80s Shoot'em Up Kit

by ETI

v 1.4

Change Log

v 1.4 : Added complete *C Sharp* version of the project

v 1.2 : Added on-screen button controls option and grid helper utility.

v 1.1 : Adjusted scale of some colliders to avoid a compatibility issue with Unity 4.5x 2d physics

v 1.0 : first release

Requirements

This asset requires Unity 3D (Free or Pro) 4.3.4f1 or later version

Overview

General Description

'80s Shoot'em up Kit' is a Unity complete project package aimed to give you hints on creation of your own **Unity native 2D** shooter or similar game.

The project was designed to be cross-platform, and optimized to run smoothly on mobile devices.

You will find the basic elements you can expect from a classic 8-bits arcade shooter and a complete implementation of the various input methods.

Key Features

- Firing system : shoot delay, bullet count limiter, Auto Fire and fire rate penalty - fire rate penalty is applied on (regular) Fire button kept down (encourages intensive button press)
- Object Pooling example : optimize the application by reusing elements rather than instantiate and destroy them
- Event system : set checkpoint, play or stop an audio clip, spawn an item, change level...
- Keyboard, joystick, mouse, and touch screen, non interfering inputs
- Run the project in a unique scene or handle multiple levels

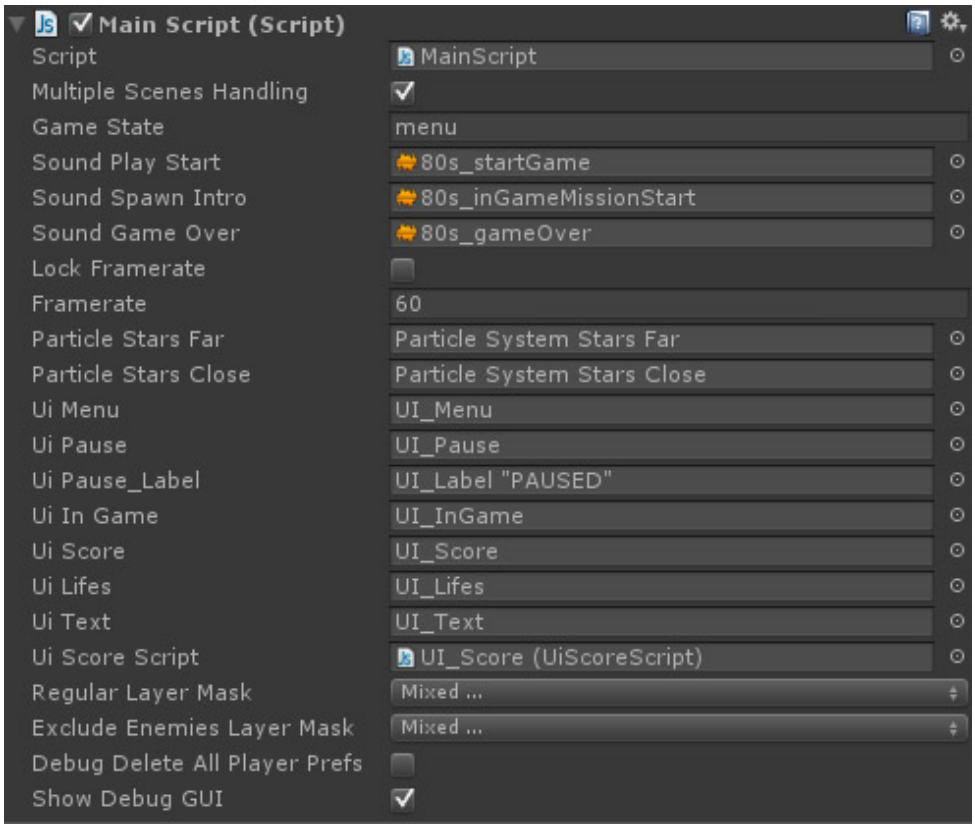
Use cases

- Help beginner developer to learn how to create his own 2d game, or have a working template to build upon.
- Give intermediate developer another point of view on workflow, tips, and optimization
- Discover technics not only suitable for an horizontal shooter game but applicable to various kind of Unity 2D projects.

System Overview

Main script

MainScript controls the main functions of the game. It receives and manages the different game and UI events. This script is attached to **Main Camera**.



Main Camera

All elements relatives to the camera position like UI, player, background, are placed inside the **Main Camera** gameObject. The Main script itself is a component of the camera.

Main Camera 2d box collider component

On the camera itself you will find a box collider that serves two purposes :

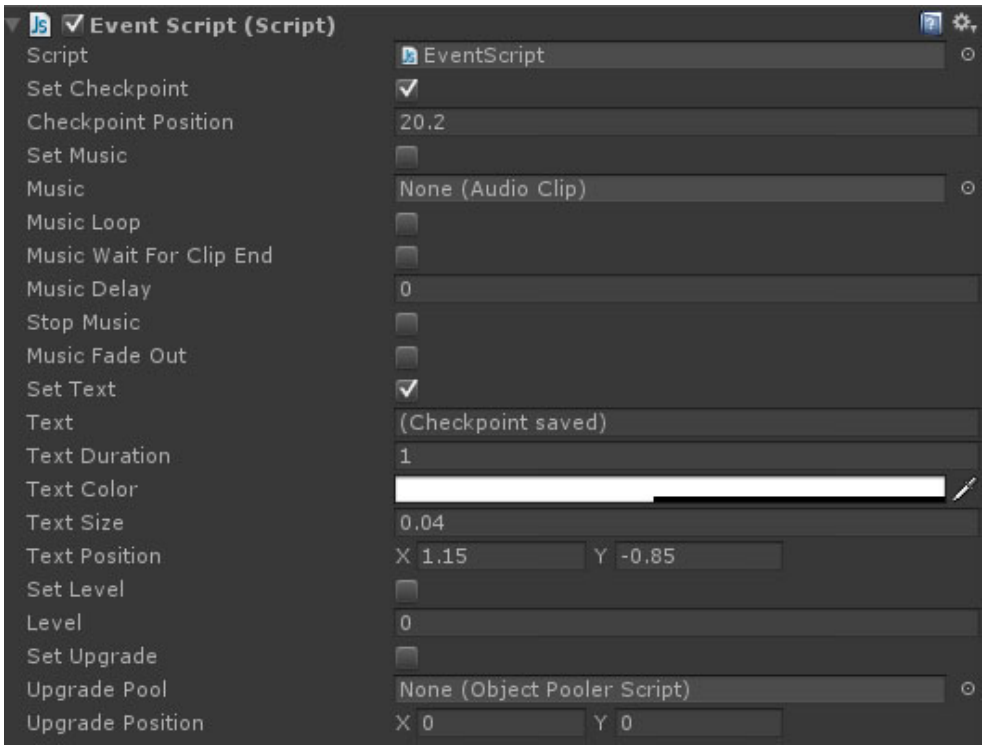
- Detect the enemies 'in view' when player spawns (or when collecting special **blast** item) to destroy them
- Collide with events objects to activate them

Collision check is performed in **MainScript**.

Event system

Event system is used to fire events based on camera position. When an event is in camera sight (in fact enters in **Main Camera**'s 2d box collider) it will be activated.

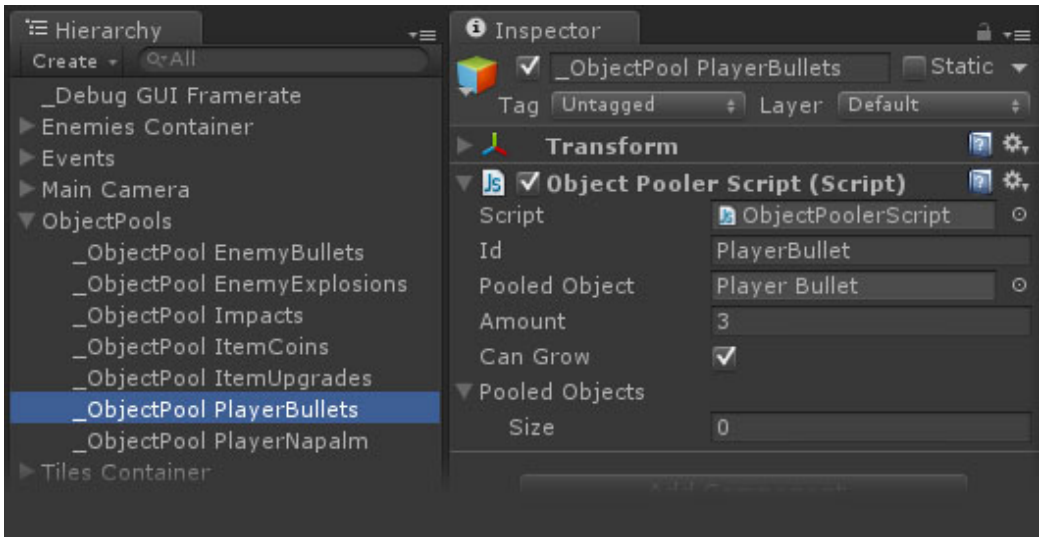
Events can be configured to set a checkpoint, start or stop an audio clip, display text (using **UI_Text** gameObject), set a level, spawn an upgrade.



You will find various parameters for each option, and you can use several options together on the same event.

Pooling System

Pooling object technic is invaluable for memory access optimization and for this project it is mainly used to manage bullets, impacts and items.
The pooling system is managed by the script **ObjectPoolerScript**.



Instead of instantiating and destroying an object with **Instantiate ()** and **Destroy ()** :

- Use **objectPool.Spawn()** to use this object from pool
- Use **object.SetActive(false)** to deactivate it and make it available again from its pool

Player

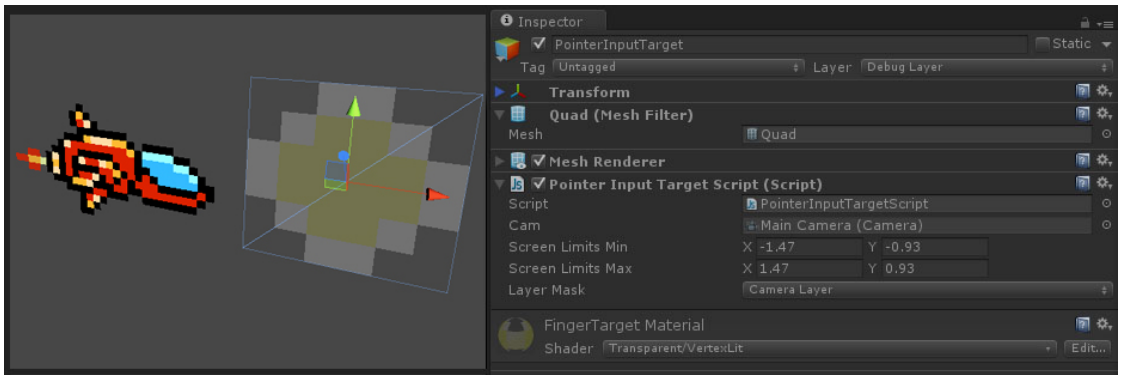
The **PlayerScript** manages user inputs, and ship's interaction with its surroundings.
It player's relative values like player's score, lifes, speed, upgrade levels ...

The mouse and touch screen inputs are managed in conjunction with **PointerInputTarget** gameObject.

PointerInputTarget

PointerInputTargetScript takes care of the pointer (mouse or touch screen) inputs.
It basically targets the position of the mouse pointer/user's finger.

If no other input method is detected, **PlayerScript** will use **PointerInputTarget** for position reference.

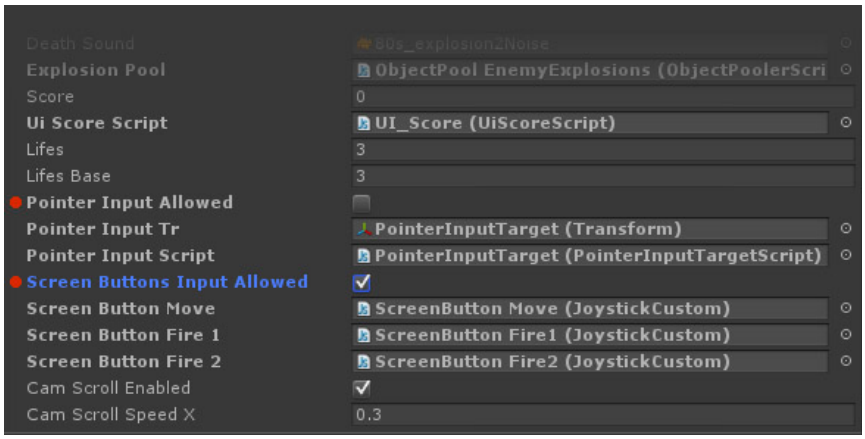


On Screen Buttons (New)

This control type offers an alternative to the regular drag touch screen control (**PointerInputTarget**). You can find it under "*Hierarchy Tab/**OnScreenButtons***".

While Pointer Input type (mouse/touchscreen input), keyboard and joystick inputs can be used at the same time, on-screen buttons inputs when used will disable other controls.

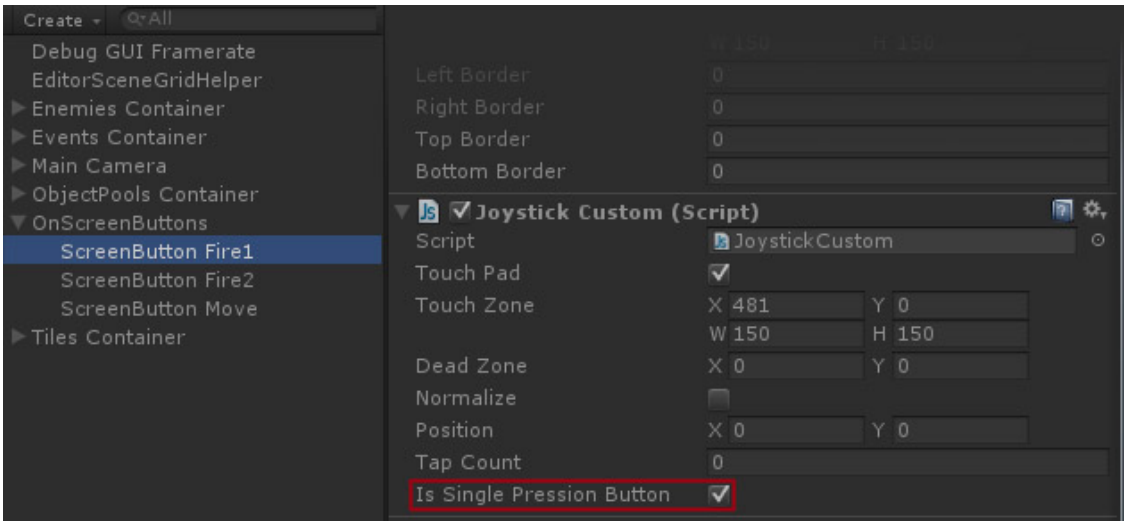
You can activate on-screen buttons control in the player script (**PlayerScript**) by checking the boolean **ScreenButtonsInputAllowed**. If checked, it will disable pointer input (**PointerInputAllowed**) at start.



The script **JoystickCustom.js** is a modified version of the Joystick.js from the “Penelope iPhone” Unity Tutorial.

The script can check if button was just pressed down at a given time, or over several frames. Use **IsFinger()** and **IsFingerDown()** as you would use **Getbutton()** / **GetbuttonDown()**.

This custom version also handle single press input. Simply enable **isSinglePressionButton** if needed.
As this option is available, it is however deactivated by default in the kit.



Upgrades

Upgrades are player items released by destroyed enemies or by events (using **Event System**).

ItemUpgradeScript works in conjunction with **PlayerScript** to determine which **type** of upgrade will be spawned (see **SetItemType()** function in **ItemUpgradeScript**).

Speed Upgrade and **Weapon Upgrade** will increase player respective upgrade levels, **Bomb Upgrade** will enable bombs. **Blast** item will destroy all enemies in sight.

System's workflow – main script working cycle

You should understand the system's workflow in order to use/expand it.
All the main game logic is processed through **MainScript** (attached to **Main Camera**).

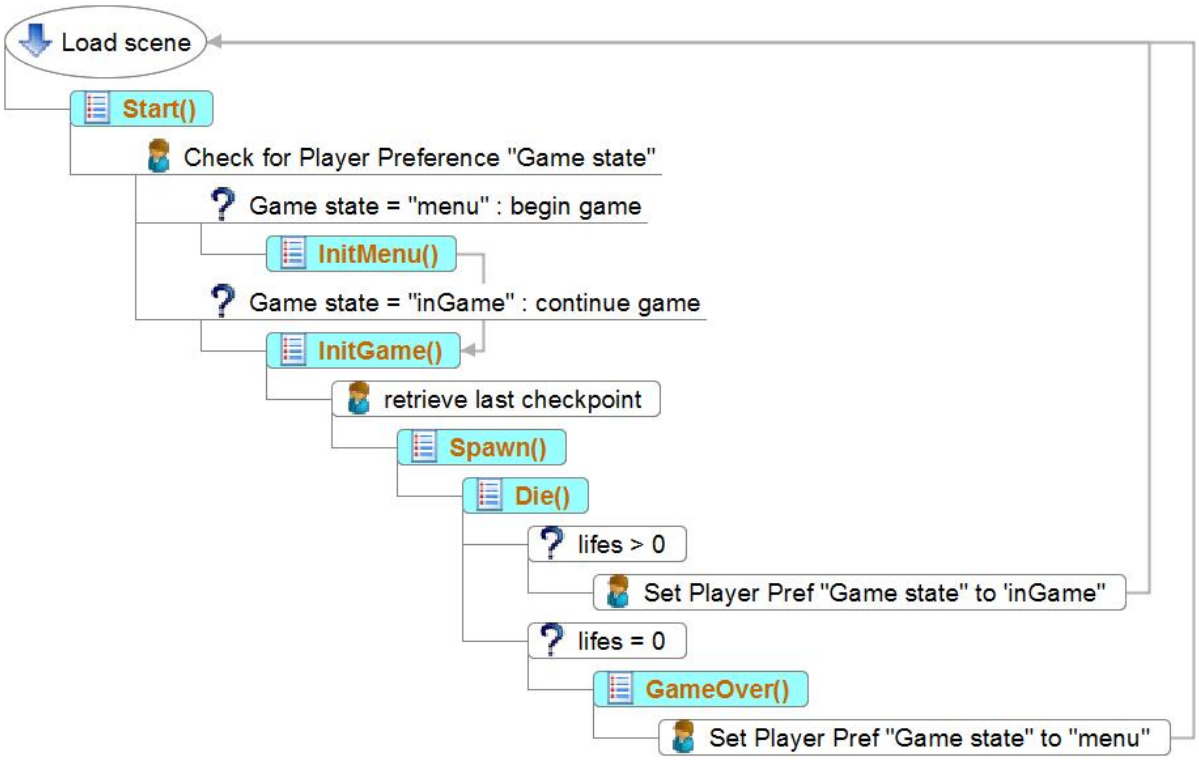
All the project runs and loops in a unique scene – *although you can load game levels additively* (see bellow). When the player die, before reloading the scene, data are stored in **Player Prefs**.

The important value, ruling the state of the game, is a string variable named **gameState**.

When the scene loads, if **gameState** is equal to **inGame**, the game session will continue according to stored data.

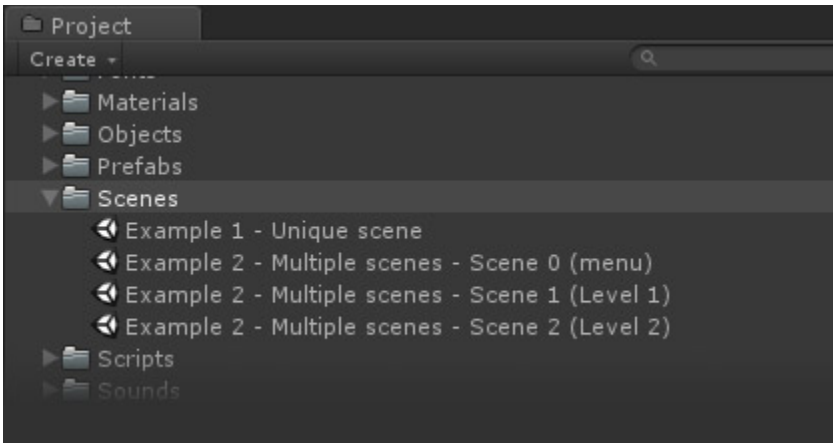
Else, **gameState** is equal to **menu** meaning that application as been reseted, and will restart at menu screen.

This diagram of the **main script's working cycle** explains the process :

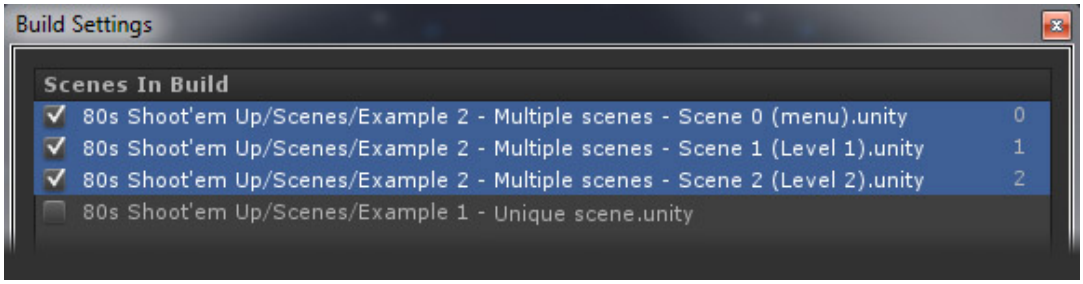


Multiple scenes handling

You can choose to encapsulate all the project in one single scene (as shown in **Example 1**), or use multiple scenes (**Example 2**). Both methods have their pros and cons.



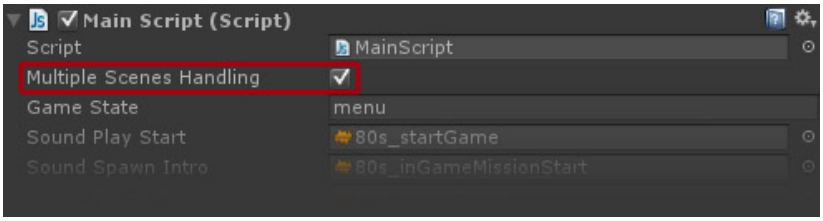
In general, for a small scale project you may prefer to use the first option.
If you use multiple scenes handling, you have to set menu to scene **0** in **Build Settings**.



Others scenes will be **loaded <u>additively</u>** to the first scene (in **InitGame()** function of **MainScript**).

Being part of core game logic, **multiple scenes handling** is managed by **MainScript**.

You must setup the variable **multipleScenesHandling** in **MainScript,** and configure the **Build Settings** appropriately.



Common methods - Enemy script case study

Most scripts in this project are using common methods and naming convention.
We will take the example of an enemy script.

For clearer reading each enemy has is own script. For a better understanding you should start by looking at the **enemy3** script which is the simplest example.

At first, an enemy is out of screen, and considered as **asleep**. As long as it is asleep no operation is performed.

We use the function **OnBecameVisible()** to wake it up. The **asleep** boolean value is set to **false** and enemy can perform its task.

As **OnBecameVisible()** and **OnBecameInvisible()** are intensively used in this project, you should note that :

- a renderer must be present (and active) on the gameObject holding the script

- when running the project in the editor, scene view cameras will also cause this function to be called

As previously, we use the function **OnBecameInvisible()** to know when an enemy is gone out of screen view, then we destroy it.

The function used to destroy an enemy or any other type of object is always named **DestroyObject()** (using pooling method or not).

```
function OnBecameVisible()
{

    // On became visible, wake up the object
    asleep = false;
    myTr.collider2D.enabled = true;

}

function OnBecameInvisible()
{

    // On became invisible, destroy object
    if (gameObject.activeInHierarchy == true) DestroyObject();

}


function DestroyObject()
{

    yield WaitForSeconds (audio.clip.length); // Wait for the end of explosion audio clip

    if (gameObject.activeInHierarchy == true) Destroy (gameObject); // Kills the game object

}


function Start ()
```

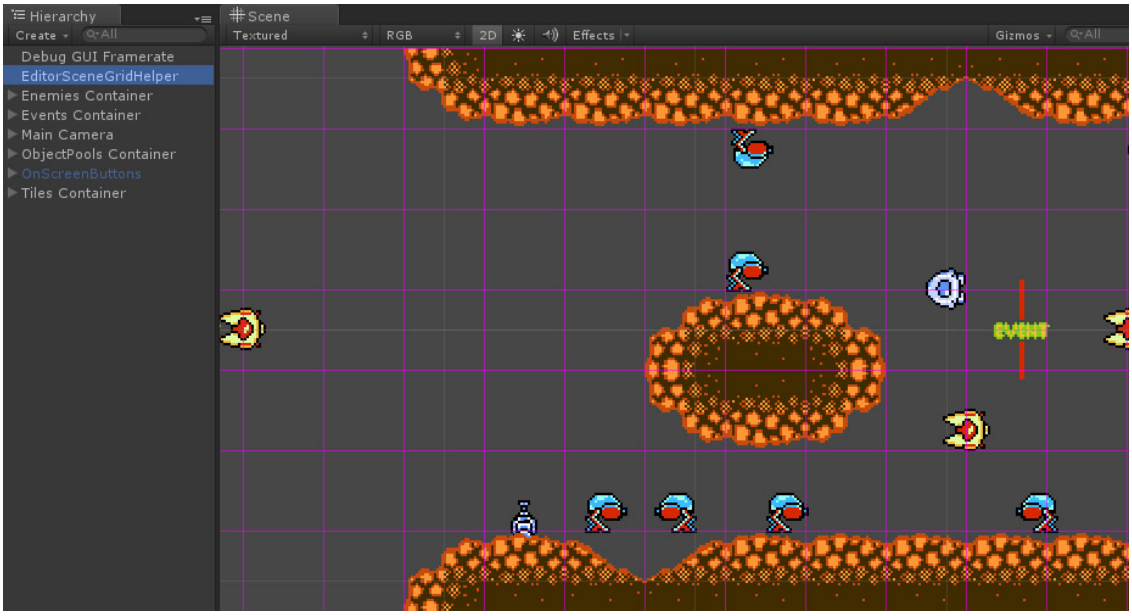
If an object is meant to be reused, we will consider it as **asleep** again and set back the value to **true**.

On some objects with particular behaviors (like the coin item) and used by pools you will find a **ready** boolean value.

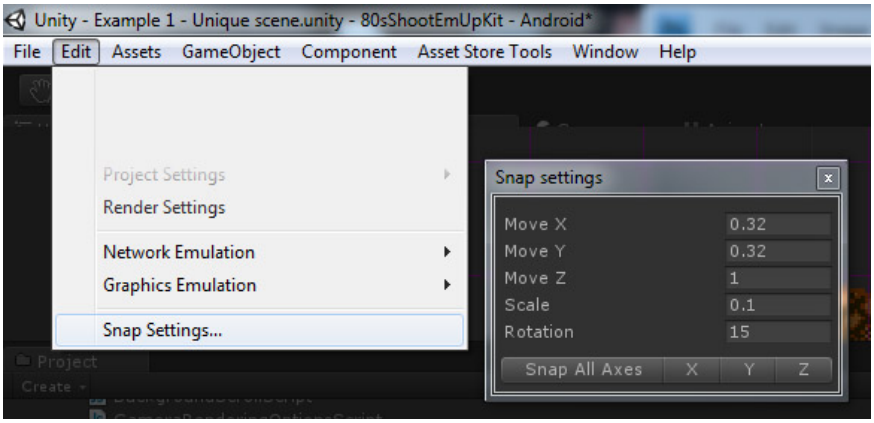
ready value is set to **true** only one time to ensure that the script has initialized, prior to perform any action.

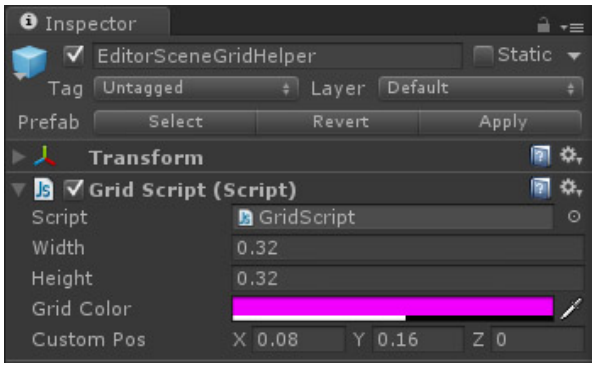
Grid Helper Utility (New)

The Grid Helper Draws a grid in editor's scene window to help tile placement.



To use in conjunction with Snap Settings ("*Edit/Snap Settings...*"), with snap values 'Move X' and 'Move Y' set to 0.32 (32x32 pixels at 100 Px/Unit).





While dragging any Gizmo Axis using the Translate Tool, hold the Control key (Command on Mac) to snap to increments defined in the Snap Settings.

You can find a **EditorSceneGridHelper** prefab under "*Prefabs/Utilities*".

Side notes, tips and tricks

Project specifications

The default project's resolution is set to 320 x 200 px at 100 px/unit.
The project resolution in units is therefore equal to 3.2 x 2 units.

Except the transparency effect and stars particles, the colors are bounds to the Nintendo Famicom/NES color palette. Color Preset Library is provided with the project.

Pixel to units value

The pixel to units value can be modified in the inspector, under texture import settings.

Scene view cameras

As already mentioned, the scene view should be hidden when you play the project in the editor, as most scripts use object visibility.

Collapsing sprite component

If you are new to the Unity native 2D tools, please note that in order to be able to move a sprite, the Sprite Renderer component has to be expanded.
Collapsing a Sprite Renderer component will lock all sprites in the scene !

Sprites batching, sprites Atlas and Sorting Layers

To optimize the batching and keep the draw calls as low as possible, there are some rules to keep in mind :

Use the minimum sprites atlases as possible. Best practice is to keep all the content in the same atlas.

Try to keep the same atlas content on the same **Sorting Layer**, and use the minimum **Sorting Layers** as possible.

For the sprites from the same atlas, rather than using **Sorting Layers**, simply organize their depth with **Order in Layer** values.

Script comments

You should find all the remaining needed informations in the scripts, where nearly each line is commented.

Support

If you have any question you can send an e-mail at support@eti-software.com

Follow ETI

ETI on facebook : http://www.facebook.com/ETISoftware
ETI on Twitter : http://twitter.com/ETISoftware