Space Rail Shooter game template

Version 1.3.2 | August 2020

© Kevin Foley 2018 – 2020
onemanescapeplan@gmail.com

Read First

- As with any plugin or template from the Asset Store, **I strongly recommend that you do not make any changes directly to the template files**. When I release updates in the future, a**ny changes that you make to files in the template may be overwritten and lost** when you download an update for this package. Here are tips on how to avoid this issue:
  - Instead of making changes to class files, you can **extend those classes** with your own child classes. Nearly all of the included classes should be fully extensible.
  - Instead of making changes directly to prefabs, meshes, textures, and other non-code assets, **create a copy of the file in another folder** and make changes to the copy.
- I have included many meshes, textures, and sound effects that I created for this template. My primary expertise is in programming, so don't expect world-class art! Anyway, **when building a game from a template, it's best to replace all of the art and sound effects**, so that your final game doesn't look too much like any other game made from the same template!

Philosophy

Why are there so many class files?

For beginning programmers, the large number of small class files in a project like this may seem overwhelming. New programmers tend to put lots of code into fewer files. Countless essays have been written on the subject, and I don't want to get into it too much here, but in brief: modern experts tend to generally agree that **putting too much code into a single class is a bad idea**. Very long classes are more difficult to read and reason about; they tend to be symptoms of issues like coupling and putting too much responsibility in one place. Additionally, **long and complex classes often become difficult to work with**.

For example, let's say that our new programmer wants to create the player. He creates a class file called "Player" and starts adding functionality to it. He adds support for handling keyboard and mouse input, moving across the screen, firing weapons, etc. He gives the class properties such as "Health", "Lives", and "Inventory". Eventually, he has a class that is thousands of lines long but covers everything that a player does.

Now the programmer wants to create the first enemy. An enemy has many of the same functions as the player, but not all of them. It moves, fires weapons, and has health, but it doesn't accept keyboard/mouse input and doesn't have an inventory or lives. Our rookie coder can't just re-use the Player class, so he probably ends up copying chunks of code from Player into a new Enemy class. Now he's got duplicate code in multiple classes, which means making the same changes in multiple places when he expands a feature or fixes a bug.

Unity is designed for you to *compose objects* by assigning multiple distinct *behaviors* (MonoBehaviour scripts), which are attached to the object as *components.* This is sometimes referred to as the component pattern. For example, say that we create a "Movement" script, a "Health" script, a "FireWeapon" script, an "Input" script, and an "AI" script. Then we could *compose* our player by assigning the Movement, Health, FireWeapon, and Input components to the player (as well as Unity's built-in components such as MeshRenderer and MeshCollider). We could compose an Enemy by assigning the Movement, Health, FireWeapon, and AI components. We could compose an obstacle, such as an asteroid, by assigning just the Health component to it.

Not only do small, reusable components make it easy to compose all kinds of objects, they're also easier to understand, debug, and upgrade. We don't need to scroll through thousands of lines of code to find the code for a specific feature – we just open the class file for that feature.

Why doesn't it have [some feature]?

Currently I freelance full-time. I have been putting this template together in my spare time. If it sells well enough to justify the time I've *already* put into it, I will certainly invest more time into additional features. I may raise the price as new features are added, but you don't have to worry about that if you've already purchased this package!

Additionally, I don't want to violate anyone else's copyrighted intellectual property. If you're wondering if I'll add some specific key feature from a real game (say, NPC wingmen who radio you with talking animal heads) the answer is probably "no", because I wouldn't want to infringe on someone else's property. You are, of course, free to add whatever features you want, but I'd suggest being mindful of intellectual property laws if you plan to release your game.

What's with the graphical style?

I just like the graphical style in early 3D games, including *Star Fox* and other Super FX chip titles for the SNES, arcade games such as *Stun Runner*, etc. Consider this the 3D equivalent of all the 16-bit-style pixel art games that still get released each year.

You'll notice that the frame ate is set to 20 fps if you select a "16-bit" preset from the **`Space Rail Shooter > Quality`** menu (see "Quality Presets"). This is because early 3D games often ran at very low frame rates due to the limited hardware power of their era. Frame rates for SuperFX games on the SNES could dip into the single digits!

Of course, you can always reskin the game and change the graphical style to suit your liking, if you don't want your game to look like it was released in the early 90's.

General Overview

Space Rail Shooter is a 3D game template for a third-person on rails shooter where the player flies a spacecraft. The template is designed with a retro style reminiscent of early 3D arcade and home console games, but can always be reskinned with modern graphics if you want to keep the gameplay style but don't like the retro look.

Within the `Assets` folder of the template, there is a `OneManEscapePlan` folder that contains two important subfolders:

- `OneManEscapePlan/Scripts`: This folder contains my general-purpose code library which could be reused for many different types of games. It includes code for a health/damage system, spawn pooling, triggers, animations, UI, and more. This package does not contain any art assets (such as meshes, textures, or sound effects) or prefabs. You are less likely to need to make any changes to files in this package.
- `OneManEscapePlan/SpaceRailShooter`: This folder contains all files that are specific to this game template. This includes scripts, meshes, textures, animations, prefabs, scenes, etc. You are more likely to need to make changes to files in this package.

It is best practice not to make any edits to original template files, because your edits could be overwritten when you download an update for the template. Instead, you should start by making a new folder where you will store all of your own assets and customizations (see "Getting Started", below). When you wish to make changes to one of the template's scripts, it's best to create your own child class that extends the script and overrides any functionality you want to change. When you wish to make changes to most other types of assets (such as scenes, animations, or textures), it's usually a good idea to make a copy of the asset and make changes to the copy. For prefabs, if you are using Unity 2018.3 or newer, you can make a variant of the original prefab.

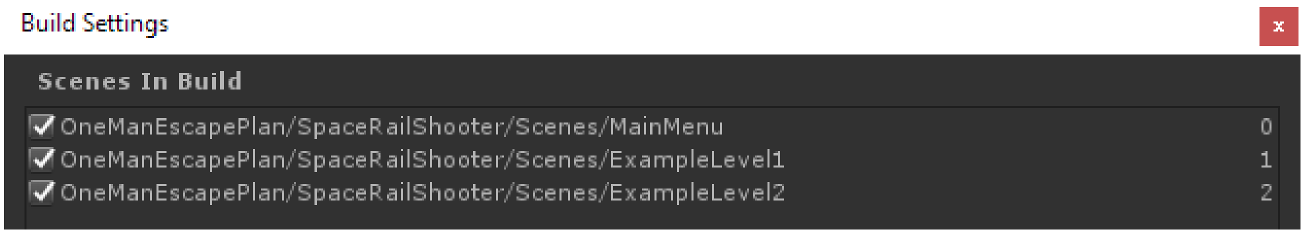
This template is not a complete "reskin-and-release" package. It currently contains one example level, plus the beginning of a second level for testing level progression and a few additional features, and a few demo scenes to illustrate certain features. I have not included additional levels because they can be time-consuming to implement and you'd probably end up completely replacing them anyway.

Since this is my first Asset Store release, your feedback is highly valuable in shaping the direction of future development. I want to make sure that the template is easy to understand

and work with. If you have any suggestions or complaints, please contact me at onemanescapeplan@gmail.com .

Getting Started

The first thing you should do is open **File > Build Settings** and make sure that the correct scenes have been added to the “Scenes in Build” list, as shown here:



```
☑ OneManEscapePlan/SpaceRailShooter/Scenes/MainMenu        0
☑ OneManEscapePlan/SpaceRailShooter/Scenes/ExampleLevel1   1
☑ OneManEscapePlan/SpaceRailShooter/Scenes/ExampleLevel2   2
```

If these scenes are missing, you will need to add them by opening each scene and clicking the “Add Open Scenes” button in Build Settings.

Before you start making any changes, you’ll probably want to test out the included features! Open up the “Main Menu” scene and hit “Play”. From here, you can change your settings, or jump in and play.

Next, you should start a new working directory for your game. Create a new folder within the “Assets” folder, and name it after your game (e.g. `Assets/SuperAstroBlaster`). If you haven’t decided on a name for your game yet, just pick something – you can always change it later.

Within your new folder, you should create subfolders that will house different types of assets (e.g. `Assets/SuperAstroBlaster/Audio, Assets/SuperAstroBlaster/Materials/, Assets/SuperAstroBlaster/Scenes, Assets/SuperAstroBlaster/Scripts, Assets/SuperAstroBlaster/Textures`). Next, I recommend copying the “ExampleLevel1”, “ExampleLevel2”, and “MainMenu” scenes from `Assets/OneManEscapePlan/SpaceRailShooter/Scenes` into your new `Scenes` folder, so that you can make changes without affecting the original files.

Adding the Unity Post-Processing Stack (optional)

Next, you may want to add the Unity Post-Processing Stack from the Asset Store. This allows the template to activate the anti-aliasing post-processing effect; you could also use it to add other post-processing effects to your game to give it a unique style. After you have installed the post-processing stack, add **POST_PROCESSING** to your **Scripting Define Symbols** so that the *Space Rail Shooter* template knows that the post-processing plugin is installed.

After adding the Post-Processing Stack, you may receive errors about the “`MinAttribute`” property. This is a compatibility issue between the Post-Processing Stack and Unity 2018.3. To fix this issue, double-click the error to open `MinDrawer.cs`, and replace each usage of `MinAttribute` with `UnityEngine.PostProcessing.MinAttribute`

If you choose not to install the Post-Processing Stack, the camera in the example levels will have a missing script. You can ignore this, or remove the missing script.

Browsing the Code

The template contains scripts in two places: `Assets/OneManEscapePlan/Scripts` and `Assets/OneManEscapePlan/SpaceRailShooter/Scripts` . With these folders, the scripts are organized into subfolders that are hopefully self-explanatory.

At the beginning of nearly every script file, there is a C**omplexity** rating and a list of applicable **Concepts**. The complexity rating ranges from “Beginner” to “Expert”. The concepts list includes topics that I consider particularly relevant to understanding the script; these may be general programming concepts, features of the C# language, features of the Unity engine, or even features of the OneManEscapePlan or SpaceRailShooter packages. For some advanced or obscure concepts, I have included links to relevant documentation or articles on the web.

Together, the Complexity and Concepts guidelines are intended to help you get a better idea of which scripts you can feel comfortable working with, based on your own skill level. However, keep in mind this is a subjective rating – depending on your personal experience, you and I may not always agree on how complex or advanced a script is, or which concepts are the most relevant.

Shaders

The *Space Rail Shooter* template comes with several retro-style shaders to emulate the look of 3D console and arcade games from the early 90s. The shaders, along with documentation and example scenes, are located in `Assets/OneManEscapePlan/Retro 3D Shaders`.

Rendering at Low Resolutions with Pixel Camera

The *Space Rail Shooter* template was specifically designed to emulate the look of early 3D console and arcade games, including support for rendering at retro resolutions (such as 256×224, a resolution common among 8-bit and 16-bit consoles). However, rendering retro games at low resolution in Unity is not quite as simple as just changing the overall game resolution.

In the Editor, Unity renders at whatever resolution you have the Game window set to. If you increase the scale of the game window using the scale slider, the image is scaled using point filtering (also called “nearest neighbor” scaling). This simply blows up the size of each pixel without changing its appearance at all. That’s exactly how we want the engine to behave when rendering low-res retro games – but it’s not how Unity behaves by default in *builds.*

In a build, if the camera is rendered at a lower resolution than the window, the image is then scaled up using bilinear filtering, which blurs the image. When scaling a very low resolution such as 256×224 to a modern monitor’s native resolution (1920×1080 or greater), bilinear filtering produces a blurry mess (think 240p videos on YouTube).

Render Textures to the Rescue

To solve the blurriness problem, we have to scale up the rendered image manually. This is done using the **PixelCamera** component, which we attach to any camera that we want to render at low resolution. PixelCamera renders the scene according to a **RenderSettings** asset.

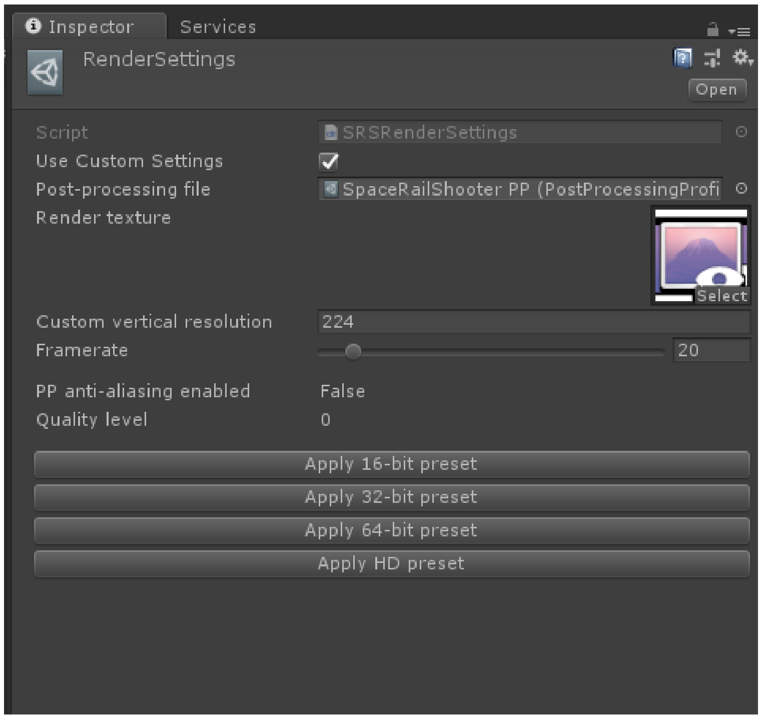
When we have enabled a custom vertical resolution (such as 224), rendering is done in a two-step process. First, the camera renders to a **RenderTexture**. Then, the rendered image is blitted into the display backbuffer, from which is is displayed directly onto the monitor. Point filtering is used for the blitting process, so that the image is scaled up without any blurring. There is a small performance cost for blitting, but this should be offset by the fact that you are resolution at a low resolution.

The upscaling process process is already built into *Space Rail Shooter*, so you don’t need to worry about how it works; you just have to configure your Render Settings and apply the PixelCamera component to your cameras. If you prefer not to render at a retro resolution, you can simply uncheck the “Use Custom Settings” box in the Render Settings used by PixelCamera, or disable/remove the PixelCamera component entirely.

The Render Settings asset

The PixelCamera component requires you to select a **RenderSettings** asset that describes how the scene will be rendered. If you wish to have different cameras render with different settings, you can create multiple RenderSettings assets. The *Space Rail Shooter* template comes with several RenderSettings assets, located in `SpaceRailShooter/Resources/Settings.` New RenderSettings assets can be created from the Create menu (`Assets > Create > Render Settings`).

The RenderSettings asset has the following fields:



Script: SRSRenderSettings
Use Custom Settings: ✓
Post-processing file: SpaceRailShooter PP (PostProcessingProfi
Render texture
Custom vertical resolution: 224
Framerate: 20
PP anti-aliasing enabled: False
Quality level: 0

- **Use Custom Settings**: If this box is checked, the other fields are shown and used. If this box is not checked, the other fields are hidden and the camera renders at the window resolution and native framerate. Uncheck this box if you do not want your game to have a pixelated retro look.
- **Post-processing file:** This optional field is where you can specify a post-processing file. Some of the presets (described below) will turn the anti-aliasing on or off in the post processing file. Note that for this to take effect in your game, you must also attach the post-processing script to your camera using the **Post Processing Behaviour** component.
- **Render texture:** This required field is where you specify the render texture which the camera will be rendered to, before the rendered image is scaled up to the window resolutions
- **Custom vertical resolution:** The vertical resolution you want your game to render at. The horizontal resolution will be dictated by the aspect ratio of the game window; for example, if the game is running in fullscreen on a 1920 x 1080 monitor, the aspect ratio is 16:9. If your vertical resolution is 224, the horizontal resolution will be `(16 / 9) * 224 = 398`
- **Framerate:** The refresh rate you want your game to render at (frames per second). While modern games typically target 60fps, 3D games in the 16-bit era could not be rendered at such a high framerate, and often were rendered at 20fps or lower. You may wish to select a value around 20 – 30 to give your game a more authentic retro look.

- **PP anti-aliasing enabled:** This read-only field shows whether anti-aliasing is currently enabled in the Post-processing file you selected above (the field is hidden if you didn’t select a postprocessing file)
- **Quality level:** This read-only field shows which quality level is selected in Unity’s Quality Settings window.

- **Presets:** There are several buttons which can be used to apply preset settings based on different generations of home consoles. The presets are described below
    - **16-bit:**
        - 224p
        - 20 fps
        - Anti-aliasing off
    - **32-bit:**
        - 240p
        - 30 fps
        - Anti-aliasing off
    - **64-bit:**
        - 480p
        - 30 fps
        - Anti-aliasing on
    - **HD:**
        - 720p
        - 60 fps
        - Anti-aliasing on

When you select a preset, the Editor does several things:

- It adjusts the vertical resolution and framerate in the RenderSettings asset
- It adjusts the anti-aliasing setting in the post-processing settings file, if you selected one
- It attempts to select the corresponding quality in the project Quality settings
- It attempts to locate materials that use the "RetroCheckeredShader" and update the "Checker Size" parameter to go with the new resolution (see section "*Shaders*")

Additional Notes

Keep in mind that changing resolutions can mess up pixel fonts or UI layouts. I have done my best to configure the included scenes and prefabs so that they won't break when you change resolutions.

Most of the included presets are designed to emulate the resolution and frame rate of early 3D home consoles. If you don't want your game to look *that* retro, select the "HD" preset, or uncheck the "Use Custom Settings" box to render at the native resolution and refresh rate of the game window.

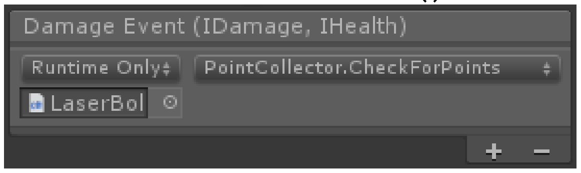
Scoring System

The scoring system includes several components and interfaces which are used together to award the player with points when he or she kills enemies. This system is somewhat complex because we only want the player to receive points when his or her weapons kill an enemy. We accomplish this by having the player's projectiles report points back to the player after they kill a target.

- **`IScore`**: Interface for objects that have a score. **`Player`** implements this interface.
- **`IGivePoints`:** Interface for objects that can give points to the player. **`ScoreValueComponent`** implements this interface.
- **`ICollectPoints`:** Interface for objects that collect points for a point receiver. **`PointCollector`** implements this interface.
- **`IReceivePoints`:** Interface for objects that receive points. **`Player`** implements this interface
- **`PointCollector`:** Collects points for the player. This component is attached to the *LaserBolt* prefab.
- **`ScoreValueComponent`:** Indicates that an object is worth points. This component is attached to enemies.

Usage Overview

By default, the Player class is configured to have a score and receive points. Attach the **`PointCollector`** component to the prefabs of any projectiles fired by the player. The **`ProjectileLauncher`** script will automatically configure these projectiles to return collected points to the player. In the inspector, add a new listener to the **DamageEvent** of the projectile's `DamageTrigger`. Have the new listener call the `PointCollector`'s `CheckForPoints()` method, so that the collector picks up enemy deaths.


Damage Event (IDamage, IHealth)
Runtime Only | PointCollector.CheckForPoints
LaserBol

Attach the **`ScoreValueComponent`** to each enemy and give them a score value. When the player's projectiles kill enemies, they will automatically add points from the `ScoreValueComponent` to the player's score.

Details

This section can be difficult to digest by reading it on its own. I recommend looking at the scene objects and prefabs referenced in this section, to more easily understand the relationship between the different parts.

The **`Player`** component implements the ***`IScore`*** and ***`IReceivePoints`*** interfaces. The **`PlayerSpacecraft`** also implements the *`IReceivePoints`* interface. When a `PlayerSpacecraft` is assigned to the `Player` object, the `Player` listens for the `GainPointsEvent` on the `PlayerSpacecraft`.

Each **`ProjectileLauncher`** nested under the player spacecraft GameObject will automatically detect that the `PlayerSpacecraft` implements `IReceivePoints`. When it fires, it will check if the projectile prefab implements the `ICollectPoints` interface (which it should, via the **`PointCollector`** component). If so, the `ProjectileLauncher` will set the **PointReceiver** property of the `PointCollector` to the `PlayerSpacecraft`.

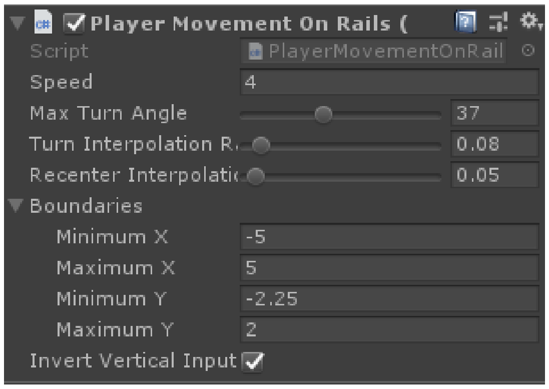
The **DamageEvent** of the projectile's **DamageTrigger** should be pointed to the **`CheckForPoints()`** method of the projectile's `PointCollector`. When the projectile hits an enemy, the PointCollector will check the damaged enemy and see if it now has 0 health. If so, it will collect points from the enemy and forward them back to the `PlayerSpacecraft`. This will invoke the PlayerSpacecraft's **GainPointsEvent**, which the `Player` component will detect. The `Player` will then add the points to its own **Score**.

Player and Camera Movement

Space Rail Shooter is designed for an on-rails experience, with the player generally restricted to limited horizontal and vertical movement while constantly progressing forward in a straight line. However, it is possible to redirect the player's heading, which we'll discuss later on.

The player's spacecraft, as well as the main camera, are placed together in the "**PlayerContainer**" GameObject, which uses the `PlayerContainer` script to move forward at a constant speed.

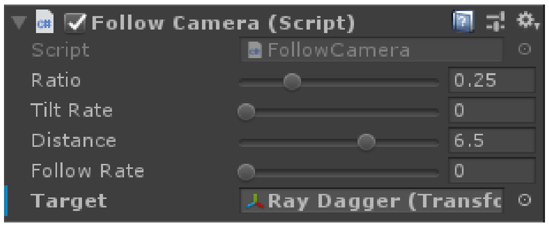
The **`PlayerMovementOnRails`** component is attached to the player spacecraft to enable limited horizontal and vertical movement within the player container. It has the following properties:



```
Player Movement On Rails (
Script                      PlayerMovementOnRail
Speed                       4
Max Turn Angle              37
Turn Interpolation R        0.08
Recenter Interpolati        0.05
Boundaries
  Minimum X                 -5
  Maximum X                 5
  Minimum Y                 -2.25
  Maximum Y                 2
Invert Vertical Input       ✓
```

- **Speed:** How fast the ship moves horizontally and vertically
- **Turn Interpolation Rate:** How quickly the ship turns
- **Recenter Interpolation Rate:** How quickly the ship returns to a face-forward orientation when the player is not providing any input
- **Boundaries:** The maximum distance that the ship can move from the center of the screen

The **`FollowCamera`** component is attached to the camera to make it track the player. It has the following properties:



```
Follow Camera (Script)
Script          FollowCamera
Ratio           0.25
Tilt Rate       0
Distance        6.5
Follow Rate     0
Target          Ray Dagger (Transfo
```

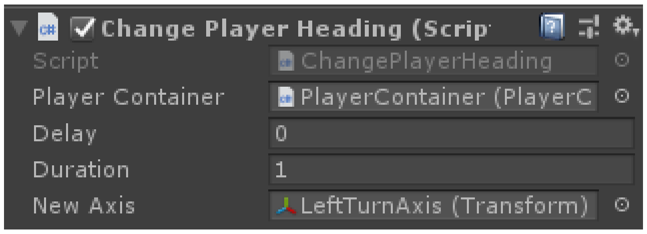
- **Ratio:** controls how far the camera rotates to face the spacecraft. A ratio of **1** means that the camera always faces the target exactly; a ratio of **0** means the camera never rotates at all.
- **Tilt Rate:** Controls how much the camera tilts with the player. This is set to **0** by default because tilting does not work with the starfield background in ExampleLevel1
- **Distance**: If "follow rate" is not 0, this controls the distance that the camera attempts to maintain from the target

- **Follow Rate:** Interpolation rate at which the camera follows the target. In the template, this is set to 0 because the camera is inside the PlayerContainer and never moves on its own
- **Target:** The GameObject that the camera will follow.

Changing the player's heading

In some cases you may not want the player moving in a straight line for the entire level. This could be to make the level layout more interesting, or because you want to give the player the option of taking an alternate path through the level. In either case, you'll need to rotate the PlayerContainer to face the new heading.

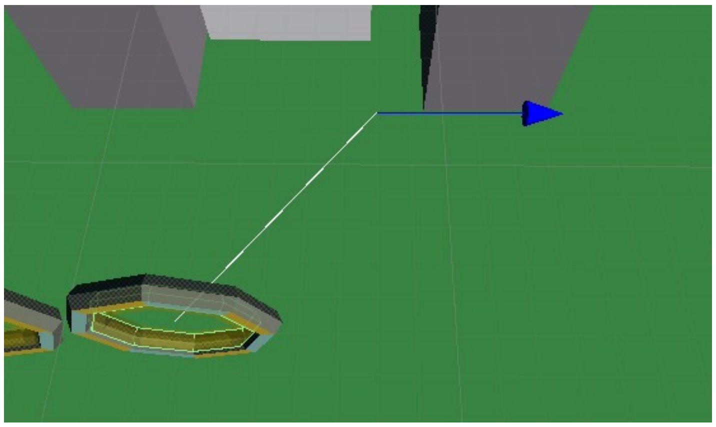
The `ChangePlayerHeading` script can be used to change the player's heading by moving the PlayerContainer onto a new vector. When activated, the PlayerContainer will be smoothly animated from its current position and rotation to the new axis' position and rotation.



Change Player Heading (Scrip

Script	ChangePlayerHeading
Player Container	PlayerContainer (PlayerC
Delay	0
Duration	1
New Axis	LeftTurnAxis (Transform)

ChangePlayerHeading is demonstrated in ExampleLevel2. Fly through one of the rings to trigger a 90-degree turn in that direction.

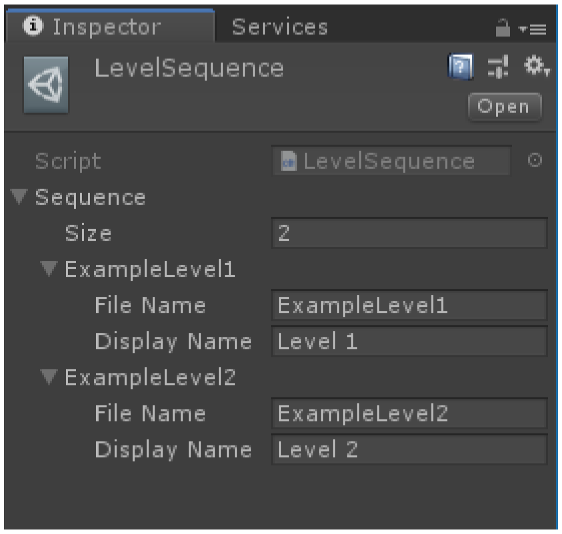


In the editor, a white line is drawn from the ChangePlayerHeading object's position to the position of the new axis, and a blue line is used to indicate the forward direction of the new axis.

Level Progression

The template includes support for a linear (non-branching) progression of levels, with a “Level Complete” screen to show the player’s score at the end of a level.

Defining the Level Sequence



Script	LevelSequence
Sequence	
Size	2
ExampleLevel1	
File Name	ExampleLevel1
Display Name	Level 1
ExampleLevel2	
File Name	ExampleLevel2
Display Name	Level 2

The **LevelSequence** script defines a Scriptable Object asset where you can specify what order your gameplay scenes will be arranged in, and give them display names.

Each entry in a LevelSequence contains two fields:

- **File Name:** the name of the scene file (no extension)
- **Display Name:** the name that will be displayed in the Level Complete screen after finishing a level

There is an included example at `Assets/OneManEscapePlan/SpaceRailShooter/Resources/LevelSequence` . You can create new LevelSequence assets from the “Create” menu (**Assets > Create > Level Sequence**) .

Progressing From One Level to the Next

The **LinearLevelManager** script is somewhat similar to Unity’s SceneManager, but is designed specifically for navigating between scenes in a LevelSequence. It provides methods for proceeding to the next level or a specific level, as well as retrieving basic information about the level sequence. If you are going to stick with a linear level progression, you’ll need to include a **LinearLevelManager** instance in each scene. The “System” prefab includes an instance of LinearLevelManager.

You should place a trigger at the end of each level that calls the `CompleteLevel()` function of the **Player** script. This, in turn, tells the PanelManager to spawn the **“Level Complete” panel,** which displays the player’s score and offers buttons to continue or exit. When the player clicks “Continue”, the LevelCompletePanel calls the NextLevel() function of the LinearLevelManager.

For an example of the level progression system in action, see ExampleLevel1. There is a trigger at the end of the level which displays the Level Complete panel, from which you can proceed to ExampleLevel2.

Branching Paths

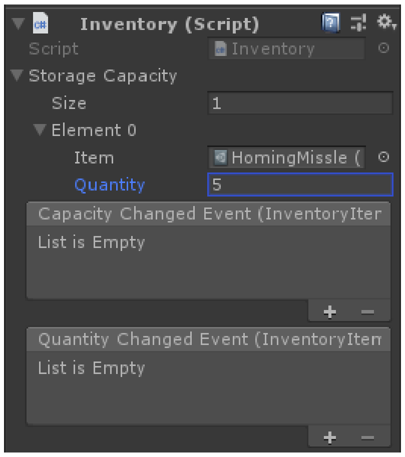
Although the template does not include built-in support for branching paths in level progression, you could implement this by extending the LevelCompletePanel script and overriding the `Awake()` and `Continue()` functions. You may wish to create a BranchingLevelManager as an alternative to the LinearLevelManager. Specific implementation details will depend on your goals (e.g. do you want the branching path to be controlled by what the player does during a level, or will the player select a path from a map screen?)

Power-ups and inventory

The template includes a system for power-ups which can be collected by the player, as well as an inventory system where collected items can be stored. Power-ups can activate a gameplay effect or add something to the player's inventory. Several example power-ups are included in the template.

The power-up and inventory systems have several components which should be attached to the player spacecraft:

- **Inventory**: This component must be attached to the player spacecraft to allow the player to collect inventory items from power-ups. Here you must also define the "Storage Capacity", which indicates what types of items the inventory can store and the maximum quantity of each. Use a negative quantity to indicate that the inventory can hold an unlimited number of the item.
- **Power Up Collector:** This component must be attached to the player spacecraft to allow the player to pick up power-ups. It will detect when the player is touching a power-up, and activate the power-up or add it to the player's inventory as applicable.


Inventory (Script)
Script: Inventory
Storage Capacity
Size: 1
Element 0
Item: HomingMissle (
Quantity: 5
Capacity Changed Event (InventoryIter
List is Empty
Quantity Changed Event (InventoryIten
List is Empty

In this example, the inventory can hold up to 5 Homing Missiles

To create the collectable power-ups, you create GameObjects which have a collider and power-up component attached. There are three types of power-up components included with the template:

- **Health Powerup:** Restores the player's health when it is collected
- **Shield Powerup:** Actives the "Barrier shield" when it is collected
- **Inventory Item Powerup:** adds a quantity of an item to the player's inventory when it is collected.

Inventory items must be defined using **Inventory Item** assets. There are two example inventory items in `Assets/OneManEscapePlan/SpaceRailShooter/Resources/InventoryItems`**.**

- **BarrierShield:** This item is not currently used, but could be used to make the barrier shield an inventory-based item instead of an instant-activate powerup
- **HomingMissile:** This item is used as ammunition by the "MissileLauncher" projectile launcher attached to the Ray Dagger player spacecraft.

You can create new inventory item assets using **Create > Inventory Item**.

There are three included power-up prefabs:

- **MisslePowerup**: This adds homing missiles to the player's inventory when collected
- **HealthPowerup**: This adds health to the player

- **Shield Powerup:** Creates a rectangular “barrier shield” in front of the player which blocks incoming enemy projectiles, but still allows the player to shoot through it. The shield lasts for a few seconds and then disappears.

  I tried to give the shield a unique “energy grid” look by having it unfold into a 9x5 grid of squares. However, this obscures the player’s view somewhat. If you don’t like the way the shield looks, you can replace the grid square sprite, reduce the alpha (opacity), or completely replace the visual effect with something else.

Power-ups can either be manually placed at specific points in the level, or they can be dropped by enemies when those enemies are killed. To make an enemy drop power-ups, add the “**Drop On Death**” component.

Creating Enemies

Enemies are composed from a variety of different components. The best way to learn how to build enemies is to study the included scenes and prefabs, but you can use this section for reference.

Each enemy must at minimum have the following components:

- **Mesh Renderer**
- **Collider**
- **Rigidbody**
- **Spacecraft:** Here you can set the name of the spacecraft (for example, "M39 Skybird"). This is not currently displayed in the game, but included for possible use in the future. The `Spacecraft` component also handles pooling if the enemy is spawned by a SpawnPool.
- **Health Component:** This is where you define the maximum health of an enemy, and can set up event handlers for when the spacecraft takes damage or is killed.

Most enemies should also have these components:

- **Score Value Component::** This is where you define how many points the player will receive if they kill this enemy (may not be applicable for invulnerable enemies, or minor enemies that you don't want to affect score)
- **Flash When Damaged**: This component causes the enemy to temporarily show a different material after taking damage. It is a good idea to add this to any enemy which takes more than one shot to kill; otherwise, it's very difficult to tell whether your shots are actually hitting enemies. I suggest using the "DamageMat" material, which flashes yellow-orange and looks suitably retro.
- Some type of movement pattern – discussed below

Movement Patterns

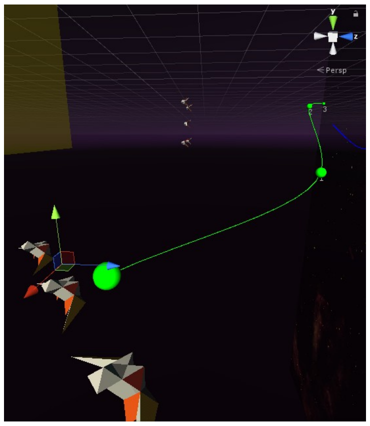
To give your game more life, you'll want most enemies to move in some manner. The *Space Rail Shooter* template comes with many different **movement pattern** and animation scripts which you can utilize to give your enemies unique styles of movement. Movement patterns and animations can range from simple repeating animations (such as bobbing up and down) to complex scripted paths.

Here is a breakdown of many of the included movement patterns and related scripts:

- **WaypointComponent**: This component simply contains a single Vector3 field representing a current waypoint. Many movement patterns interact with this, either by changing the waypoint's position, or moving the AI craft towards the waypoint.
- **Simple Waypoint Mover**: This component moves the waypoint at a fixed velocity, either relative to the GameObject or relative to the world.

- **Move Towards Waypoint:** This component causes the GameObject to approach its waypoint by turning towards the waypoint while moving forward.
- **Four-Point Pattern:** This component causes the GameObject to randomly zig-zag around the waypoint. See the "Elemental Dart" prefab for an example
- **Scripted Pattern:** This component lets you define a sequence of movement instructions, each consisting of a direction, speed, and duration. Although this allows for complex flight patterns, it is complicated and not particularly user friendly, so I don't recommend using it unless you really need it.. Generally, you'll want to use the Follow-Curve patterns when you want a ship to follow a path.

There are two movement patterns which are designed to move the waypoint along a path (defined by a SplineCurve) . This can be used to give enemies complex repeating movement patterns (e.g. to create an enemy that moves in a continuous corkscrew pattern), or used in conjunction with triggers to create complex scripted animations.



The Follow-Curve patterns must be assigned a **Spline Curve** that defines the actual path that the enemy will follow. The ship will follow the shape of the curve path *relative to the starting position of the ship.* For this reason, it's generally a good idea to place the enemy right where the path starts, so you can see where the path will actually lead it.

For prefab enemies, you can include the Spline Curve with the prefab. It will automatically be un-parented when the prefab is instanced, so that the curve doesn't move with the prefab.

It's difficult to fully explain the follow-curve behavior in text. You can see examples of the follow-curve patterns in action in the "ExampleLevel1" scene, as well as the "FollowCurveDemo" scene.

- **Follow-Curve-By-Speed Pattern**: This component causes the waypoint to move along a path at a fixed speed (the speed is not affected by the distance between points on the path).
- **Follow-Curve-By-Time Pattern:** This component causes the waypoint to move along a path over a fixed amount of time (e.g. 5 seconds). The movement speed at a given position along in the curve depends on the distance between points in the curve; if the points are closer together, the GameObject will move slower; if they are further apart, the GameObject will move faster.

There are some additional animation scripts you might use to further enhance enemy movement:

- **Simple Mover:** Moves the GameObject at a constant speed in a specific direction, either relative to the GameObject or relative to the word. Set the Movement Space to "Self" and use a positive Z value to make the enemy move forward

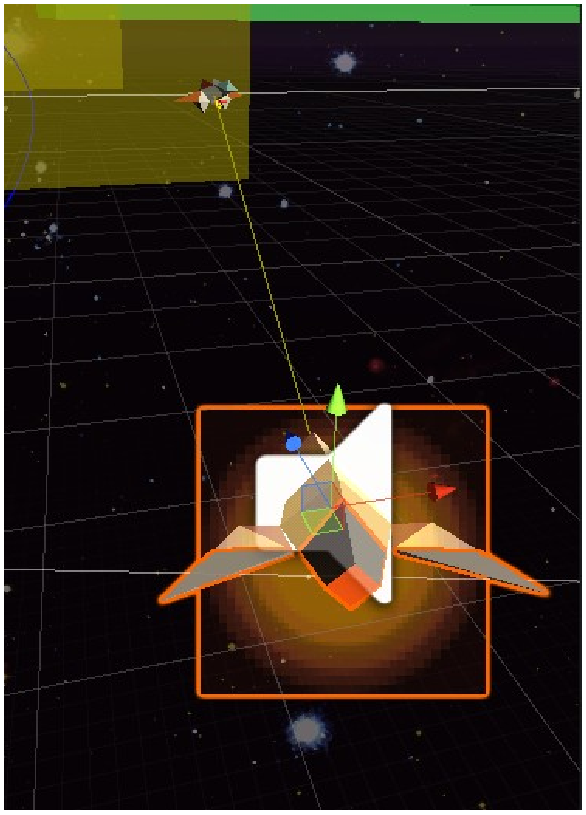
- **Sine Wave Mover:** Bobs the GameObject back and forth along a selected axis. Many of the included ship prefabs use this to make their animations more interesting or to make the enemy more difficult to shoot.
- **Look At Player:** Causes the GameObject to rotate to face the player. When combined with Simple Mover for forward movement, this causes an enemy to home in on the player. See the "Preybird" prefab for an example

Demo Scenes

The project comes with several small scenes under `SpaceRailShooter/Scenes/Demos`. These scenes are for developer reference and non-interactive; they demonstrate various features of the template, and can be handy if you want to see how certain things work. Some demo scenes do not contain a camera, so you may want to ensure that "Maximize on Play" is not selected before running them.

- **FollowCurveDemo**: Demonstrates the differences between **`FollowCurveByTimePattern`** and **`FollowCurveByDistancePattern`**; also shows demonstrates what the "loop" option does
- **HomingMissileDemo**: Demonstrates how the homing missile tracks targets, and how the explosion can destroy multiple targets
- **HyperspaceJumpDemo**: Demonstrates the `HyperspaceJumpEffect` with both "Jump In" and "Jump Out" settings. Note that for the purposes of the demo, two separate carriers are used to create the illusion of a single ship jumping in and out.
- **MovementPatternsDemo**: Demonstrates many of the movement patterns that can be applied to NPC/enemy spacecraft, along with examples of how patterns can be combined for more complex behaviour.
- **ScoringSystemDemo**: Demonstrates how projectiles fired by the player can give points to the player when they destroy enemies. You can review the components attached to each object in the scene to get a better understanding of the relationship between different parts of the scoring system.

“Draw Connections” system



The Draw Connections system is a custom Editor feature that can make your life as a developer easier by visually illustrating the connections between different objects in your scene. For example, in the image on the right, a yellow line is drawn from the selected homing missile to its target. This makes it easy to verify that the missile has a target and to make sure that the missile is heading towards its intended target.

To use the Draw Connections system, just add the “DrawConnections” attribute above a field that references another GameObject or MonoBehaviour in the scene, or above a field containing any type of UnityEvent.

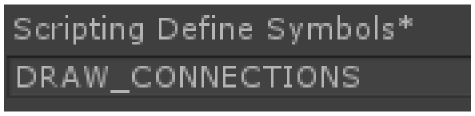
For example, the yellow line drawn from the missile to its target is achieved through this code in **`TurnTowardsTarget`:**

```
[DrawConnections(ColorName.PulsingYellow)]
[SerializeField] private Transform target;
```

For GameObject/MonoBehaviour references, a line will be drawn to the referenced object if it is non-null and present in the scene. For UnityEvents, lines will be drawn to all scene objects that have listeners to the event configured in the inspector.

Connections are only drawn from objects that are currently selected in the scene hierarchy. If you want to visualize *all* connections between *all* objects in your scene, try selecting all objects in the scene with **Edit > Select All** (ctrl-A).

Note: the Draw Connections system requires “DRAW_CONNECTIONS” to be present in your scripting define symbols *(under Player Settings)****. To disable the feature, simply remove this define symbol.***



Scripting Define Symbols*

DRAW_CONNECTIONS