and user interface mechanics. In the following chapters, this documentation explains all components involved in this kit, so you can see where you might want to start. Ready?

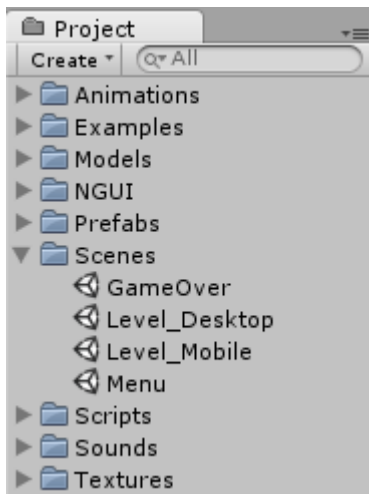
## 2. First Steps

---

If you are new to Unity3d, please take a quick break and get dirty with its main functionalities first, because this documentation will assume you have some basic knowledge regarding the interface and its editor tools.

📁 Import the 3D Tower Defense Starter Kit unitypackage into an empty project.

Once the import has finished, you'll see all project files listed in the Project panel.



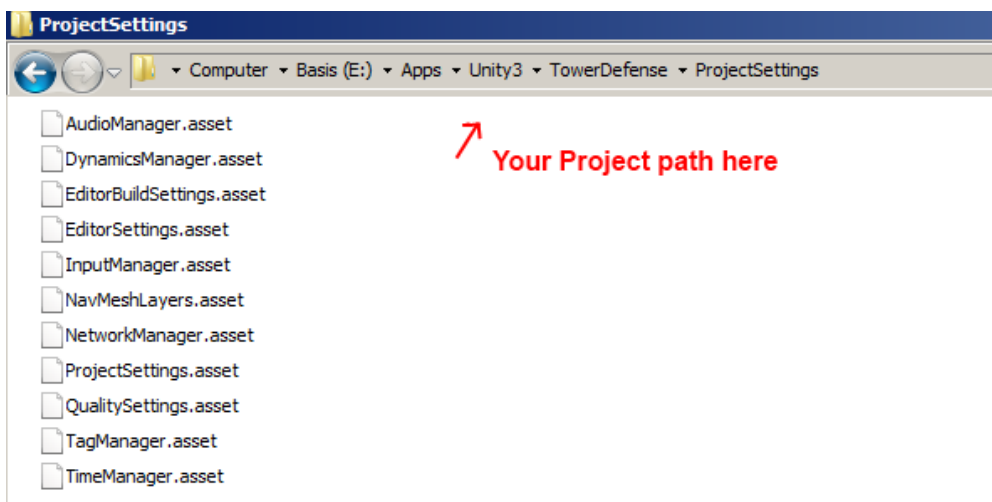
This kit contains 4 game scenes:


Menu, Level\_Desktop, Level\_Mobile and GameOver.

The example scenes for each important manager component are located in the folder named “Examples”, but we start investigating the game scenes first and take a look at them on the go.

**If you're using Unity4, please also read the README file!!**

To ensure all tags and layers are imported correctly, please open “ProjectSettings.zip”, close Unity, then extract it in your Project folder > ProjectSettings and overwrite what's there.



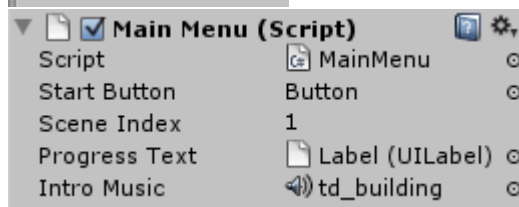
 Please open the scene “Menu”.

Users will see this scene first when playing the game. Besides a few gameobjects, there is not that much interaction here. You could extend this scene by providing various settings for the actual game like sound volume, graphic quality or difficulty settings.



The “Manager Scripts” gameobject holds all relevant components in this scene. The Audio Manager component is described in the next section separately, thus we select the “GUI” gameobject.

UI Root (2D) contains all NGUI elements.




Here we access a reference to the NGUI button “Button” used as start button. In its onclick event, it will load the scene with index value 1 of the build

settings. “Progress Text” visualizes the streaming progress for webplayer builds while loading level index 1. “Intro Music” is a sound clip that gets played via the Audio Manager at start. Finally, “GUI” has a child gameobject named “\_obsolete” attached to it. The obsolete version uses the built-in GUI with minimal differences, but it isn’t recommended since Unity’s GUI solution has serious performance impacts.


Let’s continue with the next scene.

### 3. Scene Setup

---

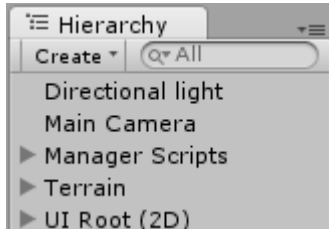
 Open the scene “Level\_Desktop”.

This is the main scene of the game. It contains all needed functionality to control every single aspect.

 Hit Play to see them in action.

Instructions...

- Move Camera:            arrow keys or scroll wheel pressed
- Rotate Camera:        hold right mouse button
- Zoom in & out:        scroll wheel
- Control tower:        right-click on tower (produces additional damage)

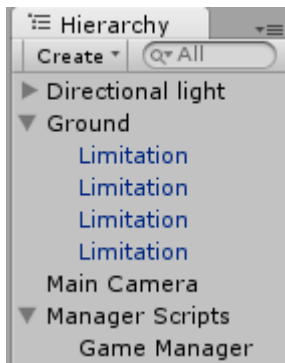


These are all gameobjects that make up the scene. A directional light source, a camera, all manager components, the terrain with all obstacles and the NGUI UI Root object.

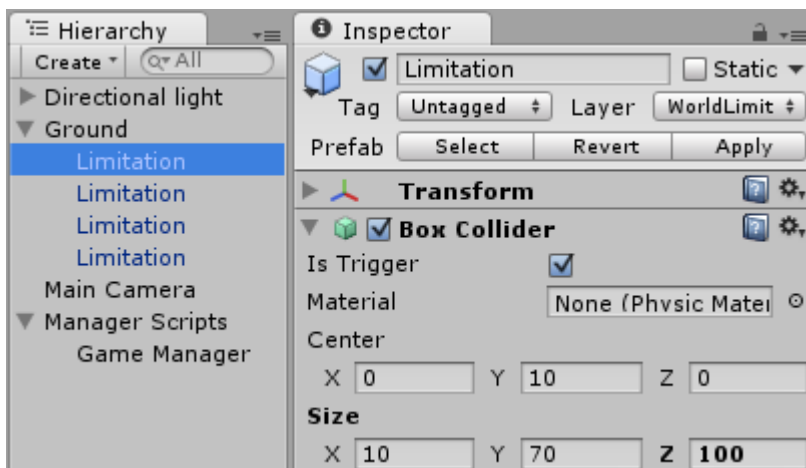


The Main Camera has a script attached to it called “Camera Control”. This script is responsible for the camera movement behavior. Also, it limits the space where users can navigate around. There’s a world setup necessary for this limitation.

 Please open the example scene “Example\_World” for investigating the world setup.

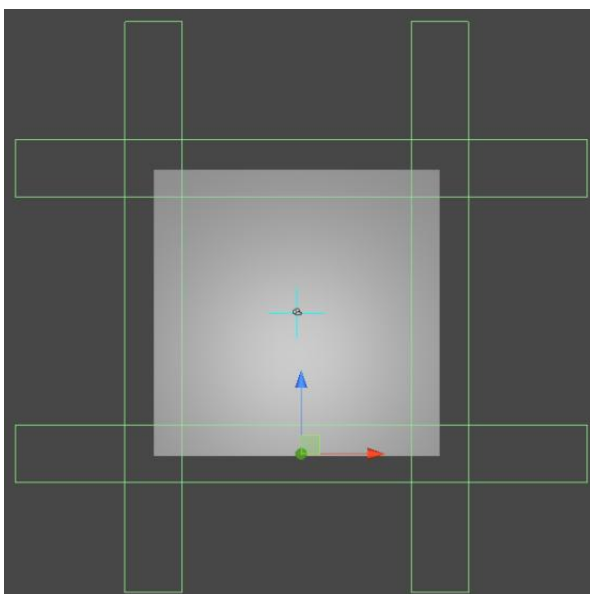


That's the minimalistic approach of the scene "Example\_World". Here we have a ground plate, the parent of a couple of "Limitation" prefabs, the camera and a "Game" manager object.




The "Limitation" prefab is just a gameobject holding a box collider marked as trigger.

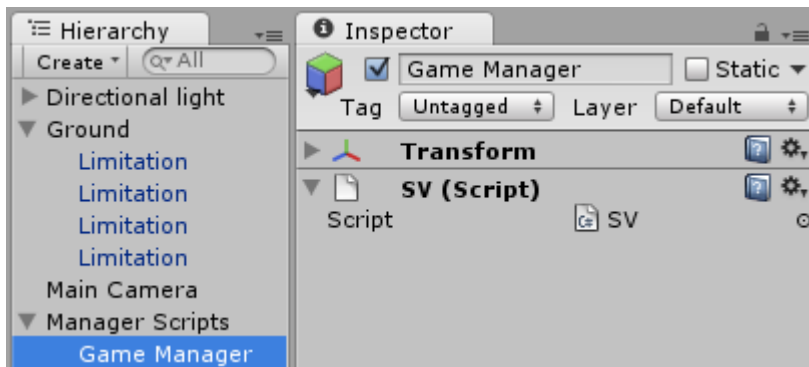
We assigned the Layer "WorldLimit" to the Limitation gameobject. This is very important, because the camera will check for obstacles in its way. If it hits a gameobject on the layer "WorldLimit", the user won't be able to move further in this direction.



The limitations always have to fully surround the playzone.

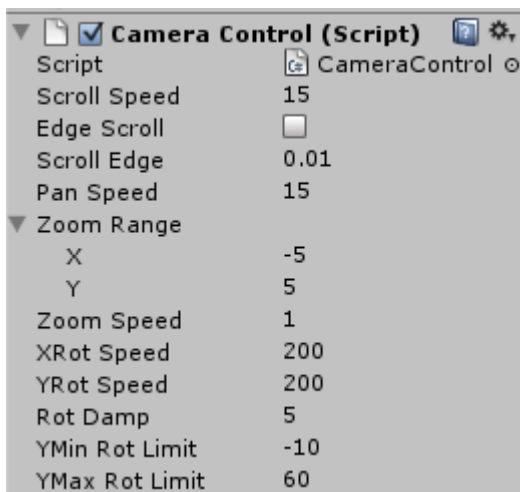
Here you can also see the light blue gizmo lines coming from the camera and pointing in each direction. Once they intercept with the box colliders, the camera stops moving.

Press play to see how the camera collides with the limitations. 



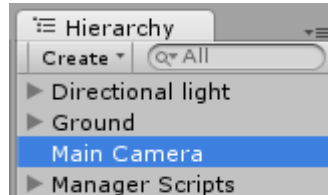
The script “SV” (short for static variables ) initializes the layers and masks at startup based on layer names, so the camera can dynamically use them to raycast against enemies,

towers, grids and obstacles. In this particular case, we only need it for the obstacle raycast, though. There are no custom actions needed in the script, besides you want to change the layers or mask names or want to add a few more.



Finally, the “Camera Control” script itself.

Only used in the desktop version, since the mobile version has its own camera control script using a joystick component.



Scroll Speed: movement speed while using “Edge Scroll”.

Edge Scroll: whether edge scrolling should be enabled for this camera.

Scroll Edge: percentage of the screen edge where edge scrolling becomes active.

Pan Speed: movement speed while panning using arrow keys or mouse wheel pressed.

Zoom Range: total height for zooming in and out, starting from min to max height.

Zoom Speed: movement speed while zooming in or out.


XRot Speed: rotation speed while rotating on the x-axis.

YRot Speed: rotation speed while rotating on the y-axis.

Rot Damp: rotation damping factor.

YMin Rot Limit: lower rotation angle limit while rotating on the y-axis.

YMax Rot Limit: upper rotation angle limit while rotating on the y-axis.

 Re-open the scene “Level\_Desktop”.

To recap the important things on the previous pages, here’s a quick summary:

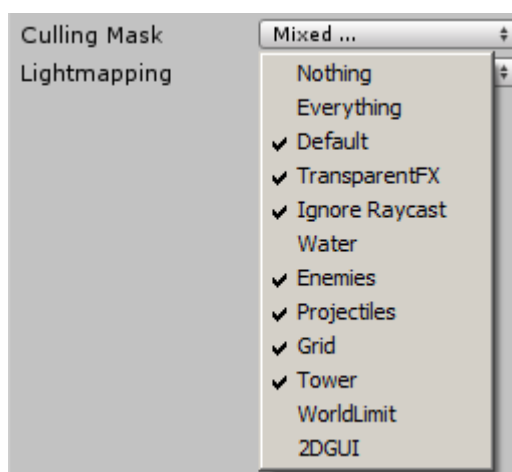


- All obstacles (the Terrain and all children) are in the layer “WorldLimit”, so the camera can’t move through them.
- Limitations surround the playzone.
- The script “SV” is attached to a gameobject in the scene for initializing layers and raycast masks.

Additional notes regarding the scene setup of “Level\_Desktop”:

- The terrain has the tag “Ground” assigned to it, so that in self-control mode, the crosshair stays flat on the ground, rather than looking to the camera.
- A lightmap was baked through Unity’s light mapping system that can be found under Window > Lightmapping.

The directional light was previously set to “BakedOnly” to bake the lightmap, then put to “RealtimeOnly” afterwards. Since it should not affect the baked static objects anymore, its culling mask was set accordingly:



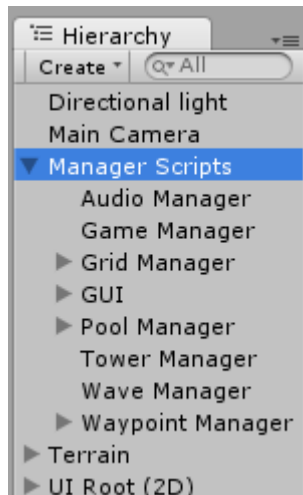
With these settings, the light should only affect all non-baked objects that get spawned at runtime.

These settings are just mentioned for the sake of completeness and are not required. You may want to change or skip them for your own level accordingly.



## 4. Manager Objects

---



Manager objects handle all kind of different functionality in the game. They are parented to the “Manager Scripts” gameobject and in this chapter, we will go through all of them one by one. Also there are many example scenes demonstrating the use of these components.

You can find prefabs of all of them in the folder:

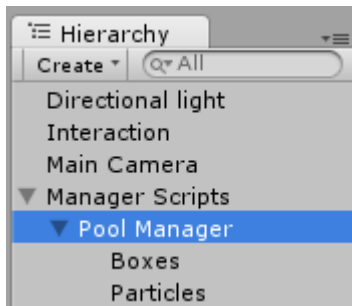
Prefabs > Managers

## 4.1.Pool Manager

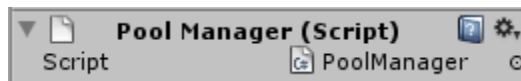
Script Connections: none

Garbage collection calls are expensive and cause stutter if used extensively. That means, Unity's Instantiate() and Destroy() commands can eat up performance very quickly. The Pool Manager avoids this bottleneck by instantiating objects once and then reusing the same instances over and over again.

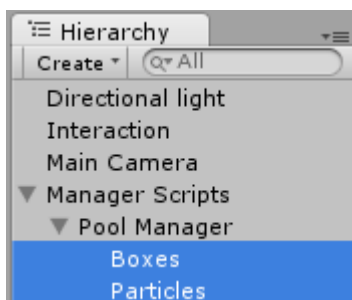
🔗 Open the example scene "Example\_Pool".



The Pool Manager is a single gameobject holding the script "Pool Manager". It manages its children, these are the actual pools.



This script contains a dictionary with all pools parented to the Pool Manager, initialized at startup. As well as methods for adding a new pool at runtime, checking for existing pools or destroying them. Please refer to "PoolManager.cs" for an overview of all methods.

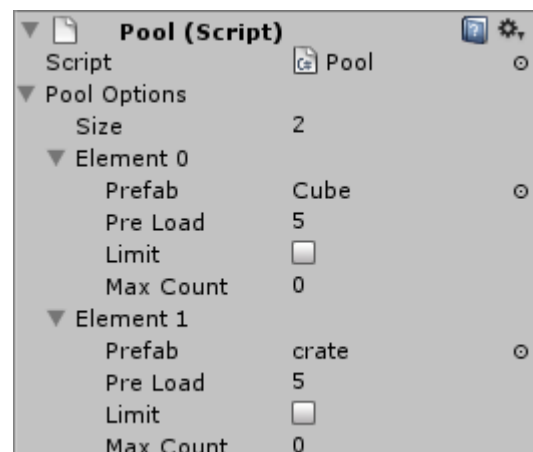


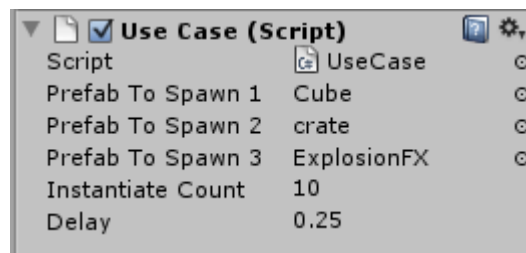
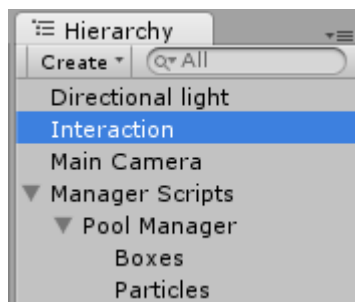
Pools are gameobjects with a script named "Pool" attached to them, parented to the Pool Manager. These gameobjects can be named whatever you like (with a few exceptions described later on).

Pools contain the actual prefabs you want to instantiate.

Pre Load: Determines how many instances should be initialized at start ready for use.

Limit: Whether the instance amount should be limited to the Max Count variable.





„Interaction“ has the script „Use Case“ attached that demonstrates the use of pools.

To spawn prefabs, you need a reference to the prefab and the name of the pool you want the prefab to be in. In the script above, both Cube and crate are spawned within the pool “Boxes”, whereas ExplosionFX is spawned within “Particles”. If a pool does not contain the passed prefab, it will automatically be added to that pool.

The actual commands are very easy and adapted from Instantiate() or Destroy().

```
PoolManager.Pools[pool name].Spawn(prefab, position, rotation);    //spawns an instance
PoolManager.Pools[pool name].Despawn(prefab instance);            //despawns an instance
```

Please refer to “Pool.cs” for all supported methods.

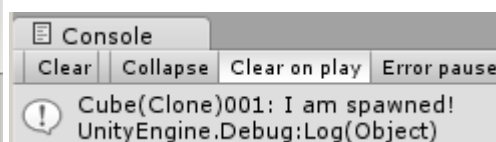
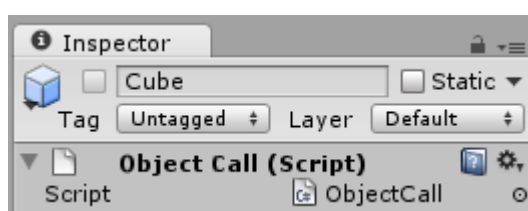


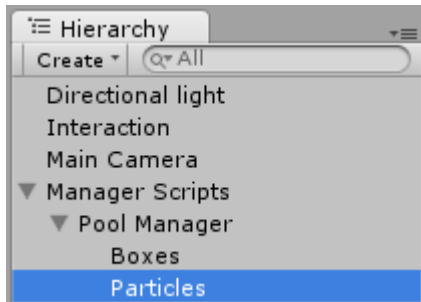
Hit play to see how the prefabs are spawned and despawned over time.

At start, every prefab gets preloaded in a disabled state. Spawn() activates the instance at the desired time and Despawn() disables it again. If the preload amount doesn’t suffice for activating more instances, the corresponding pool will automatically instantiate new instances. Also note how instances are reused several times.

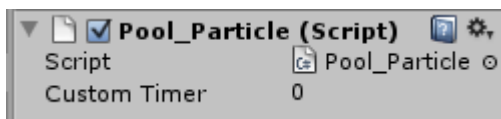


Whenever an instance gets spawned or despawned, the Pool sends a message to the instance and calls its OnSpawn() or Despawn() method respectively. In this scene, the Cube prefab uses this method to debug a message to the console.





In our Starter Kit, there are a few exceptions to the renaming of pools. Pools have to be named exactly as seen below for their specific use case: Enemies, Particles, Projectiles,... that's because the Wave Manager will search for a pool named "Enemies", TowerBase will search for a "Projectiles" pool, Pool\_Particle searches for "Particles" and so on. You can easily change the names in their scripts.

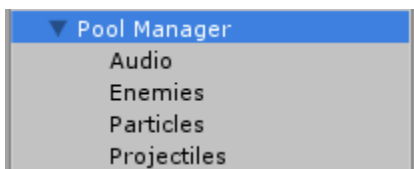


With particle effect prefabs, we likely don't want to manually call Despawn() when it finished simulating.

This script has to be attached to the main parent object of a particle effect and will automatically despawn the object when it's over. If the "Custom Timer" variable is not set, it will get the total duration based on all parented effects. Otherwise this variable serves the ability to despawn particle effect at a user defined time duration.



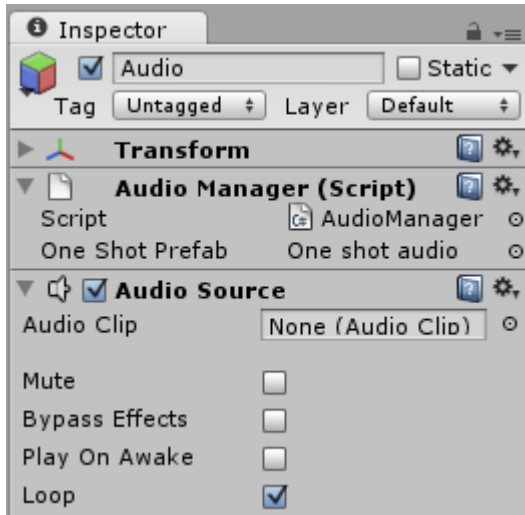
Please re-open the scene "Level\_Desktop".



Here we utilize pools for all objects that spawn very often: audio clips, enemies, particle effects and projectiles. UI elements for the progress map have their own pool, not parented to this manager object. We do not pool towers, because they won't get instantiated, destroyed and reused that often. But that depends on your game scenario, you could easily implement despawning of towers if you make sure that their properties reset at the time of activation (through OnSpawn() or OnDespawn() messages).

## 4.2. Audio Manager

### Script Connections: Pool Manager



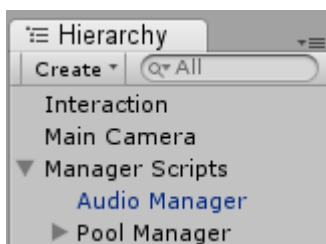
The Audio Manager plays all sounds within the game. They are called by their corresponding scripts, which pass a sound clip reference to the Audio Manager, which then plays the clip for them.

The Audio Manager gameobject consists of the Audio Manager script and an Audio Source component. Every 3D one shot sound that gets played on the Audio Manager will spawn a “One shot audio” prefab via the Pool Manager, which is placed at the desired position in 3D space. That’s why we need to set a reference to the prefab in the Audio Manager.

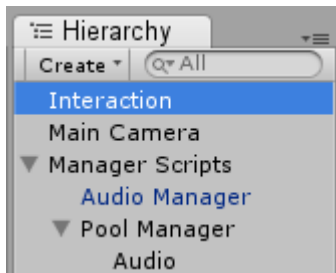
On the Audio Source, “Play On Awake” is unchecked, since we want to control the audio playback on our own. Additionally, “Loop” is checked: 3D sounds get a spawned prefab to run on and 2D clips will directly run on the Audio Manager reference, therefore the Audio Source will only be used for background music and this should obviously run in a loop.



Open the next example scene, “Example\_Audio”.



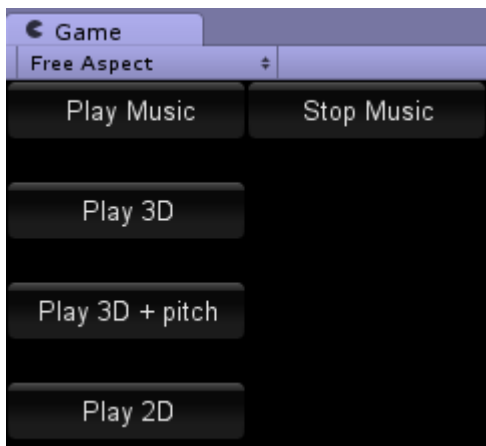
The only thing you need to do to play your sounds is to instantiate the Audio Manager prefab (located under Prefabs > Managers) and to set up the Pool Manager accordingly.



“Interaction” holds a simple sound test script with audio clips.



Press play to utilize the different kind of sound files. Also note how 3D sounds are randomly positioned in the scene and a pitch factor is applied to one of them.



There are a few methods available in the Audio Manager component. You can call them from every script using

[AudioManager.Play\(...\)](#) or [.Play2D\(...\)](#)

For a more detailed usage please investigate “AudioTest.cs” and “AudioManager.cs” directly.

### 4.3. Grid Manager

---

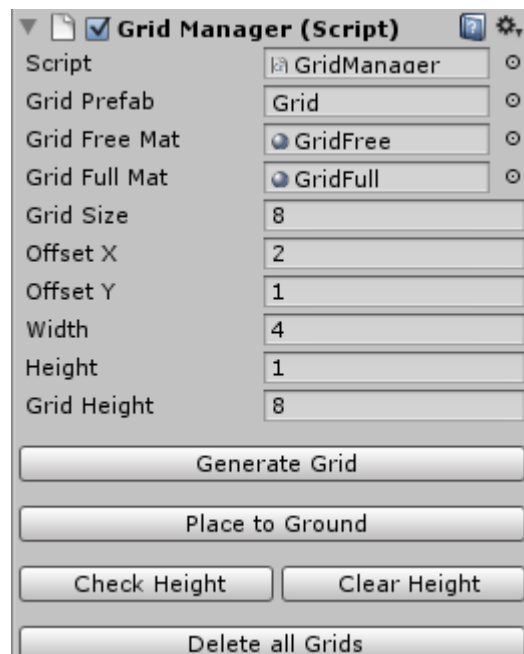
Script Connections: none

The Grid Manager simplifies the creation of grids. Grids are platforms positioned in the game where the user will be able to place towers onto. The GUI is responsible for raycasting against grids and the placement of towers, the Grid Manager only creates the grids.

🔍 Please open example scene “Example\_Grid”.



Select the Grid Manager gameobject and have a look at its editor component.



Grid Prefab: prefab to instantiate when generating a grid.

Grid Free Mat: material that indicates that the grid is available for towers, we can place a tower onto it

Grid Full Mat: material that indicates that the grid is occupied by towers, we can't place a tower onto it

Grid Size: size of the grid in total units

Offset X & Y: offset between grids when generating more than 1x1 (Width \* Height) grids

Width & Height: width and height square measures when generating multiple grids

Grid Height: allowed grid height value for finding ground through button “Check Height”

Generate Grid: generates the grid with above settings, they are parented to the manager

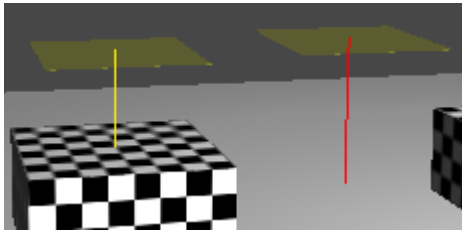
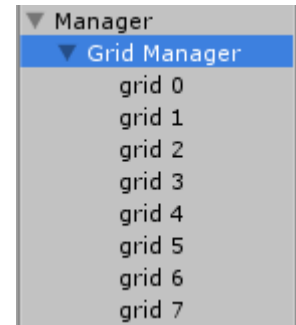
Place to Ground: causes a ground detection raycast and places all grids onto collided objects

Check Height: checks against ground with the length of “Grid Height” and switches the grid's material to “Grid Free Mat” or “Grid Full Mat” depending on if it hit something below it

Clear Height: frees all grids and switches their material to “Grid Free Mat”

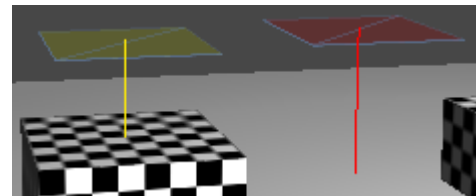
Delete all Grids: deletes all grids generated by the Grid Manager after a confirmation

For generating new grids, simply enter a “Width” and “Height” value and press the “Generate Grid” button. New grids are placed at the position of the Grid Manager gameobject and are automatically renamed and parented to it. You can then place them anywhere in your scene.



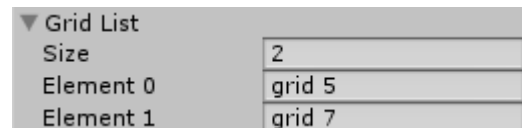
There are several pre-placed grids in this example scene. The yellow line indicates that the grid’s distance to the obstacle is within the value of “Grid Height”. A red line indicates that the distance exceeds its value.

All of them are not marked as occupied, even if two of them are obviously too high in the air. Let’s change that by clicking “Check Height”. Now, their material has switched to “Grid Full Mat”. If you do not want to use height checks, click “Clear Height”.



**Grid List** The Grid Manager script stores a list of all occupied grids in the scene.

Whenever grids are marked as occupied, through the height check or if the GUI placed a tower onto them, the corresponding grids are added to the



Grid List. Towers can’t be placed onto grids contained in this list and the GUI has to check it before placing a new tower onto the grid targeted by the user. Do not delete occupied grids in the scene without adjusting this grid list, because this will lead to inconsistency.

Lastly, grids have to be on the layer “Grid”, so that the GUI can raycast against them before buying a tower, for finding free grids in the scene.



Please re-open the scene “Level\_Desktop” to investigate the next manager object.



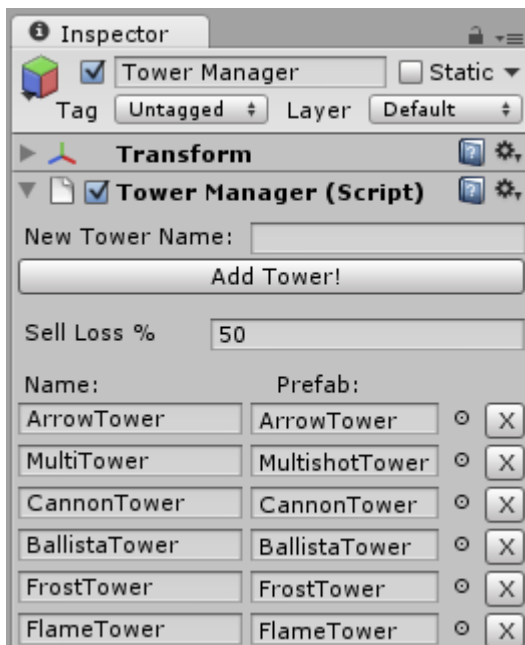
## 4.4. Tower Manager

---

Script Connections: none



The Tower Manager component stores all tower names, prefabs and TowerBase + Upgrade script references. This way, the GUI can easily access needed properties of tower prefabs at runtime for displaying them on tooltips. Also, this gameobject acts as a container for instantiated towers in the game.



Setting up towers is explained later on, but adding them to the game is very easy. Simply type in a tower name in the “New Tower Name” text field and press “Add Tower!” Now there’s a new row at the end of the list where you can select the corresponding tower prefab to instantiate during the game. “X” removes the selected row.

“Sell Loss %” is a percentage value that you will lose from your resources when selling a bought tower in the game, therefore you can’t infinitely buy and sell towers.

The properties of towers are controlled by their corresponding projectile prefabs.

ArrowTower: shoots one arrow.

MultiTower: shoots multiple arrows.

CannonTower: deals explosion damage on impact. Ground only.

BallistaTower: shoots one large arrow. Air only.

FrostTower: deals frost damage on impact and slows enemies down.

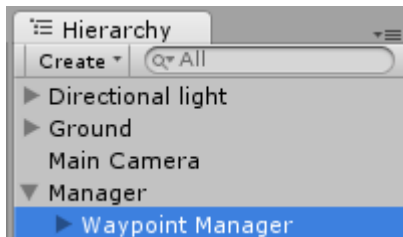
FlameTower: deals area-of-effect and damage over time. Ground only.

## 4.5. Waypoint Manager

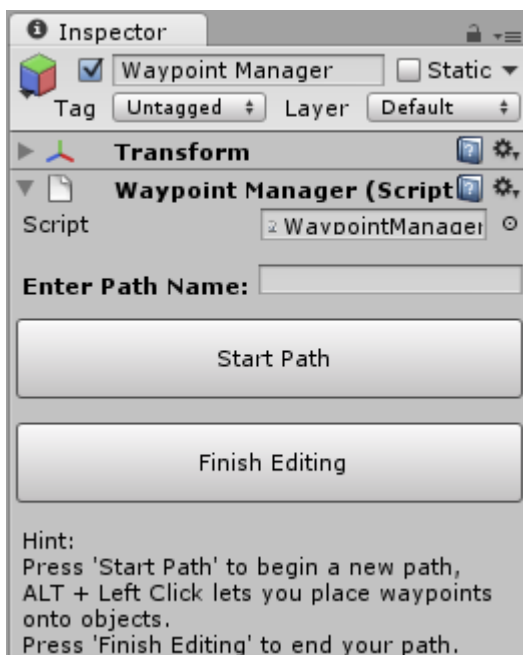
---

Script Connections: none

🔍 Please open example scene “Example\_Waypoint”.



The Waypoint Manager gameobject lets you create new paths for your enemies on which they should run on. This path creation tool and the movement script “TweenMove” for the enemies is based on our Simple Waypoint System plugin, however they are considerably modified for this Starter Kit.



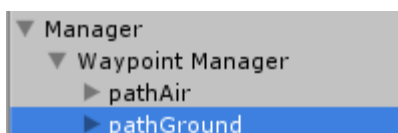
For entering the “path creation mode”, just name your new path in the “Enter Path Name:” text field and press “Start Path”.

You can place new waypoints for the newly created path by holding ALT and pressing the left mouse button.

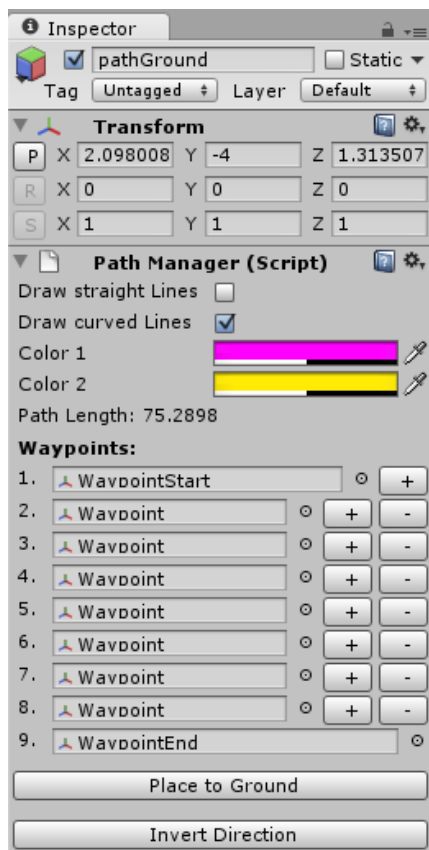
Please note that you always need some environment objects in your scene for being able to place waypoints onto them.

Hint: Do not select other objects while creating a path as this will skip the finalization process that renames the first and last waypoints.

To finish a path, click on “Finish Editing” instead.



There are already two paths in this example scene. After the creation part, paths are automatically added to the Waypoint Manager gameobject, expand it to investigate them.



The Path Manager component manages waypoints and the visualization of a specific path.

In the scene view all waypoints now get a small info box above them, including a number corresponding to this overview. You have the option to draw straight or curved lines between waypoints (this does not affect movement). The first color field is used for the starting and ending waypoint of this path, the second as line color. “Path Length” shows the full length of the path.

To move an existing waypoint, simply click its transform slot in this waypoint list, so it gets highlighted in the hierarchy and move it through Unity’s built in x/y/z handles.

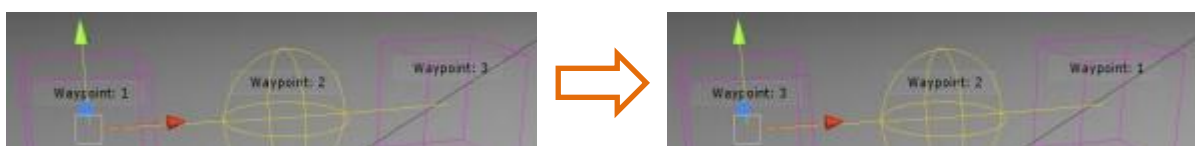
For adding new waypoints to an existing path, press the small “+” button next to a waypoint slot. This will create a new waypoint after the selected one.

Removing a waypoint is as easy as that. Just press the small “-” button next to the waypoint you want to remove. This action adjusts the waypoint array, deletes the selected waypoint from your scene and updates connections between them. The first and last waypoint of a path cannot be deleted, just delete the whole path gameobject instead.

The button “Place to Ground” will do exactly what it says: The Path Manager calls a ground detection for every method, which places them onto collided environment.



The last button, “Invert Direction”, inverts the order of waypoints which belong to this path.



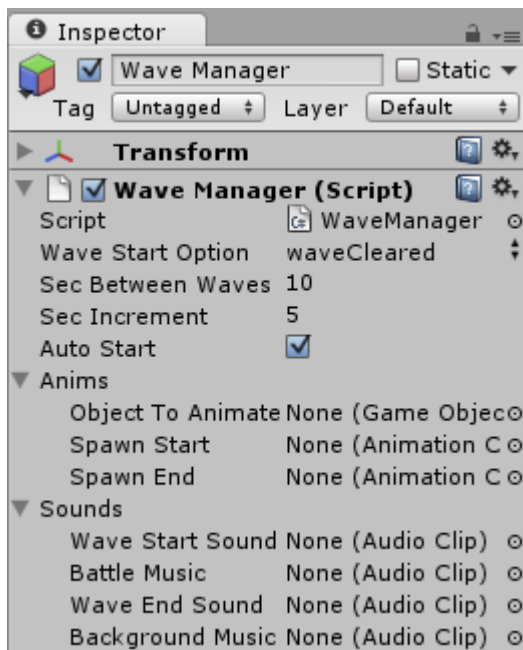
## 4.6. Wave Manager

Script Connections: Audio Manager, Game Manager (indirectly)

🔍 Please open example scene “Example\_Wave”.



This manager object lets you define waves of enemies spawning during the game. In this example scene we utilize the same paths demonstrated in the last section for letting enemies walk on them. Also, the scene is prepared with an Audio Manager that will play the wave sounds, as well with a Pool Manager that defines one test enemy pool.



Here you can set basic settings which affect all waves. During the game, “Wave Start Option” regulates the spawning of new waves. There are several wave options available:

waveCleared: wait till current wave is cleared

interval: wait defined interval before starting a new wave

userInput: wait for the user pressing a button when all current enemies were spawned

roundBased: wait for the user pressing a button when the current wave is over

Sec Between Waves: time interval between waves, used by option waveCleared & interval

Sec Increment: the time interval “Sec Between Waves” gets incremented by this value after each wave, so the longer the game lasts, the longer the user has to wait between waves

Auto Start: If checked, automatically starts the first wave at game launch

Anims class: Here you can trigger an animation component of an object at wave events

Object To Animate: object that holds the animation component

Spawn Start: Animation to play on the object when the spawning starts

Spawn End: Animation to play on the object when the spawning ends

Sounds class: Sounds to play at different wave events

Wave Start Sound: sound to play whenever a new wave starts

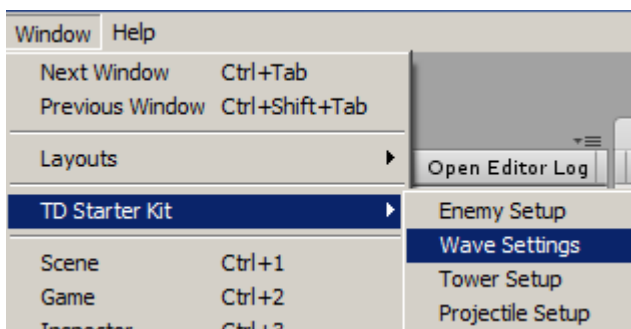
Battle Music: looping music to play during a wave

Wave End Sound: sound to play whenever a wave ended

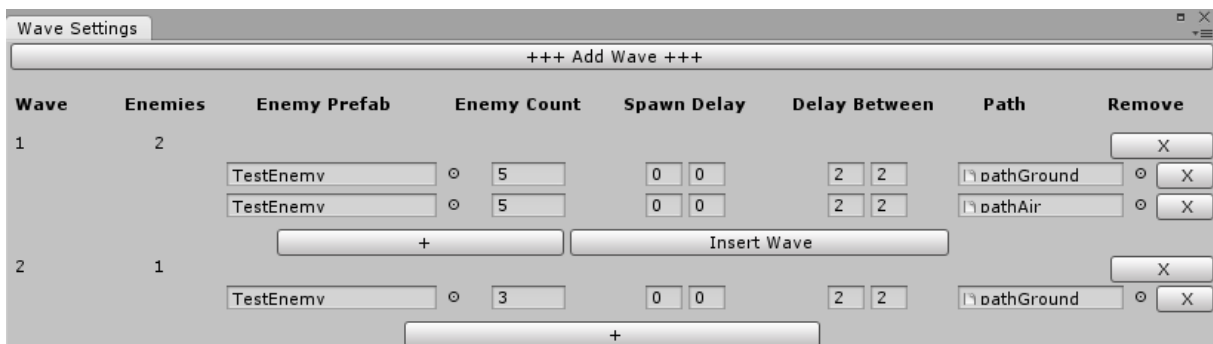
Background Music: looping music to play after a wave ended

As this example scene does not have a UI, we continue with a wave start option that does not require a button.

The Wave Manager (WaveManager.cs) plays sound clips corresponding to wave events during the game via the Audio Manager. Additionally, it sets and accesses various Game Manager variables (listed in the next section) and checks if there are still enemies alive or if there is still a wave left, before spawning the next wave. At the last wave, this component toggles Game Manager's game over flag. This will make more sense after the next section.



Let's open the "Wave Settings" editor window under Window > TD Starter Kit > Wave Settings.



This editor window needs a Wave Manager in the scene to show up correctly. Here you can specify what type of enemies should appear at each wave. They are instantiated at the first waypoint of a path and then walk this path till the end using their movement scripts.

+++ Add Wave +++: inserts a new wave at the end

Wave: lists the wave index

±: inserts a new row in this wave

Insert Wave: Inserts a new wave after this one

Enemies: amount of different enemy types in a wave

Enemy Prefab: enemy prefab to instantiate in this wave

Enemy Count: amount of enemies to spawn of the prefab in the same row

Spawn Delay: delay at start before spawning this enemy type, random from min to max

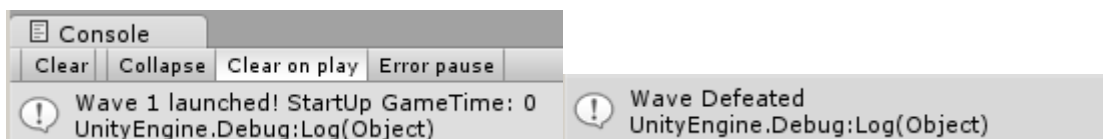
Delay Between: delay between the instantiation of enemies, random from min to max

Path: the path on which this enemy type should run on, created by Waypoint Manager

Remove: button to remove a whole wave or only a single enemy row within a wave



Hit play to start the waves automatically on launch (due to Wave Manager's "Auto Start" option) and note how enemies walk on the path till the end before they get removed after playing a die animation.



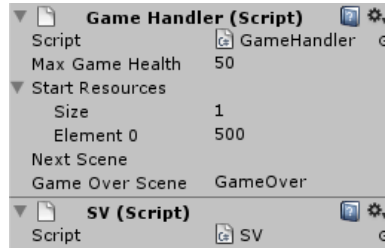
## 4.7. Game Manager

---

Script Connections: none



Please re-open the scene “Level\_Desktop”.



The Game Manager and its “Game Handler” component stores all relevant game info and decides if the game was lost or won. Including how many enemies were killed, our

health, resources and current wave count.

“SV” was explained in chapter 3. For a list of all variables controlled by the Game Handler, please open “GameHandler.cs”.

Max Game Health: the user’s health points at start

Start Resources: initial value of resources for buying towers (here: one resource)

Next Scene: next scene/level to load. If no scene was set, “Game Over Scene” is used instead

Game Over Scene: scene to load when the game is over

The Game Handler contains a method called “CheckGameState()”, that gets invoked at game launch and continues to check our health points every second. If we have less than or equal to 0 health (we lost), it finishes the game and loads the game over scene. If its “gameOver” variable was set to true through the Wave Manager (last wave is over), it examines our remaining health points and decides if we won the game. Also, the Game Manager contains two static functions that get called by enemies:

SetResources(int index, float points): called on enemy death to increase a specific resource

DealDamage(float dmg): called if it has reached its destination for subtracting resources

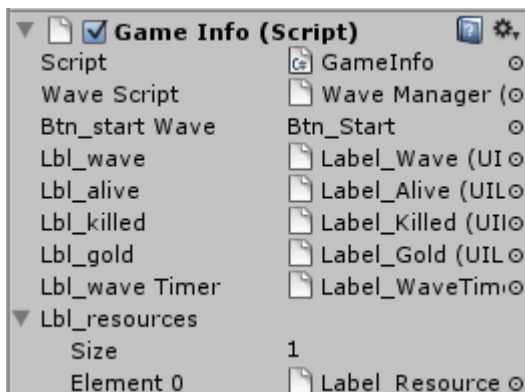
## 4.8. GUI

---

Script Connections: Pool Manager, Audio Manager, Grid Manager, Game Manager, Tower Manager, TowerBase, Upgrade, SelfControl, NGUI

The GUI combines nearly all of the previously discussed manager objects, therefore it is the most complex script in this kit. It utilizes NGUI for visualizing all the different UI elements. This documentation won't explain how to use NGUI, if you are not familiar with it please take a look at its extensive documentation and video tutorials here: [Link](#).

This Starter Kit uses the distribution version of NGUI. If you own NGUI, the first thing you likely want to do is to get rid of its watermark. To do that, simply open a new scene, delete the NGUI folder of this kit and import your full version.



Let's begin with this script, that displays all game related information on the screen as well as the button to start new waves.

All of these properties are references to either NGUI buttons or NGUI labels. You can position them anywhere you wish on the screen.

It mainly accesses the GameHandler component to display its static variables listed below. Also, it checks the current wave status and disables or enables the start button accordingly.

Wave Script: reference to the Wave Manager for being able to check the current wave status

Btn\_start Wave: NGUI button for starting the next wave on its OnClick event

Lbl\_wave: label for displaying the wave index and total amount of waves

Lbl\_alive: label for displaying the amount of enemies currently alive in the game

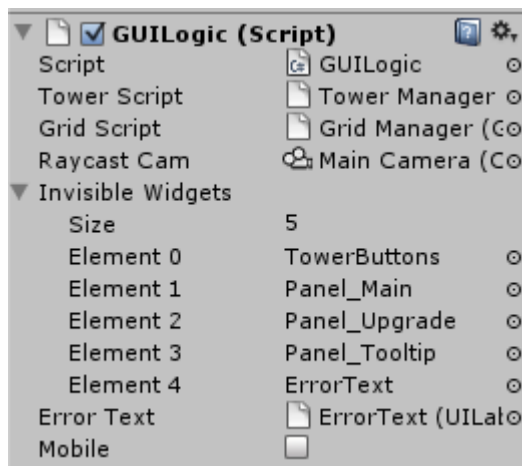
Lbl\_killed: displays the amount of enemies that were killed during the game

Lbl\_gold: displays user's health

Lbl\_wave Timer: displays the time until the next wave starts

Lbl\_resources: displays the amount of resources for building and upgrading towers





This script contains all GUI related logic, which control the user's actions during the game. It serves as base for your own GUI implementation that will be built on top of this script.

Tower Script: reference to the Tower Manager for placing new towers when they were bought

Grid Manager: reference to the Grid Manager for checking if a targeted grid is still free

Raycast Cam: camera utilized for raycasting against grids and towers, this is mostly the main camera

Invisible Widgets: a list of UI elements that should start invisible and disabled for activating them during the game on other UI events

Error Text: reference to a NGUI UILabel for showing a message to the player during the game

Mobile: If toggled, can be used to enable a mobile device control scheme. (see chapter "Mobile Setup")

### Important methods:

**void** RaycastGrid(): checks if the mouse is over a grid, sets "currentGrid"

**void** RaycastTower(): checks if the mouse is over a tower, sets "currentTower"

**public void** CancelSelection(**bool** destroySelection): deactivates/destroys all selections

**public bool** CheckIfGridIsFree(): returns a boolean whether or not "currentGrid" is free

**public void** SetTowerComponents(**GameObject** tower): gets necessary tower components

**public bool** AvailableUpgrade(): returns a boolean whether there's a level to upgrade left

**public bool** AffordableUpgrade(): returns a boolean whether the next level is affordable

**public float[]** GetUpgradePrice(): returns an array of resources needed for the next level

**public float[]** GetSellPrice(): returns an array of resources at which the tower gets sold

**public void** InstantiateTower(**int** clickedButton): creates a tower depending on the button

**public void** BuyTower(): buys the floating tower and places it on top of the current grid

**public void** UpgradeTower(): upgrades the selected tower to the next level

**public void** SellTower(**GameObject** tower): sells the tower passed in as argument

**public IEnumerator** DisplayError(**string** text): displays a text on the screen for a few seconds

**public IEnumerator** FadeIn/FadeOut (**GameObject** gObj): fade in/out NGUI widgets



GUIImpl.cs is the implementation of GUILogic.cs and combines the base logic with customized methods.

--Build Fx: particle effect on placing a new tower

Panels: various panels with multiple UI elements

--Main: exit menu with main and cancel button

--Upgrade Menu: panel for upgrading a tower

--Tooltip: basic tooltip panel when selecting a tower

Buttons: for interacting with parts of the game

--Main Button: to show or hide all tower buttons

--Tower Buttons: the parent of all tower buttons

--Button\_sell: sells the selected tower

--Button\_upgrade: upgrades the selected tower to the next level, if possible

--Button\_abort: closes the upgrade menu

--Button\_exit: button to leave the self-control mode

--Mobile shoot: button to shoot with towers while controlling them, only for the mobile version, thus empty in the desktop version

Labels: labels for displaying various tower properties in tooltips

--Tower Name: the name of the selected tower

--Properties: tower properties, such as damage, projectile type or upgrade properties

--Price: the price of this tower for purchasing it, for each resource

--Sell Price: the amount of resources you get when selling this tower, for each resource

Sound: sounds to play on various GUI events

--Build: played if a tower was bought

--Sell: played if a tower was sold

--Upgrade: played if a tower was upgraded

Control: self-control variables, necessary for initialization

--Crosshair: crosshair prefab to instantiate when controlling a tower

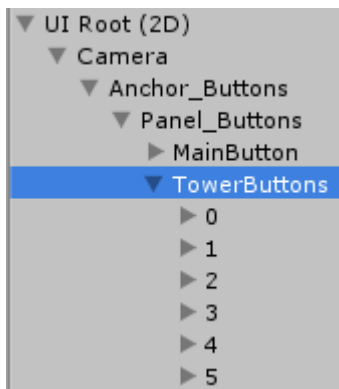
--Aim Indicator: prefab to indicate a visible line to the crosshair when controlling a tower

--Tower Height: height above a tower where the camera gets moved to when controlling the tower

--Rel Slider: UISlider, used as reloading bar when controlling a tower

--Rel Sprite: UISprite, used to change the sprite color and differ between reload/ready

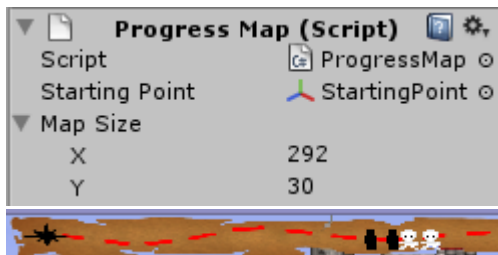
You can position and replace every widget as you wish, there are only two limitations of this GUIImpl script to take into account, regarding naming conventions of tower buttons.



If a tower button was clicked, so the user wants to buy a tower, the GUI needs to know what tower was clicked. In this case, we renamed all buttons with numbers starting from zero. The button then calls the method “CreateTower([GameObject](#) button)” in the script, that parses this number to an int value and passes it to the GUILogic script, that accesses the corresponding tower slot of the Tower Manager list.

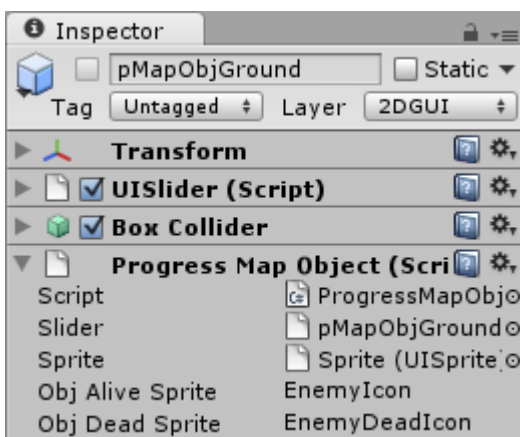
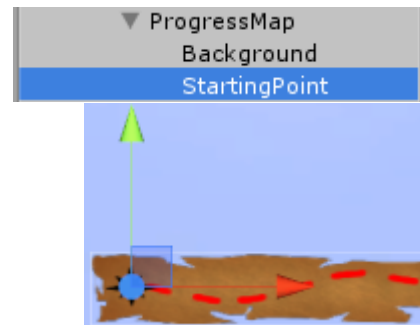
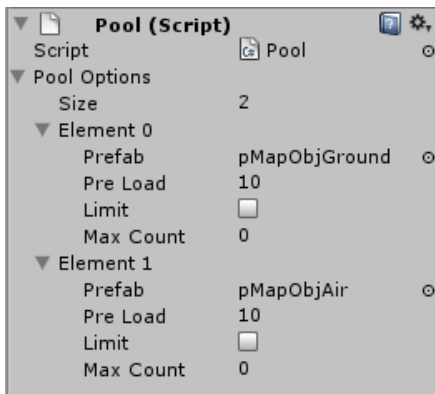
The second limitation concerns the “exit game, go back to main menu”-button, that has to be named “MainMenu”, called by ExitMenu([GameObject](#) button). You could rename it, if you change this method in GUIImpl.cs accordingly.

If you want to change the GUI, take a look at GUIImpl.cs first. It isn’t necessary to modify the basic methods of GUILogic.cs if you just want to switch, add or remove buttons to the game, since GUILogic.cs is mostly independent from any GUI code.



Continuing with the next GUI component, the Progress Map draws the enemy's path progress on a straight map.

This component does use the PoolManager to instantiate prefabs of new "Progress Map Objects" on the map. "Starting Point" is just a gameobject in the GUI, which contains the pool, and is positioned where on the map these prefabs should spawn. "Map Size" is the size of the map on the screen. Since the starting point has an offset from the left, X is a bit smaller than the actual map texture. Modify, test these values and repeat until they fit.



A progress map object consists of a NGUI UISlider, a box collider and the "Progress Map Object" script.

With multiple enemies on the progress map, each slider value represents the path progress for this object, going from zero (start) to one (goal). TweenMove.cs calculates these values per object.

Slider: reference to the UISlider component for this object

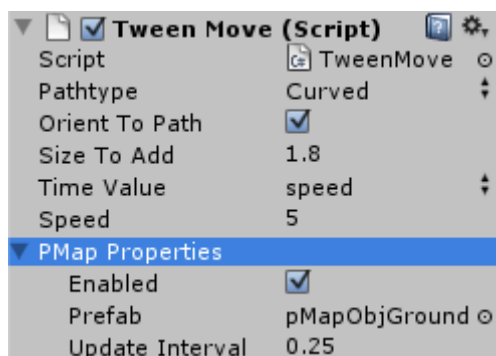
Sprite: reference to the UISprite component, usually the foreground texture of the slider

Obj Alive Sprite: the texture name in the selected NGUI atlas used while moving on the map

Obj Dead Sprite: texture name in the selected NGUI atlas used to display on object's death

The prefabs “pMapObjGround” and “pMapObjAir” are using different icons while moving on the progress map. You can duplicate these prefabs to create your own progress map objects, but make sure to adjust the slider’s “size” values depending on the size of your map texture.

To add an enemy to the progress map, you simply have to setup a progress map and check TweenMove’s “Enabled” checkbox for your enemies. Then, drag the “Progress Map Object” prefab in the slot “Prefab”. “Update Interval” is the time frame at which the progress gets updated.



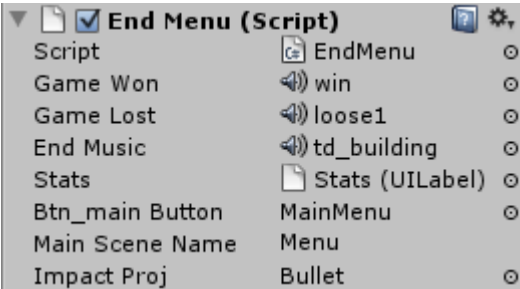
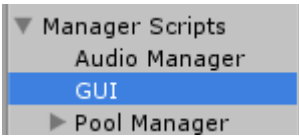
The movement script has to add itself to the progress map on game launch, then it calculates its path progress during the game and forwards this value to the progress map. The progress map only maintains one dictionary for all objects that use the progress map and updates their positions on the map corresponding to the value passed in. For a more detailed programming overview, please open “ProgressMap.cs”, “ProgressMapObject.cs” and “TweenMove.cs” and take a look at its heavily documented code.

As for the Game Over scene, the GUI transmits its game related values to the last scene and sets a NGUI label with this played game data.

For this implementation, we set the “GameHandler” script to “DontDestroyOnLoad”, so it does continue to exist in the next scene. Then we can access all static values of this script throughout the game.

⬅ Please open the last scene, “GameOver”.

There’s a very short script attached to the GUI gameobject, named “EndMenu”.

		<p>This script plays a game over sound, sets the game statistics to a label and instantiates a cannonball projectile every few seconds for fun.</p>
--	--	---

Game Won: sound to play if GameHandler’s gameOver variable is true

Game Lost: sound to play if GameHandler’s gameOver variable is false

End Music: background music to play after Game Won / Lost sound

Stats: the label that will represent the game statistics

Btn\_main Button: button for returning back to the main scene

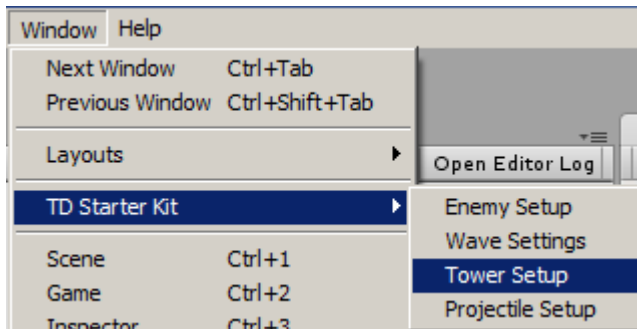
Main Scene Name: name of the main scene

Impact Proj: projectile to instantiate as fun effect

Please investigate “EndMenu.cs” for further programming hints.

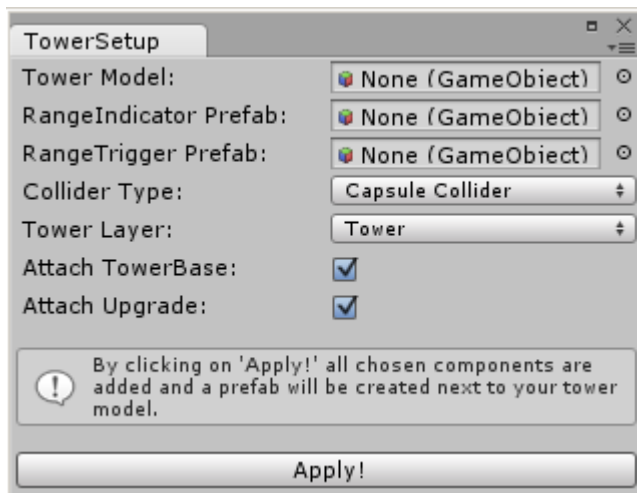
This concludes the introduction to all GUI components and script connections between them in this Starter Kit, so we continue with editor widgets and more specific settings regarding towers, projectiles and enemies.

## 5. Setting up Towers



Towers consist of multiple components and settings regarding colliders and layers. There's an editor widget that simplifies the process of creating a tower prefab.

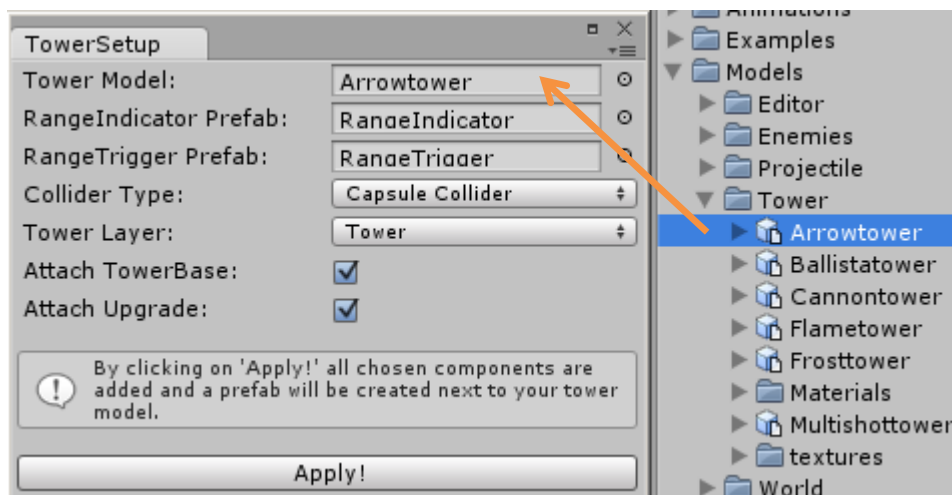
You can find it under Window > TD Starter Kit > Tower Setup.



Tower Model: your model (fbx, obj, etc.) to use as a tower prefab  
RangeIndicator Prefab: gameobject that visualizes the range of the tower for users  
RangeTrigger Prefab: collider that maintains a list of all enemies in range  
Collider Type: collider for raycasting against the tower and selecting it during the game

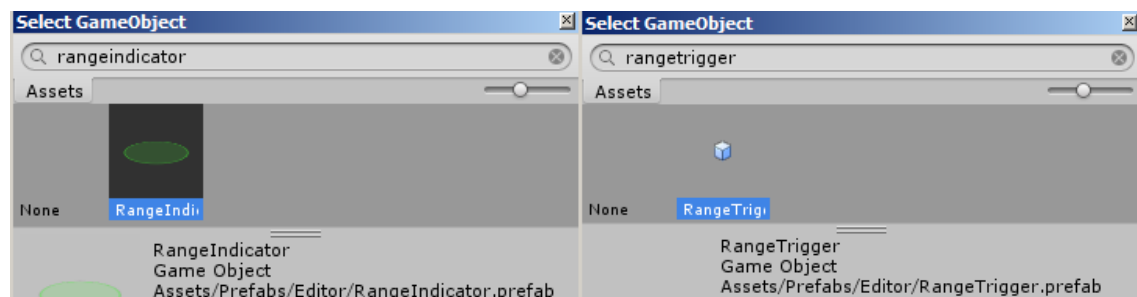
Tower Layer: layer indicating that this prefab will be a tower. "Tower" by default  
Attach TowerBase: whether the basic tower script should be attached to the tower  
Attach Upgrade: whether the upgrade script should be attached to the tower

First, let me explain some tower concepts in this Starter Kit. Every tower holds a list of all enemies in range. It can attack those enemies in a user defined interval. A collider, the RangeTrigger prefab, adds new enemies to this list and starts invoking the "attack-mode" whenever they enter the tower's range. The RangeIndicator Prefab has no other purpose than showing the range. TowerBase.cs controls all tower logic, such as filtering the list of enemies regarding field of view and shooting, meaning the instantiation of projectiles. Upgrade.cs does contain properties for this tower, like damage, shoot delay and price. It also updates the size of RangeTrigger's collider at each tower level. Ok, let's create a new tower!



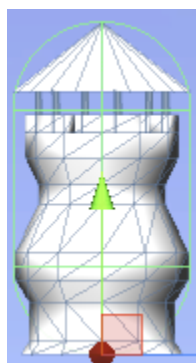
Drag & drop the arrow tower model located under Models > Tower in the “Tower Model” slot.

Assign the range indicator and range trigger to the corresponding fields and leave the other settings as they are.



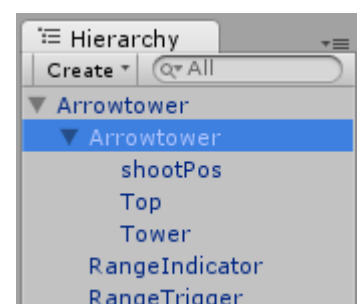
RangeTrigger holds a script named “RangeTrigger.cs”. If the RangeTrigger prefab is parented to a tower, it will recognize other enemies in range and add them to the attackable enemy list of the tower. Then it’ll start the tower’s StartInvoke() method in TowerBase for invoking field of view checks and starting to shoot. It also adds a reference of the tower to the enemy’s tower list at the same time.

The small info text above the button implies that we are going to create a prefab. Click “Apply!”. Now a prefab has been created next to our tower model. Drag & drop it into the scene to make further adjustments.



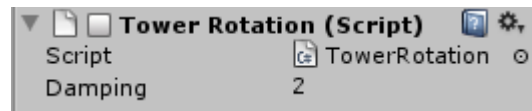
Disregard the textures at this point, these were disabled on the fbx file.

The Tower Setup script automatically tries to adjust the selected collider to match the tower model. If it does not fit exactly, you have to resize it accordingly. Our GUI will raycast against this collider for selecting the tower.





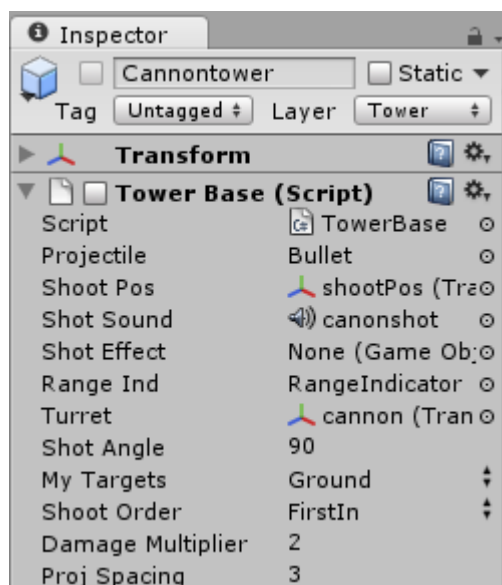
The main gameobject has to be located on the bottom of the tower, so it gets directly placed on the ground when buying a tower. RangeTrigger has to be placed at the middle of the tower, as the range should start from the middle. But you could also place it where you wish. If you have a rotating tower part (like a turret), you can attach the script “TowerRotation” to this part (e.g. see BallistaTower as a referencer). Please also refer to other towers for investigating their particular setup and gameobjects.



## 5.1. Tower Base

---

Script Connections: Upgrade, TowerRotation, (Enemy-) Properties, Projectile, Pool Manager, Audio Manager, (RangeIndicator)



As explained a few pages back, this component controls everything that has something to do with the tower logic.

Projectile: projectile prefab to spawn when shooting

Shot Pos: position to spawn the projectile

Shot Sound: sound to play on shot

Shot Effect: effect to instantiate on shot

Range Ind: range indicator prefab

Turret: rotating tower part (optional)

Shot Angle: field of view restriction for turrets

My Targets: attackable enemy types for this tower

Shoot Order: sorting of enemies by preference

Damage Multiplier: damage gets multiplied by this value when self-controlling this tower

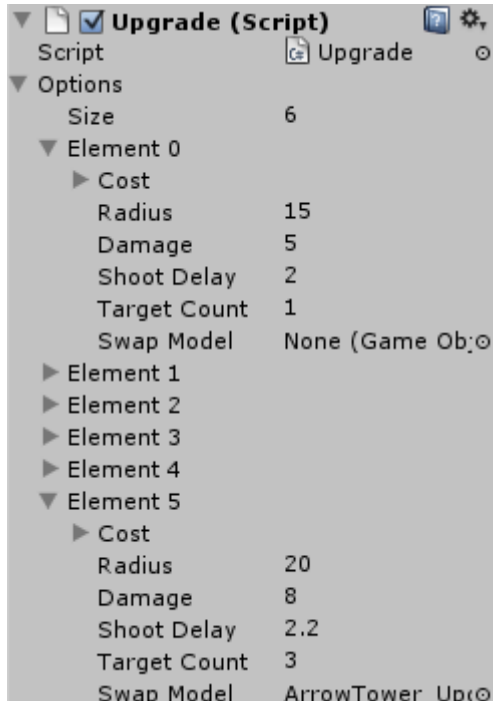
Proj Spacing: spacing between multiple projectiles when self-controlling this tower (the amount of projectiles is set in Upgrade)

Please open “TowerBase.cs” for an insight into programming the tower logic.

## 5.2. Upgrade

---

Script Connections: TowerBase, RangeTrigger, (RangeIndicator)



The Upgrade script resizes RangeTrigger's range at start and for every level. It also sets various settings for the tower and its projectile in an upgrade-oriented way.(the image shows 5 upgradeable levels)

Cost: in the first element, this value determines the price to buy this tower. At later levels, this value determines the cost to upgrade this tower (per resource, resize array for multiple resources).

Radius: range for adding enemies nearby

Damage: damage value that gets forwarded to the projectile prefab at spawn

Shoot Delay: delay between two shots, meaning projectile instantiations

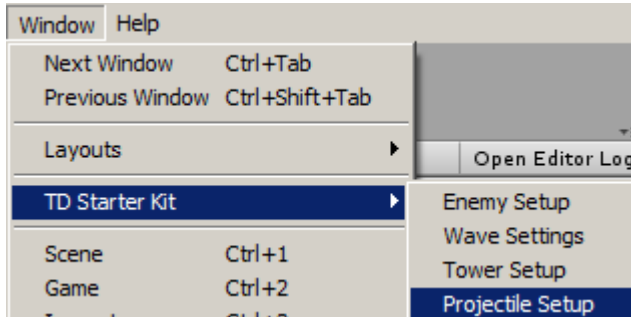
Target Count: amount of attackable enemies at the same time

Swap Model: switch the tower to the new tower prefab defined here

For swapping out the tower with another tower model at an upgraded level, you only have to attach the TowerBase script to the new tower prefab. The Upgrade script will be carried over to the new tower automatically. This allows you to specify the Upgrade script once and use different tower options or even different projectiles per upgrade. In the above image taken of the ArrowTower prefab (brown roof), it will switch to a multi-shot arrow tower at the fifth level (red roof). Please investigate the "ArrowTower" and "ArrowTower\_upgraded" prefabs for further understanding.

## 5.3. Projectiles

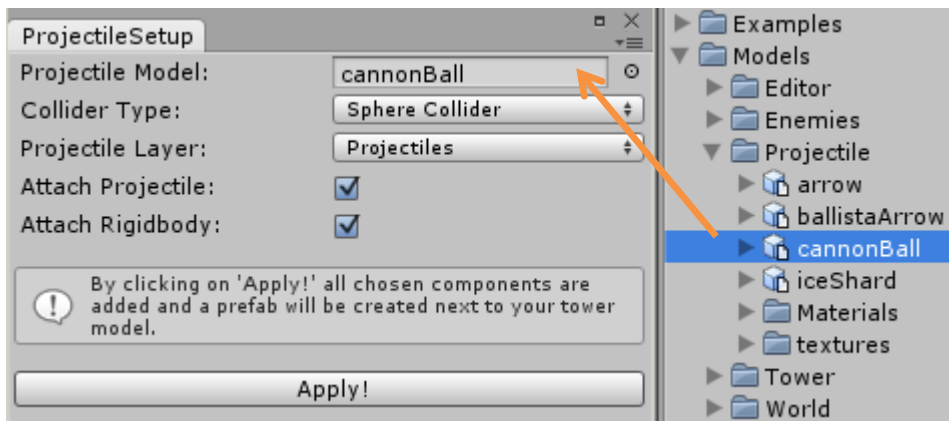
Script Connections: Audio Manager, Pool Manager (for enemy properties too)



Creating a projectile prefab follows nearly the same process as creating a tower. It's even simpler than that.

You can find an editor setup widget under Window > TD Starter Kit > Projectile Setup

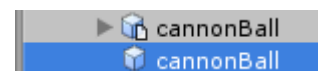
For a projectile, there are 3 components and 1 layer selection needed. A collider with “Is Trigger” unchecked, for colliding with the environment and enemies. A rigidbody with “Is Kinematic” checked and “Use Gravity” unchecked to assist this behavior. Most importantly, the “Projectile” script, which contains the projectile logic. Lastly, the projectile has to be on the layer “Projectiles”.



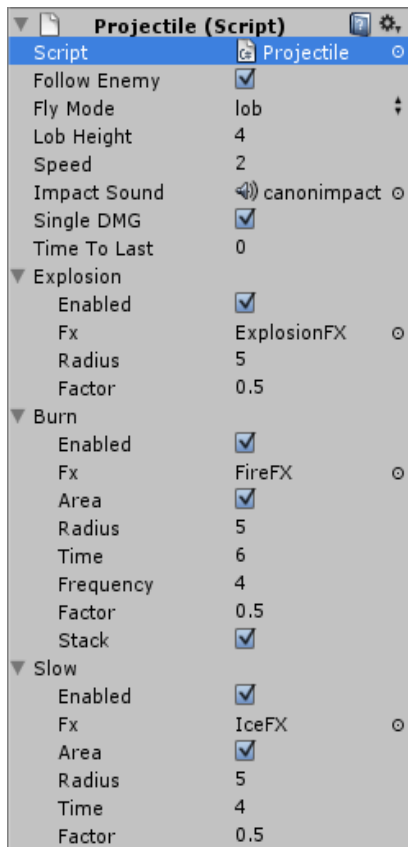
Create a new projectile by dragging a projectile model into the first slot.

That's all it takes!

We leave all other settings as they are. Click on “Apply!” to create a prefab out of your model. All components of the prefab should already have the correct settings. However, maybe you want to adjust the collider.



I just checked all of these options in the Projectile script for demonstration purposes. If you would create a projectile like that, it would destroy everything, but eat up your performance simultaneously. Here's a list of what these settings do:



Follow Enemy: whether this projectile should follow the enemy in a homing missile style

Fly Mode: type of trajectory, lob or straight

Lob Height: max height in Fly Mode lob

Speed: movement speed of this projectile

Impact Sound: sound to play on impact

Single DMG: whether it should deal damage to the enemy it hit, uncheck for area of effect damage only

Time To Last: Seconds before despawning the projectile, it gets attached to the enemy for this time

There are three different types of projectiles selectable:

Explosion, Burn and Slow. You can combine multiple types for a unique impact behavior.

Explosion: option for area of effect damage

- Enabled: whether this option should be enabled
- Fx: area of damage effect to instantiate at impact
- Radius: range in which enemies are affected
- Factor: percentage value for dealing damage based on the tower's damage

(example: Tower damage = 4, Factor = 0.5 ---> deals 2 damage to all enemies in range)

Burn: option for damage over time

- Enabled: whether this option should be enabled
- Fx: damage over time effect to instantiate at impact. This effect will be attached to your enemies over the full duration!
- Area: whether this option should affect an area
- Radius: range in which enemies are affected (if "Area" was checked)
- Time: duration of the damage
- Frequency: how often damage is dealt to the enemy over the duration
- Factor: percentage value for dealing damage based on the tower's damage
- Stack: whether or not this option should stack for multiple damage

(example: Tower damage = 4, Factor = 0.5, Frequency = 4, Time = 2  
---> deals 0.5 damage every 0.5 seconds, 2 damage in total over 2 seconds)

Slow: option for slowing enemies down

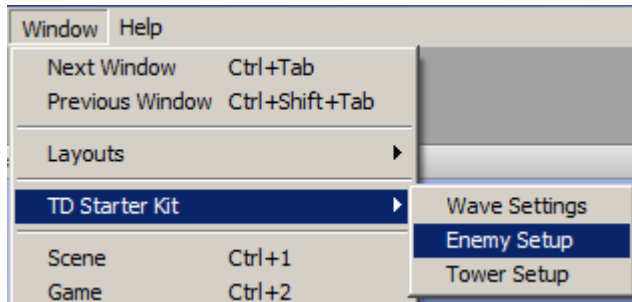
- Enabled: whether this option should be enabled
- Fx: slow effect to instantiate at impact
- Area: whether this option should affect an area
- Radius: range in which enemies are affected (if “Area” was checked)
- Time: duration of the slow effect
- Factor: percentage value for slowing enemies down based on their speed

(example: Enemy speed = 10, Factor = 0.8 ---> slows enemies down to speed value 8.

This option does not deal damage!)

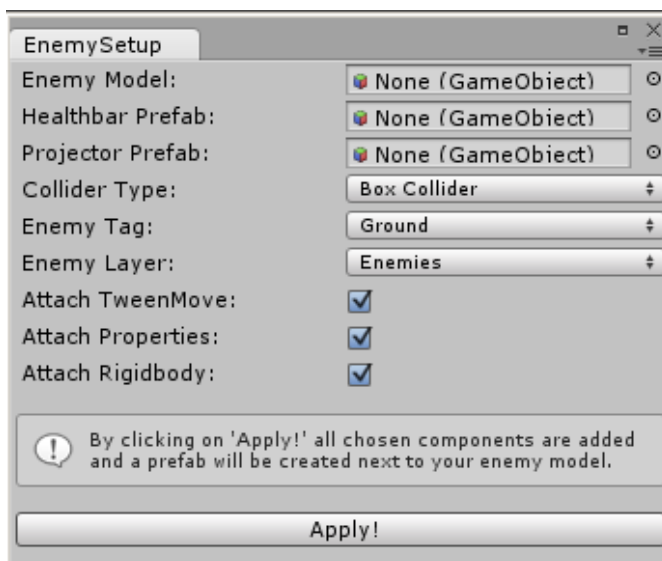
Once you set up everything, meaning your towers and projectiles, do not forget adding them to the Tower Manager, while projectiles and their effects belong in the Pool Manager.

## 6. Setting up Enemies



Like towers, Enemies consist of multiple components and settings regarding colliders and layers. There's an editor widget that simplifies the process of creating an enemy prefab.

You can find it under Window > TD Starter Kit > Enemy Setup.



Enemy Model: your model (fbx, obj, etc.) to use as an enemy prefab

Healthbar Prefab: prefab that indicates the health of an enemy

Projector Prefab: projects a shadow texture below the enemy

Collider type: type of collider which should be added to this enemy, the collider collides with projectiles

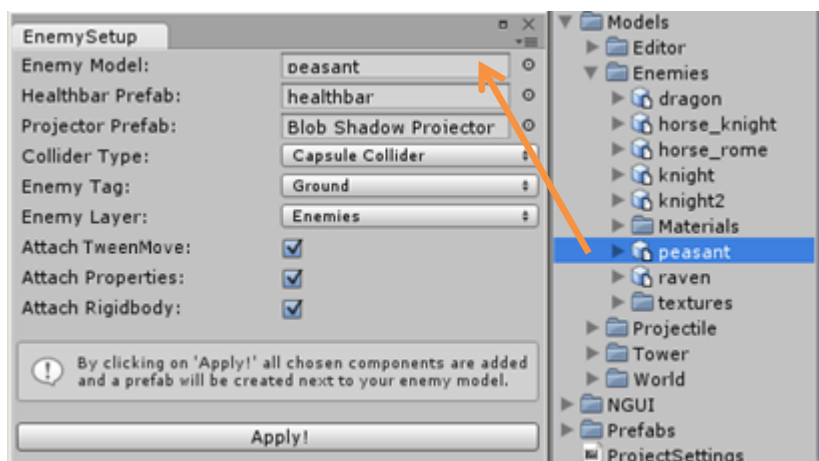
Enemy Tag: Enemy type Ground/Air

Enemy Layer: layer indicating that this prefab will be an enemy. "Enemies" by default

Attach TweenMove: whether the movement script should be attached to the enemy

Attach Properties: whether the properties script should be attached to the enemy

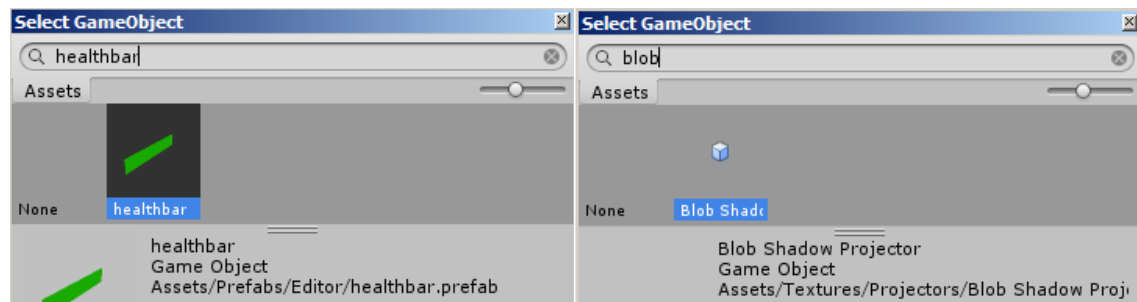
Attach Rigidbody: whether a rigidbody component should be attached to the enemy



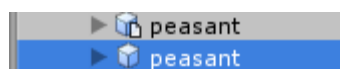
Now, we start creating a completely new enemy as a sample.

Drag & drop the peasant model located under Models > Enemies in the "Enemy Model" slot.

Assign the healthbar and the shadow projector to the corresponding fields.



As collider type, we select a capsule collider since that should match our peasant model and leave the other settings as they are. The small info text above the button implies that we are going to create a prefab. Click “Apply!”.

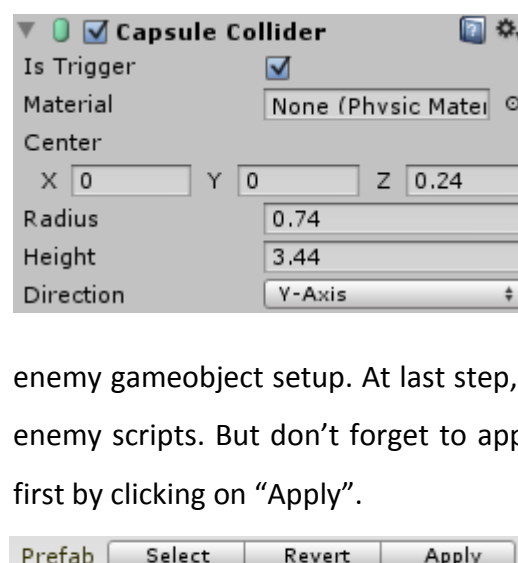
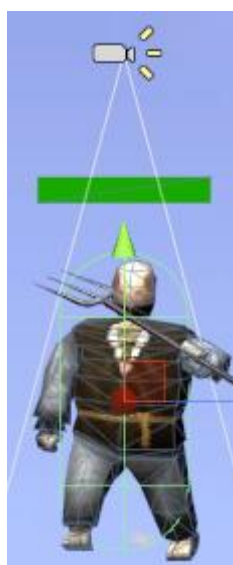


Now a prefab has been created next to our enemy model. Drag & drop it into the scene to make further adjustments.



First, we have to reposition the blob shadow projector and the healthbar that were attached to the model with Unity’s built in 3D handles, so they are correctly located above the model.

The position of the main gameobject determines the target for your projectiles. For example if we want that projectiles should shoot in the middle of the model, we have to reposition all parented objects down-/upwards. See the image below, where the pivot point is centered.

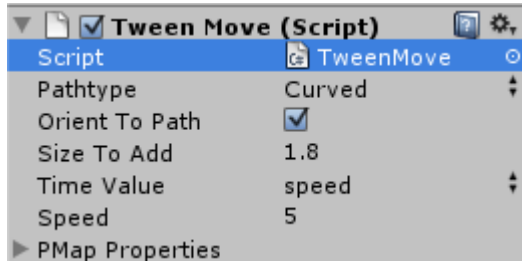


Next, we have to resize and reposition the capsule collider, so it encloses the most of the model.

This completes the basic enemy gameobject setup. At last step, you would have to modify the enemy scripts. But don’t forget to apply your changes to the prefab first by clicking on “Apply”.

## 6.1.Movement

Script Connections: HOTween, GUI Progress Map



Pathtype: curved or linear movement

Orient To Path: whether this object should orient to its direction of travel

Size To Add: additional height to the enemy so it actually walks “on” the path

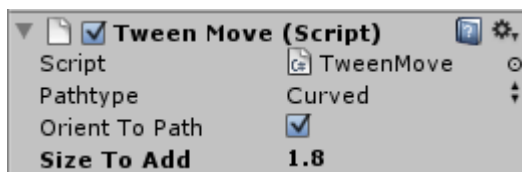
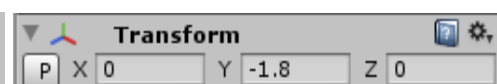
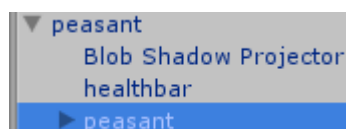
Speed: speed between waypoints or time for the whole path, based on “Time Value”

Time Value: whether this option should move based on speed or a time value

PMap Properties: whether this enemy should use the progress map, see chapter “GUI”.



At the last page, we repositioned our enemy peasant model so our projectiles will shoot at its center. Since this movement script gets attached to the main gameobject, it will also move the model based on its center. With no additional height, the result looks like the image to the left and is definitely not what we desire. Check your peasant model for your modified height value.



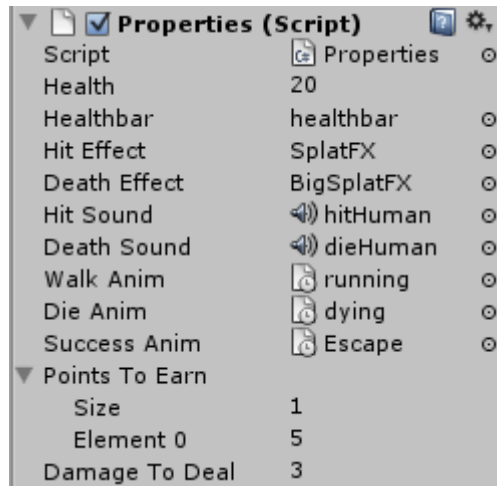
Set the additional height, “Size To Add”, to the inverted value. Now your object should correctly walk on the path.

Besides the actual movement, this script does also handle the visibility on the progress map based on the current path progress. It calculates the progress 4 times a second, what could be a little performance hungry with hundreds of enemies. Also, the method for slowing this enemy down through a projectile goes here. Please refer to “TweenMove.cs” for a more detailed insight on the implementation.



## 6.2. Properties

Script Connections: Pool Manager, Game Manager, Audio Manager, TweenMove



This script is responsible for reacting to projectile impacts, entering the range of towers, adding points to our resources or dealing damage to our health points.

Health: total health of this enemy object

Healthbar: healthbar prefab, that gets adjusted on every hit based on the health points

Hit Effect: effect to instantiate if a projectile hits this enemy (once every 2 seconds)

Death Effect: effect to instantiate at death

Hit Sound: sound to play if a projectile hits this enemy

Death Sound: sound to play at death

Walk Anim: walk animation to play at start

Die Anim: die animation to play at death

Success Anim: animation to play at path end

Points To Earn: points to add to each resources if this enemy dies through a projectile

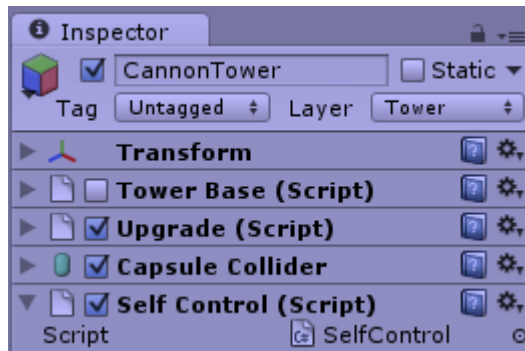
Damage To Deal: substract user's health points if the enemy reached the last waypoint

The "Properties" script makes sure to maintain the enemy's health points and constantly rotates the 3D healthbar to the camera. Furthermore, it maintains a list of all towers in range. This is important, because if the enemy dies it should signalize its death to all towers. All possible types of damage methods called by projectiles are integrated here such as Hit() (normal hit), Slow() (forwards the call to TweenMove) and DamageOverTime(). It also adds a reference of itself to the Pool Manager, so projectiles can easily access enemy variables through a dictionary and do not need to get the component every time.

After tweaking these settings in the inspector, you can now drag the prefab in a slot in the Wave Settings editor window and let it spawn! Additionally, do not forget to add the prefab and its effects to the Pool Manager.

## 7. Interactive Tower Control

---



The Tower Control script does not have public variables, because of its self-initialization. It gets attached to the tower by the “GUIImpl” script, if a tower was selected for controlling.

Its “Initialize()” method needs several components of the “GUIImpl” script. In this method it initializes all necessary components, such as a reference to the “TowerBase” script, which sets all needed tower properties (current level, radius, delay, etc.), disables the tower rotation and camera control logic, as well as setting a reference to the crosshair and aiming indicator line renderer.


On mobile devices, it centers the crosshair to the middle of the screen, so users can rotate the tower with the joystick (follows on the next page) and reposition the crosshair accordingly, while in the desktop version users are able to aim with the mouse. In the “repositioning”-part, it also makes sure that you can only aim towards valid targets, meaning e.g. the crosshair stays on the ground for towers of the type “Ground”-only.

For shooting projectiles, it accesses the “TowerBase” component and calls its custom “SelfAttack()” method with the current position of the crosshair. The GUI implementation then handles the reloading bar/texture on the screen.

Lastly, if the user presses the “X” GUI button to leave the tower, the “SelfControl” script terminates through the method “Terminate()”, which removes itself from the tower and resets all modified values such as camera control, tower script state or crosshair position. As normal, please take a look at the heavily documented code of “SelfControl.cs”.

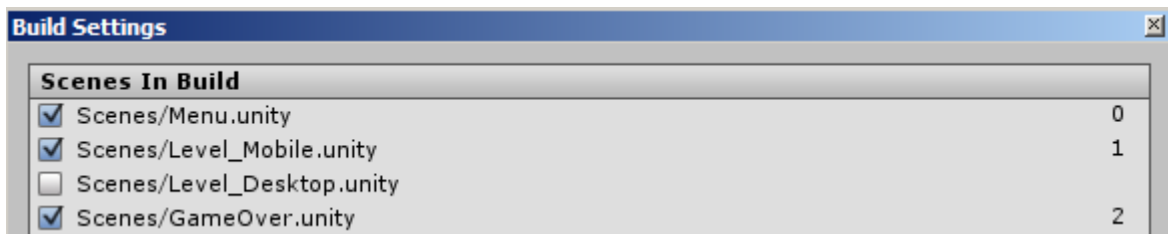
## 8. Mobile Setup

---

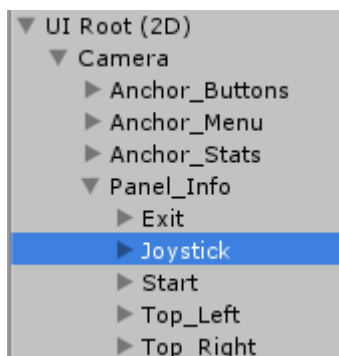
 Please open the mobile scene, “Level\_Mobile”.

This scene is obviously smaller than the desktop version. You do want to keep the level size small on mobile devices, since that’s the most important performance hit of your game. Also, you have to make sure that you have a reasonable amount of enemies, towers and projectiles including their particle effects in the scene.

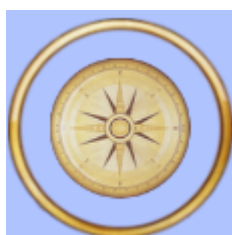
To test the mobile scene of this Starter Kit, simply switch the platform in the build settings to your device’s one and set the level selection accordingly:



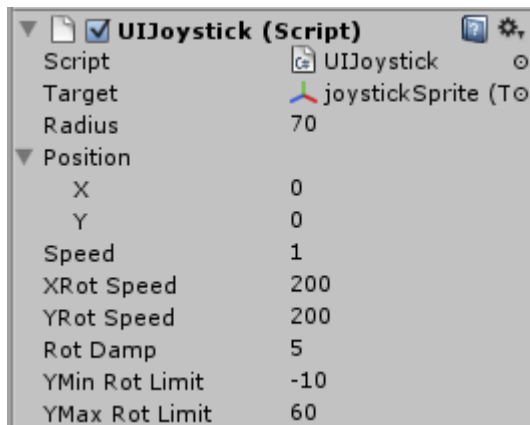
Now you can deploy it to your device!



As for the scene, there is one slight change regarding the camera control. The camera does not have an additional script attached anymore since that didn’t support mobile devices, but there’s a new component in our NGUI arsenal: A joystick controller.



Now this component controls all interactions with the camera in the game.



Target: sprite to animate while dragging the joystick around

Radius: max range for the sprite from its center

Position: indicates joystick coordinates at runtime

Speed: camera movement speed, affected by the distance of the sprite to its center

XRot Speed: rotation speed while rotating on the x-axis, while controlling a tower

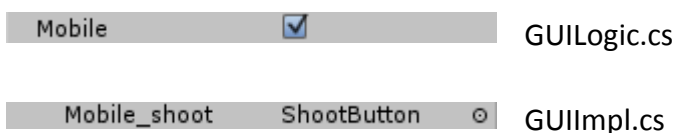
YRot Speed: rotation speed while rotating on the y-axis, while controlling a tower

Rot Damp: rotation damping factor.

YMin Rot Limit: lower rotation angle limit while rotating on the y-axis.

YMax Rot Limit: upper rotation angle limit while rotating on the y-axis.

Since the interactive tower control can't be controlled with the mouse anymore, our GUI script has another button used to shoot with towers. To enable this functionality and switch to a mobile-friendly control-scheme, the checkbox "Mobile" was toggled. Now we can enter towers by tapping on them twice.



GUILogic.cs

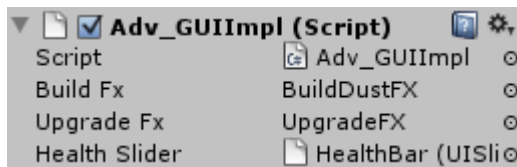
GUIImpl.cs

## 9. Advanced Example Scene

---

The example scene ‘Advanced’ serves as placeholder for more complicated mechanics and will grow in its complexity over time. Currently there are these features integrated:

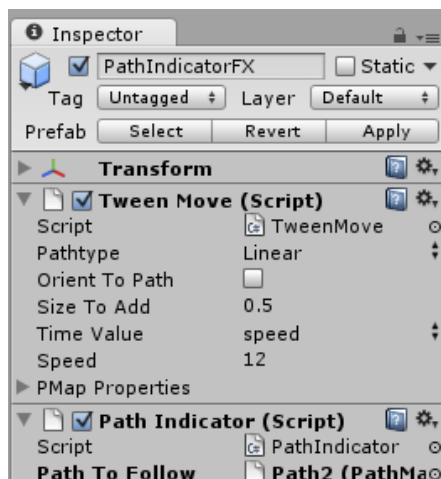
- Player health bar
- Animated reloading texture in self-control mode
- Multiple resources
- PathIndicator
- Fast-Forward button
- Laser tower



Adv\_GUIImpl.cs is another implementation of the GUILogic script, modified to include an upgrade effect and exclude all mobile related

code. You can find the full list of modifications compared to GUIImpl.cs in the first lines of code.

Let's begin with the **health bar**, that's really simple. Basically it's just a UISlider connected to GameHandler's health values, since our GameHandler stores the player's health. No further scripts required, only one line of code. See method SetHealthbar() in Adv\_GUIImpl.cs.



A **PathIndicator** is very easy to set up. It utilizes a standard iMove script for movement, so as usual, you can specify its speed on the path. Progress Map properties don't make sense in this case.

The new script "PathIndicator.cs" only takes a PathManager component for following its path and handles the emission of particles internally.



The **animated reloading texture** is a bit more entangled, but the public inspector variables should smooth things out a bit.

Control	
Crosshair	Adv_Crosshair
Aim Indicator	Adv_aimRenderer
Tower Height	8
Sprite	Sprite_Reload (10
Prefix	Reload_
Frames	14

Sprite: UISprite used for animating the texture

Prefix: naming convention for all sprites. In this case, all textures are named something like “Reload\_00”, “Reload\_01”, up to “Reload\_13”

Frames: total amount of frames used for one cycle of the texture animation

For a coding overview, please refer to the method named “DrawReload()” in Adv\_GUIImpl.cs, which is well documented.

**Multiple resources** do not require any code changes, only the resizing of all resource arrays. These are found in:

- GameHandler.cs (for display player’s resources during the game)
- Upgrade.cs (for subtracting each resource on purchase/upgrade)
- Properties.cs (for adding each resource when an enemy died)

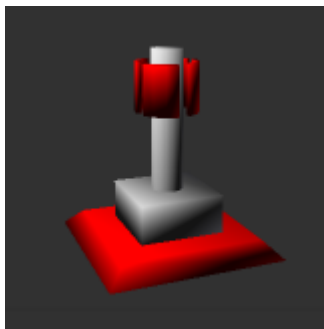
Game Handler (Script)	Upgrade (Script)	Properties (Script)
Script Max Game Health 100	Script Options Size 4	Script Points To Earn
Start Resources	Element 0	Size 3
Size 3	Cost	Element 0 0
Element 0 500	Size 3	Element 1 10
Element 1 350	Element 0 30	Element 2 5
Element 2 200	Element 1 50	
Next Scene	Element 2 20	
Game Over Scene Advanced		

Lastly, each resource needs a separate UILabel in the GUI implementation. Again, these are visualized as resources on the screen, on purchase/upgrade or when selling the tower. If you’ve defined three different resources, the size of every array has to be three!

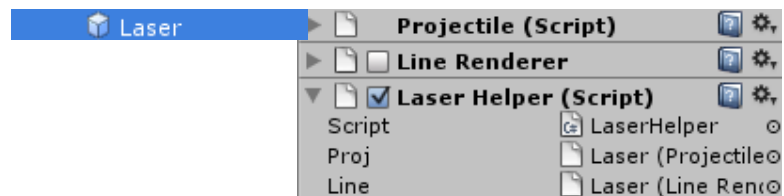
Game Info (Script)	Adv_GUIImpl (Script)
Lbl_resources	Labels
Size 3	Price
Element 0 Label_Res_Wood	Size 3
Element 1 Label_Res_Iron	Element 0 Label_Price_Wood
Element 2 Label_Res_Food	Element 1 Label_Price_Iron
	Element 2 Label_Price_Food
	Sell Price
	Size 3
	Element 0 Label_SellPrice_
	Element 1 Label_SellPrice_
	Element 2 Label_SellPrice_



The **fast-forward button** is a simple addition that you'll find in other tower defense games as well. It will speed up the game when pressed, or resume in normal speed when pressed twice. We added another reference for this button in `Adv_GUIImpl.cs` named "button\_speed". Using NGUI's delegates, it will execute the method "ChangeGameSpeed" thus setting the game's `Time.timeScale` value depending on the current value. Additionally, the method searches for the NGUI button texture and switches it to either indicate normal or double speed.



The **laser tower** is another tower demonstrating flexible projectile settings. It uses a new projectile named "Laser", which holds the default projectile script and additionally a line renderer component and the "Laser Helper" script.



The projectile is set up with a straight fly mode and high speed, so that it hits the enemy very fast to achieve something near a laser like behavior. As a laser line we do use a line renderer which gets positioned by the script. `LaserHelper.cs` needs a reference to the projectile script for knowing the actual enemy position, also a reference to the line renderer to access its starting and ending point. The script sets these positions in `Update()`. Please open `LaserHelper.cs` for more information.

## 10. Where to go from here

---

You should now be able to create your own towers, enemies and projectiles.

Additionally, you learned how to utilize all the different manager objects and how to adjust them to modify the behavior of the game.

So, the next step would be to create your own environment and to build your own tower defense game!

You likely want to rewrite or reorganize the GUI mechanics at the same time, since this is the major point when it comes to own modifications. Please do note that a starter kit GUI can't fit all game scenarios. If you have any feature requests, let me know!

From my side, there will be several updates which try to address important issues of this kit, based on user feedback and progress on the Unity game engine itself.



## 11. Contact

---

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

If you have any questions, comments, suggestions or find errors in this documentation, do not hesitate to contact us.

Visit our support thread on the Unity forums here:

<http://forum.unity3d.com/threads/130124-3D-Tower-Defense-Starter-Kit>

or send us an email at:

[info@rebound-games.com](mailto:info@rebound-games.com)

**If you've bought this asset on the Unity Asset Store, please write a short review so other users can form an opinion!**

**Again, thanks for your support,  
and good luck with your games!**

**Rebound Games**

[www.rebound-games.com](http://www.rebound-games.com)

## 12. Version History

---

### V1.0

- Initial Release

### V1.01

- **Fixes & Changes**

- Project: additional readme file, read this before you try to play the game
- EndMenu.cs: correctly destroys the Game Manager when returning to menu

- **Features**

- Game Manager: Next scene field added, allowing multiple scenes/levels

### V1.02

- **Fixes & Changes**

- WaveEditor.cs: now always saves its values when being edited by the user
- GridEditor.cs: renames new grids according to current amount of grids
- TowerBase.cs: multiple projectiles are correctly lined up at the same height when fired by players
- RangeTrigger.cs: skips adding dead enemies to floating towers, which resulted in towers sometimes shooting respawned enemies out of range

### V1.1

- **Fixes & Changes**

- GameInfo.cs: 'Start Wave' button was shown when the game was lost, fixed
- WaveEditor.cs: wave properties such as wave amount do now support undo
- TowerBase.cs: separated method for automatic and self-instantiated projectiles into two overloading methods
- SV.cs: variable 'showTooltip' renamed to 'showUpgrade'
- SV.cs, MainMenu.cs: point to readme if layers are missing
- ResetTrail.cs: new script attached to particles with trail renderers for trying to avoid artifacts when reusing them via the PoolManager. Works better than before, however this still seems like a unity bug

- Project: obsolete (Unity-)GUI removed, no GUI() calls anymore
- GUILogic.cs: major overhaul for independence. Now only serves as base for your own GUI implementation
- GUILogic.cs: new method DisplayError(string text) displays a text on the screen to inform the player, e.g. when it's not possible to buy a tower
- GUIImpl.cs: implements methods of GUILogic.cs with customized behavior
- GUIImpl.cs: even without resources, the tower tooltip menu gets shown
- GUIImpl.cs: click anywhere in the game to hide the tooltip and upgrade menu
- TowerManager.cs: added a list of 'TowerBase' scripts for tooltip access without tower instantiation
- SelfControl.cs: moved all GUI related reloading texture code into GUIImpl.cs  
SelfControl.cs: changed method Initialize() to not depend on GUILogic.cs
- ProgressMap.cs: draws the progress of enemies on their path via NGUI.  
UnityGUI related code removed. Stores a list of active 'ProgressMapObject' instances. Uses the PoolManager
- iMove.cs: progress map related code moved into inner class 'ProgMapProps'
- Project: clean-ups regarding references in other scripts as well

### ➤ Features

- WaveEditor.cs: new: 'Insert Wave' button between waves
- ProgressMapObject.cs: new script to modify properties of objects which use the progress map. It's now possible to set up objects with different icons on the progress map using prefabs
- GameHandler.cs: resources can be defined as an array, allowing multiple resources in the game. Properties.cs (pointsToEarn), Upgrade.cs (cost) and GameInfo.cs (labels) changed accordingly
- Example scenes: 'Advanced' scene added:
  - Adv\_GUIImpl.cs demonstrates a transition from medieval to a science fiction themed GUI
  - showcases multiple resources
  - healthbar for player's game health
  - path indicator (PathIndicator.cs) makes use of iMove to indicate parts of the whole path

- added two sample tower models, one projectile model (rocket), grids, three particle effects (PortalLinesFx, GoalRingFX, UpgradeFX) and a few more. See the folder 'Prefabs' next to the scene

## V1.2

### ➤ Fixes & Changes

- TowerEditor.cs: fixed serialization issue when changing tower settings
- TowerRotation.cs: auto disables on awake, prints debug message if a turret was set but script is missing
- SV\_obsolete.cs: removed since totally obsolete
- Projectile.cs: added buffer value when leaving tower radius, fixing missed hits when timescale is greater than 1
- Various checks for prefab null references before instantiating them
- GUIImpl.cs, Adv\_GUIImpl.cs: Showing the exit menu now stops the game
- Unity4 compatibility through updated scripts in separate zip file: see README

### ➤ Features

- PoolManager.cs: added method "DeactivateAllInstances" of a specific pool
- Advanced scene (see WebPlayer):
  - Fast-forward button added
  - Laser tower added (new tower model, projectile + additional script)
- iTween switched to HOTween:
  - HOTween now supports linear & curved paths - thanks to the author for a generous cooperation! –
  - Renamed iMove to TweenMove, fixed references in Properties.cs, PathIndicator.cs, WaveManager.cs, EnemySetup.cs and enemy prefabs
  - TweenMove.cs: removed support for easetypes
  - PathManager/PathEditor: added option to draw curved path gizmos
  - Restructured ProgressMap with HOTween to require less calculations

## V1.2 a – c

### ➤ Fixes & Changes

- Minor fixes. The ProgressMap Pool is now attached to its starting point object instead of the PoolManager, ensuring NGUI compatibility with version 2.2.0+.

## V1.3

### ➤ Fixes & Changes

- GUIImpl.cs: when placing new towers onto grids, their turret rotation is now fitting the grid rotation, allowing them to face the path
- Properties.cs: added randomized walk animation offset on spawn
- updated Unity4 script files “Unity4\_Files.zip” accordingly

### ➤ Features

- WaveManager.cs: added options to specify the start delay and delay between enemies as minimum and maximum values
- Properties.cs: success animation slot added, which allows the playback of an animation when the enemy reached the end of the path
- GUILogic.cs, Scene: when pressing ESC during the game, there will be a new slider for manipulating the sound volume
- Upgrade.cs: possibility to swap out the tower model per upgrade level. So when upgrading a tower, you can illustrate the change with a new model