SPACE COMBAT KIT 2.0

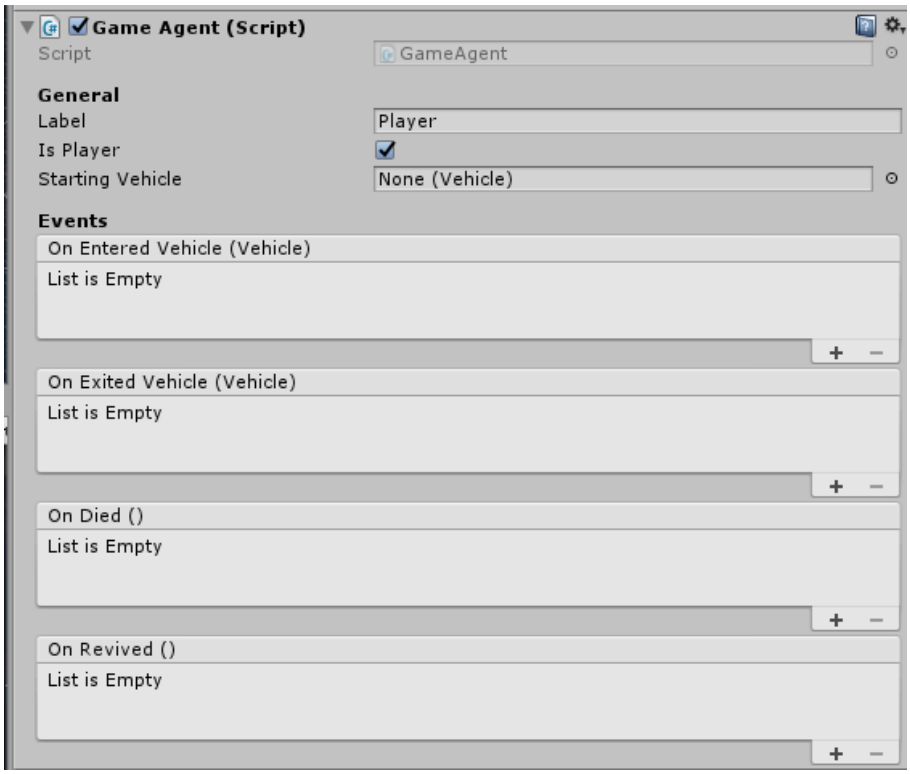
Contents

Creating The Player Or AI

The first thing to do to create a scene with the Space Combat Kit is to add a player or AI. They are both created in the same way using the Game Agent component.

Game Agent Overview

The **GameAgent** component represents a player or AI in the game, and is defined as an entity that can enter and exit different vehicles. A Game Agent does not have a physical representation in this kit – even first person characters are considered to be vehicles that a game agent controls. This makes it easy to switch between, for example, a spaceship and a character.



Inspector settings:

- **Label**: The label that appears on the radar when tracking a vehicle controlled by this game agent.

- **IsPlayer**: Whether the game agent is a player or AI.
- **Starting Vehicle**: The vehicle that this game agent will enter when the scene starts.
- **OnEnteredVehicle**: Unity Event called when this game agent enters a vehicle (may be null).
- **OnExitedVehicle**: Unity Event called when this game agent exits a vehicle.
- **OnDied**: Unity Event called when the vehicle that this game agent is in is destroyed.
- **OnRevived**: Unity Event called when this game agent is revived after dying.

Creating The Player

To create the Player:

1. Drag the provided Player prefab into the scene.

OR, to create your own player:

1. Create a new gameobject and call it 'Player' (or whatever else you like).
2. Add a GameAgent component.
3. Fill out the details in the inspector. Ensure that the *Is Player* checkbox is checked.

Creating An AI

To create an AI:

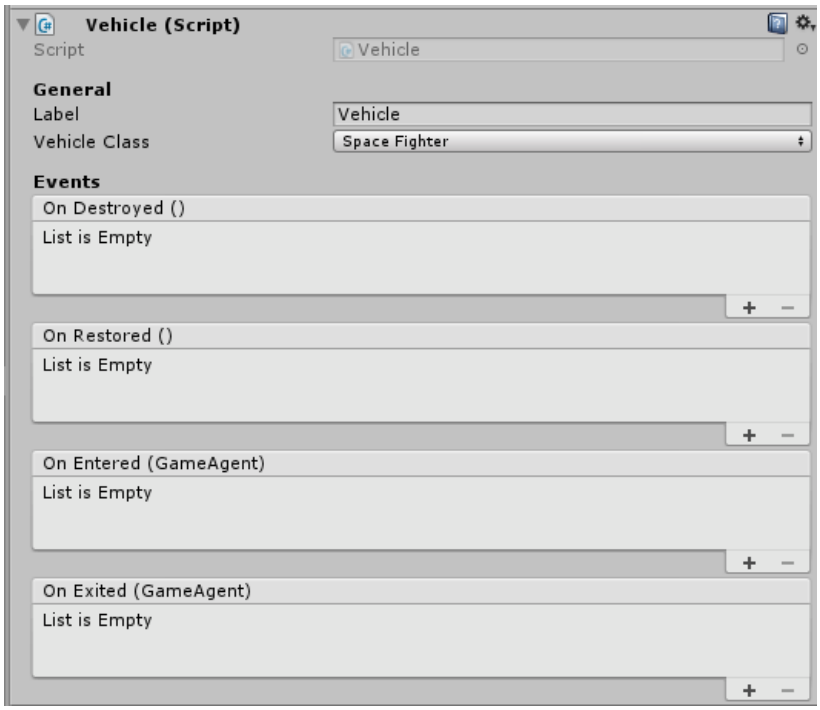
1. Create a new gameobject and call it 'AI (or whatever else you like).
2. Add a GameAgent component.
3. Fill out the details in the inspector. Ensure that the *Is Player* checkbox is not checked.

Creating a Vehicle

A vehicle is represented by a gameobject with a Vehicle component attached to its root transform, and can be entered and exited by a game agent. The vehicle can be a first person character, a space fighter, a capital ship, or any vehicle you create for your game.

Vehicle Overview

The **Vehicle** component represents a vehicle that a game agent can enter and exit, and must be added to the root transform of the vehicle.



Inspector settings:

- **Label**: The label that appears, for example, in the loadout menu for this vehicle (the Label field on the Game Agent controlling this vehicle is what appears on radar when tracking the vehicle).
- **Vehicle Class**: Provides a way to differentiate between different types of vehicles, for example with different camera controllers. Add vehicle classes by adding values to the *VehicleClass* enum in the VehicleClass.cs file.
- **OnDestroyed**: Unity Event called when this vehicle is destroyed.
- **OnRestored**: Unity Event called when this vehicle is restored after being destroyed.
- **OnEntered**: Unity Event called when a game agent enters this vehicle.
- **OnExited**: Unity Event called when a game agent exits this vehicle.

Creating a Vehicle

To create a vehicle, either add a vehicle prefab to the scene:

1. **To add a space fighter**, drag the SpaceFighterFriendly prefab into the scene.
2. **To add a first person character**, drag the FirstPersonCharacter prefab into the scene.
3. **To add a capital ship**, drag the Battleship prefab into the scene.

OR, to create your own vehicle:

1. Add a new gameobject (e.g. your ship model) to the scene.
2. Add a Vehicle component to the root transform.

Note: The Vehicle component is just the start of making a vehicle, throughout the documentation we'll be building the rest of the vehicle.

Entering/Exiting Vehicles

Entering a Vehicle When The Scene Starts

To make the Game Agent (player or AI) enter a vehicle when the scene starts, drag the vehicle (which must be present in the scene, cannot be a prefab) into the *Starting Vehicle* field in the inspector of the **GameAgent** component.

Entering a Vehicle Through Code

To make the Game Agent (player or AI) enter a vehicle during gameplay, call the EnterVehicle method on the **GameAgent** component and pass a reference to the vehicle, i.e.:

CODE:

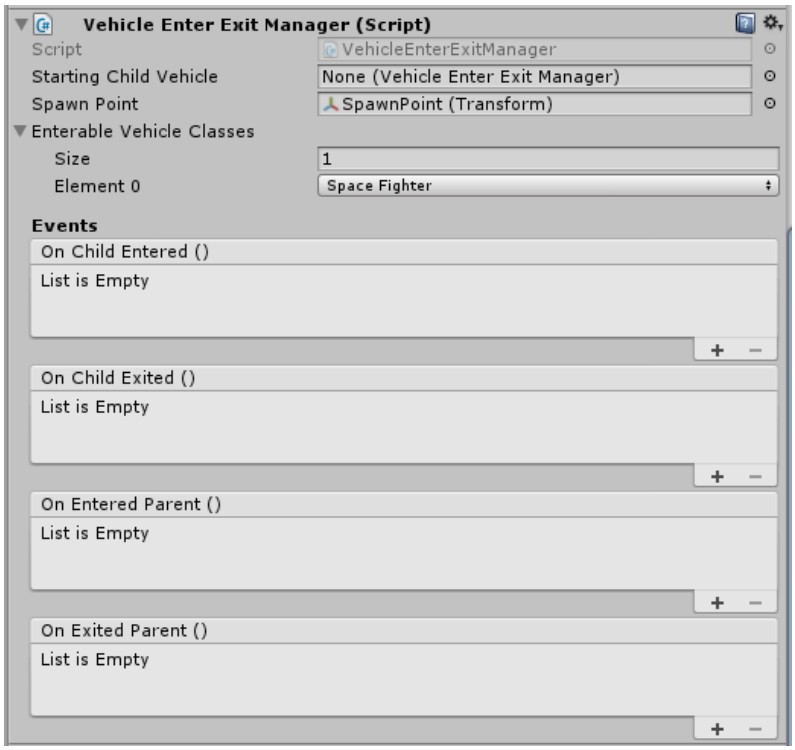
```
myGameAgent.EnterVehicle(myVehicle);
```

Entering/Exiting As A Gameplay Mechanic

To create gameplay that includes entering and exiting a vehicle, the **VehicleEnterExitManager** component is provided. Remember that even a character is a vehicle, so what is really happening is that the game agent switches vehicles – but the **VehicleEnterExitManager** component 'remembers' which vehicle it came from (e.g. a character), which it can then exit back to.

The Vehicle Enter Exit Manager

The **VehicleEnterExitManager** component manages entering and exiting vehicles during gameplay.



Inspector Settings:

- **Starting Child Vehicle**: Initialize the vehicle to exit to when the game agent exits this vehicle.
- **Spawn Point**: The transform that represents the position and rotation that the 'exit' vehicle appears when exiting this vehicle.
- **Enterable Vehicle Classes**: The vehicle classes that this vehicle can enter and exit.
- **On Child Entered**: Unity Event called when a vehicle enters this vehicle (it becomes the 'child' vehicle, and this vehicle becomes the 'parent' vehicle).
- **On Child Exited**: Unity Event called when the child vehicle exits from this vehicle.
- **On Entered Parent**: Unity Event called when this vehicle enters another vehicle.
- **On Exited Parent**: Unity Event called when this vehicle exits another vehicle.

Setting Up The Gameplay

To set up entering/exiting between two vehicles (in this example, a character and a space fighter) during gameplay:

1. Make sure the character has a collider somewhere in its hierarchy.
2. Add a **VehicleEnterExitManager** component to the root transform of both vehicles.
3. Add a trigger collider to the space fighter that represents the area that the character has to be inside to enter it.
4. Add a transform to the space fighter that represents where the character will appear after exiting it, and drag it into the *Spawn Point* field in the inspector of the space fighter's **VehicleEnterExitManager** component.

When the character enters the trigger collider of the space fighter, the space fighter will detect it and add itself to the *Enterable Vehicles* list on the **VehicleEnterExitManager** component that is on the character.

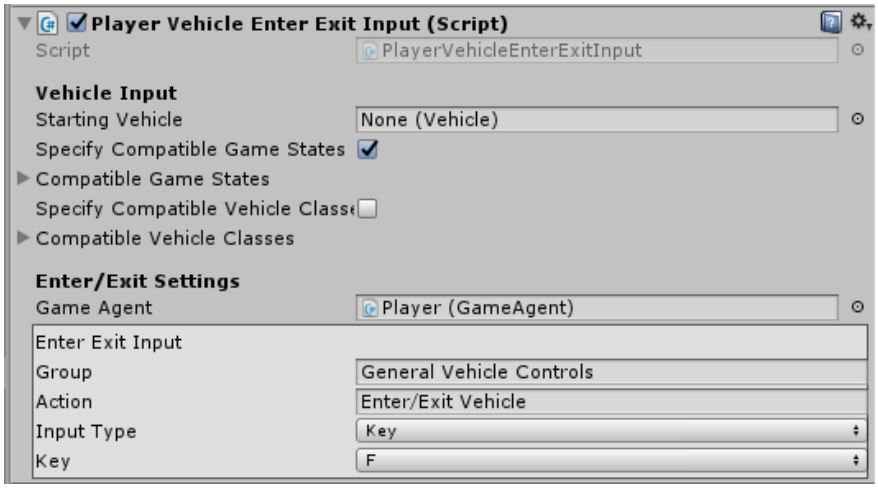
To make the character enter the space fighter when it is inside the trigger collider, call the *EnterParent* method on the character's **VehicleEnterExitManager** component, and pass the index of the space fighter in the *EnterableVehicles* list (don't pass any value to just enter the first available parent vehicle, or if you know there is only one option).

CODE:

```
characterVehicleEnterExitManager.EnterParent();
```

Setting Up The Input

To handle input for entering and exiting vehicles for you, the **PlayerVehicleEnterExitInput** script has been provided, which is a vehicle input script (see the Input section for more general information on vehicle input).



Inspector Settings:

- (See the Input section for **Vehicle Input** base class settings).
- **Game Agent**: The player game agent that this input is for.
- **Enter Exit Input** (Custom Input): The input for entering and exiting a vehicle.

Creating a Vehicle Camera

This kit includes a vehicle camera system that makes it easy to:

- Create different camera controllers for different vehicles.
- Create different camera views and configure how the camera behaves for each one.
- Track a Game Agent during entering and exiting of different vehicles.

Creating a Vehicle Camera

To add a vehicle camera to the scene:

1. Drag the provided *VehicleCamera* prefab into the scene.

OR, to create your own vehicle camera:

1. Add a **VehicleCamera** component to your camera.

Creating Camera Controller(s)

A vehicle camera controller defines how the camera behaves when tracking a vehicle of a specific vehicle class (which is a value that is set on the **Vehicle** component on the root transform of a vehicle). It is defined as a script that extends the **VehicleCameraController** class.

When the vehicle camera tracks a new vehicle, it looks in its hierarchy for a camera controller with a *Vehicle Class* (set in the inspector) that matches that of the vehicle. If one is found, it starts running. You can set up multiple camera controllers in the Vehicle Camera hierarchy for different vehicles, but only one is run at a time.

To add a vehicle camera controller:

1. Add a new gameobject as a child of the vehicle camera.

2. For a space fighter, add a **SpaceFighterCameraController** component.
3. For a first person character, add a **FirstPersonCharacterCameraController** component.
4. For a capital ship, add a **CapitalShipCameraController** component.

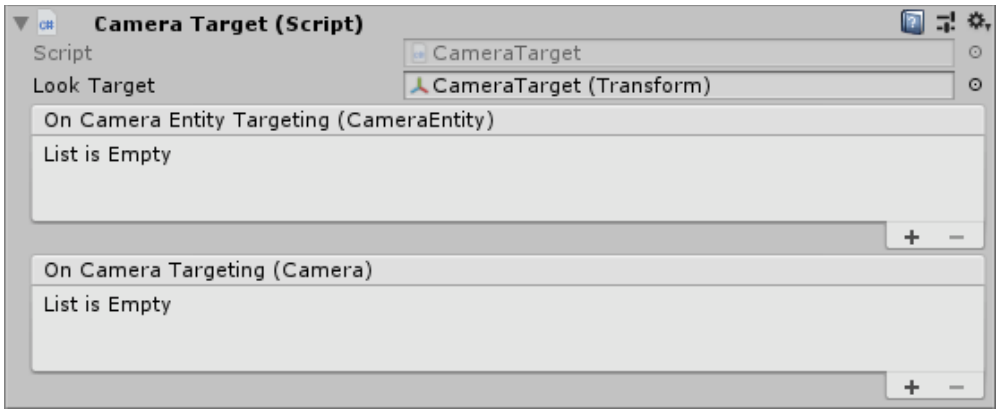
OR, to create your own camera controller:

1. Extend the VehicleCameraController class with your own script.
2. Add your script to the hierarchy of the VehicleCamera component.

Creating A Camera Target

Any vehicle, such as a ship, that you wish to follow with the camera must have a **CameraTarget** component added to the root transform. This component gathers information on the camera view targets for the vehicle so that they can be easily accessed by the camera scripts.

Add a **CameraTarget** component to the root transform of your ship or vehicle.



Inspector settings:

- **Look Target**: Provides a transform for the camera to look at when the camera is following a vehicle.
- **OnCameraEntityTargeting**: Unity Event called when a camera starts following this target, passing a reference to the **CameraEntity** component.
- **OnCameraTargeting**: Unity Event called when a camera starts following this target, passing a reference to the **Camera** component. May be used to pass a camera reference for the HUD etc.

Creating Camera Views

Note that the vehicle prefabs provided in this kit already have camera views set up, so unless you are setting up a new vehicle or need a different type of camera view, you can skip this.

To create a camera view for your vehicle:

1. Add a new game object as a child of the vehicle.

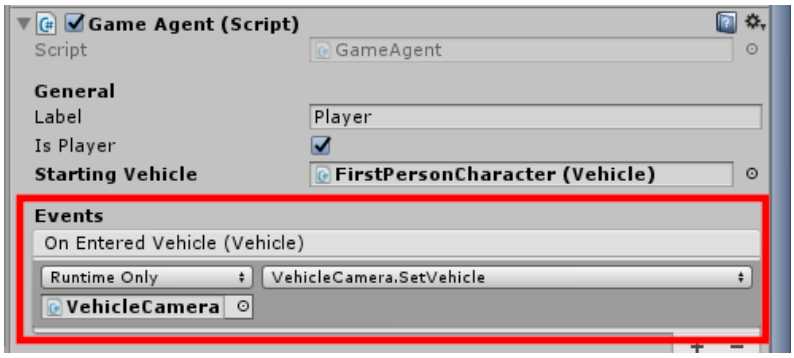
2. Position and rotate this game object according to where you would like the camera to aim for in terms of position and rotation.
3. Add a **CameraViewTarget** component and customize the values in the inspector.

Tracking a Vehicle With The Camera

To make the vehicle camera focus on a specific vehicle when the scene starts, open the inspector of the VehicleCamera component, and drag your vehicle (spaceship, character etc) into the *Starting Target Vehicle* field.

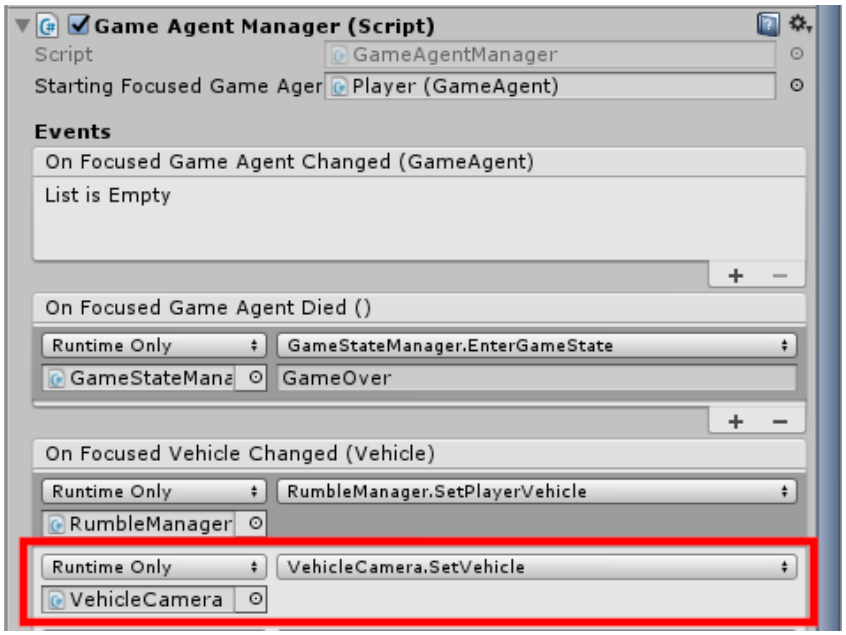
Tracking a Game Agent With The Camera

To make the vehicle camera focus on whatever vehicle a specified Game Agent is in, add the **VehicleCamera** component's *SetVehicle* method to the **GameAgent** component's *On Entered Vehicle* event:



Tracking The Focused Game Agent With The Camera

If you are using the Game Agent Manager, then, to make the vehicle camera always focus on whatever vehicle the focused Game Agent is controlling, add the **VehicleCamera** component's *SetVehicle* method to the **GameAgentManager** component's *On Focused Vehicle Changed* event:



Input (Player or AI)

In this kit, a game agent (player or AI) controls a vehicle through vehicle input scripts, which are added to the hierarchy of the **GameAgent** component. When the Game Agent enters a vehicle, it activates all compatible input scripts in its hierarchy.

A vehicle input script is defined as a script that extends the **VehicleInput** base class.

Multiple input scripts can be created for each vehicle, and input scripts can be shared between vehicles with common functionality.

Creating a Vehicle Input Script

This kit already contains input scripts for the provided vehicles, which will be looked at in greater detail in specific sections of this documentation.

To create a new vehicle input script, simply extend the **VehicleInput** base class. This base class contains the following key methods which can be overridden as necessary in the scripts you create for your game.

- **SetVehicle**: Set the vehicle that the input script passes input to (this calls the *Initialize* method, see below).
- **Initialize**: returns True if the vehicle is compatible, False if not compatible. In this method, make sure that references to all the necessary components on the vehicle are available.
- **StartInput**: starts the input (if it is already initialized).
- **StopInput**: stops the input (if it is already running).
- **InputUpdate**: Called every frame that the input script is running. Add the input code here.

Creating Vehicle Input Scripts For The Player

To create vehicle input scripts for the player:

- Extend the VehicleInput base class with your script.
- Override the *Initialize* method to make sure the vehicle has any components you need to send input to.
- Override the *InputUpdate* method to add your code that checks for input and sends it to relevant components on the vehicle.

Creating Input Scripts For The AI

Input for the AI follows the same format as for the player, except that the contents of the *InputUpdate* method contains the code for how the AI should behave.

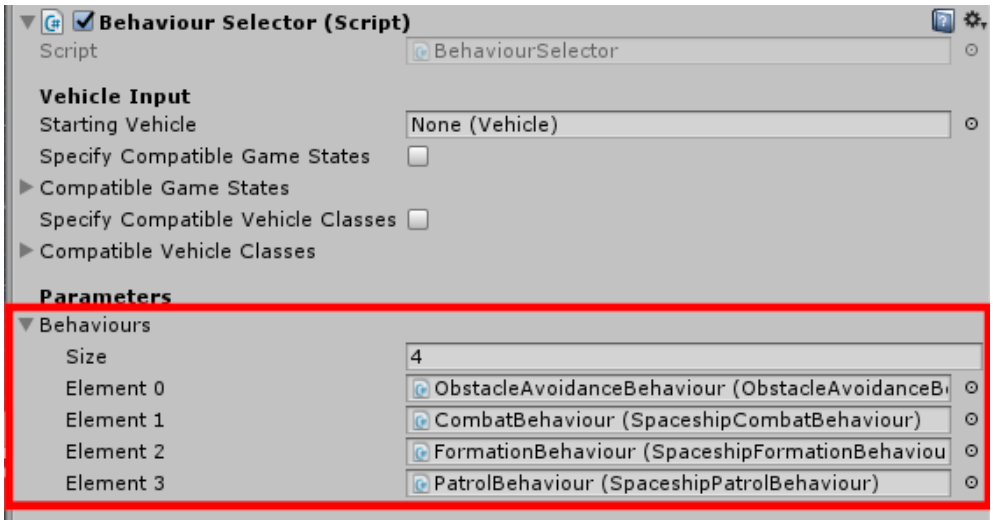
However, the code for how the AI should behave can be very complex, with many different behaviours for many different situations. For this reason, a system has been created to be able to split the behaviours into their own scripts, so that the AI's vehicle input script simply selects which behaviour to run at any time.

AI Behaviours Overview

To make creating the AI easier, it is recommended to split the code for the AI into different scripts that each define a single 'behaviour'. Then the main vehicle input script decides which behaviour to run based on the situation.

Each behaviour can succeed or fail. For example, if there are no targets around, a combat behaviour would fail. This makes it possible to move onto a different behaviour if the first one is not relevant.

For example, in the following picture you can see the **BehaviourSelector** script, which is a vehicle input script for the AI that manages a number of behaviours. It goes through each behaviour in the list defined in the inspector until it finds one that succeeds, at which point it waits until the next frame and starts again. Because each behaviour is tried in order, top behaviours are prioritized first, and the bottom ones only run if the others are not successful.



You can see that the AI will first try to avoid obstacles. If there are no obstacles it will look for enemies to fight. If there are none, it will try to fly in formation. If there are no formation members (or it is the formation leader) it will simply patrol the area.

AI Vehicle Behaviours

In the example provided above, each of the behaviours in the list is an AI Vehicle Behaviour, which is a behaviour for controlling a vehicle.

An AI Vehicle Behaviour is defined as a class that extends the **AIVehicleVehaviour** class. The key methods which can be overridden in your own scripts for this class include:

- **SetVehicle**: Set the vehicle that this behaviour is controlling (this calls the **Initialize** method, see below).
- **Initialize**: Return True if the behaviour is compatible with the vehicle, False if not. When extending this base class, use this method to check if the vehicle contains the components that your behaviour will need to control.
- **BehaviourUpdate**: the code for implementing the behaviour is added here.
- **StartBehaviour**: Start running the behaviour (enables the possibility of multi-frame behaviour).
- **StopBehaviour**: Stop running the behaviour.

AI Spaceship Behaviours

The **AISpaceshipBehaviour** class extends the **AIVehicleBehaviour** class to add a Ship PID controller (see below) and is essentially a base class for any behaviour that involves movement and steering in 3D (e.g. for spaceships).

Ship PID Controller

NOTE: PID controllers are a little complex and it is not necessary to understand them to use them effectively. The default values should work for most cases quite well. The following information is simply for anyone wishing to understand them better.

A PID (Proportional-Integral-Derivative) Controller, in simple terms, is something that takes a value known as the 'error' (e.g. the angle toward a target destination, when steering) and minimizes it by applying a function.

- The **Proportional** part involves multiplying the error by a value to generate a proportional correction value.
- The **Integral** part involves adding a cumulative small correction factor over time, such that the final correction value never lags behind the desired result.
- The **Derivative** part involves multiplying the rate of change of the error by a damping value to prevent it from overshooting the target value.

EXAMPLE

Let's say we have a ship that needs to rotate 10 degrees around the Y axis in order to be facing its target, which is an enemy ship.

- The **Proportional** value takes the error (10 degrees) and multiplies it by the Proportional coefficient (set in the inspector, let's take an example of 0.01) to arrive at a result of 0.1. Because it is desired to minimize the error, this value is inverted (multiplied by -1) and so the first part of the PID result is -0.1. This can be sent to the input to turn the ship back around to the target.
- However, if the target ship is moving fast across the front of the AI, the target can move faster than the Proportional part of the AI's steering can turn the AI's ship to face it. The

Integral value adds a small extra value to the input over multiple frames, based on the size of the angle between the AI and its target. This means that the AI will eventually 'catch up' to the target even if the target is moving away.

- The **Derivative** value is not always necessary, but it helps prevent the AI from building too much momentum and overshooting the target, which can lead to unwanted oscillating 'zig-zag' steering effects. The **Derivative** value applies a correction factor in the opposite direction to the **Proportional** value, based on how fast the angle is closing.

Modules

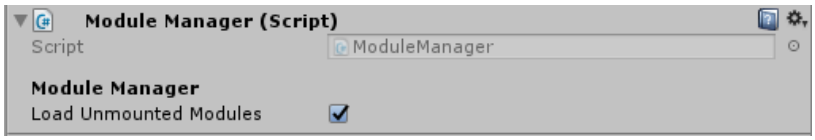
To enable you to build the functionality of a vehicle in a customizable way, this kit uses a modular system where (most) functionality is packed into modules that can be swapped in and out, even during gameplay. The structure of this involves three parts:

- **Module Managers**: The ModuleManager base class enables a derived class to be notified when a module is loaded or unloaded from the vehicle, and is necessary as a base class for components that manage or reference modules on the vehicle.
- **Module Mounts**: Module mounts enable specified types of modules to be loaded at specified points on the vehicle. Many modules can be loaded at a module mount and ready to mount, but only one can be mounted at a time (the rest are hidden/disabled).
- **Modules**: Modules are small, self-contained bits of functionality that make up a vehicle. For example, guns, missiles, shield generators, powerplants and more. They can be added to the vehicle in the editor, or selected in a loadout menu and loaded onto the vehicle when the game starts, and can be swapped out with other modules or removed at any time.

Module Managers

Overview

The ModuleManager class is a base class for any script that needs to deal with modules on a vehicle.



Inspector Settings:

- **Load Unmounted Modules**: Whether to call the event functions (see below) for modules that are not mounted by module mount (i.e., freely added to the vehicle hierarchy).

The four key methods in this class, which can be overridden in your scripts, are:

- **OnModuleMounted**: Called when a module is mounted on any of the vehicle's module mounts.
- **OnModuleUnmounted**: Called when a module is removed from any of the vehicle's module mounts.
- **OnModuleMountAdded**: Called when a new module mount is created on a vehicle.
- **OnModuleMountRemoved**: Called when a module mount is removed from the vehicle.

By overriding these functions, any scripts you create for your vehicles can always stay on top of which modules are currently loaded onto the vehicle, and update their references to components on the modules.

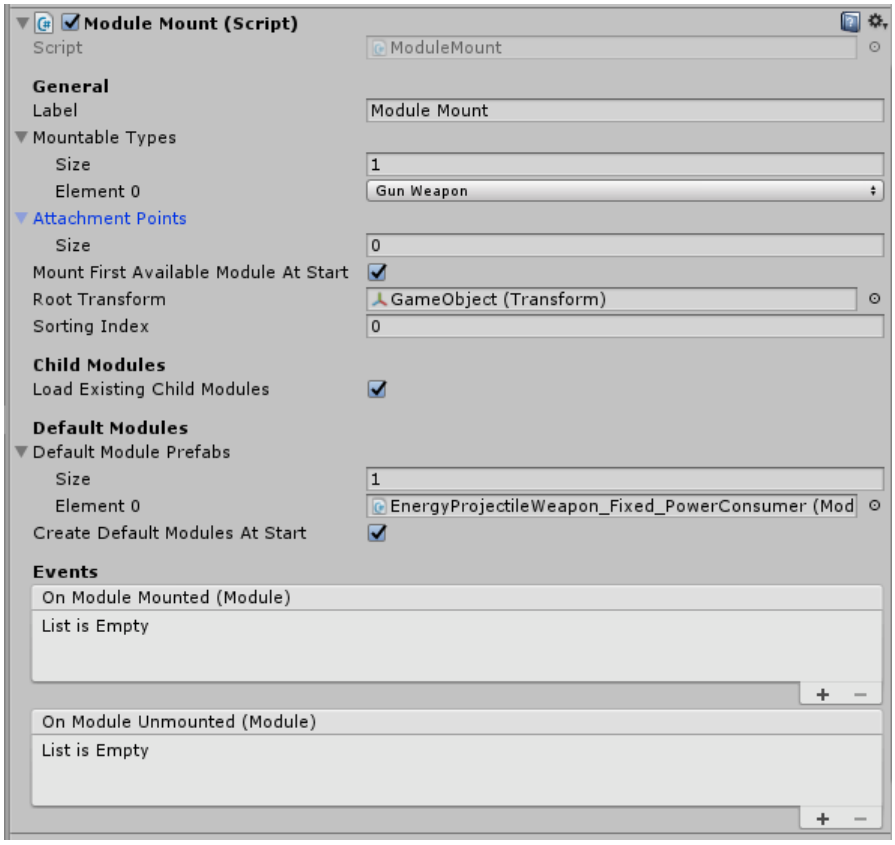
Creating a Module Manager

To create a module manager of your own, simply extend the **ModuleManager** base class with your own scripts, and override its functions to update references to relevant components on the modules when they are mounted or unmounted on the vehicle.

Module Mounts

Overview

The ModuleMount class represents a point on a vehicle where specific types of modules can be loaded.



Inspector Settings:

- **Label**: The label that appears in the loadout menu for this module mount.
- **Mountable Types**: The types of modules that can be loaded onto this module mount.
- **Attachment Points**: The attachment point(s) for the module (defaults to the transform on which the component is added).
- **Mount First Available Module At Start**: Whether to mount a module at the start, if any mountable modules are created/found.

- **Root Transform**: The root transform of the vehicle, which is a reference that is passed to modules.
- **Sorting Index**: The index of the module for purposes of ordering in a menu.
- **Load Existing Child Modules**: Whether to load modules that are found as children of the module mount.
- **Default Module Prefabs**: All the module prefabs that should be created/loaded on this module mount.
- **Create Default Modules At Start**: Whether to instantiate the default module prefabs at the start of the scene.
- **On Module Mounted**: Unity Event called when a module is mounted on this module mount.
- **On Module Unmounted**: Unity Event called when a module is unmounted from this module mount.

Adding Modules During Gameplay

To add a mountable module during gameplay, simply call the **ModuleMount** component's *AddMountableModule* function, passing a reference to the module (which must be already in the scene, cannot be a prefab).

EXAMPLE CODE:

```
myModuleMount.AddMountableModule(myNewModule);
```

This adds the module to the **ModuleMount** component's *MountableModules* list (if it is compatible with the module mount settings).

Mounting Modules During Gameplay

To mount a module during gameplay, simply call the **ModuleMount** component's *MountModule* function, passing the index of the module within the module mount's *MountableModules* list.

EXAMPLE CODE:

```
myModuleMount.MountModule(0);   // Mounts the module at index 0
```

Building The Vehicle

In the following sections, you'll learn how to build every part of a vehicle, including adding the key components to the vehicle and setting up the input.

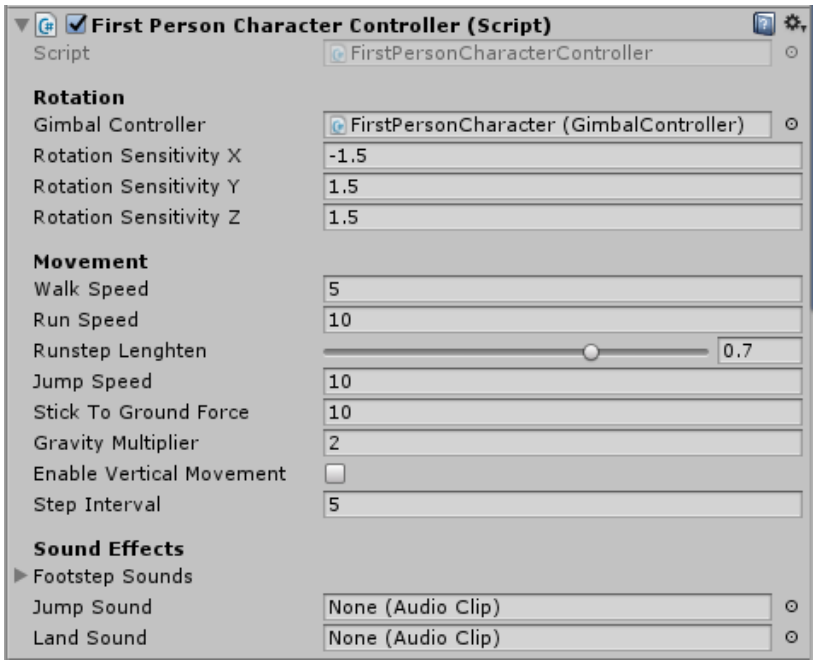
Controlling a First Person Character

Setting Up The Controller

To set up the controller for a first person character, it is very similar to setting it up with Unity's Standard Assets, and the FP character in this kit also uses Unity's CharacterController component.

To set up the first person character:

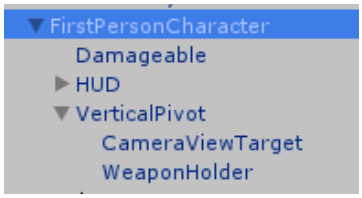
1. Add a **CharacterController** component to the root transform of the character, and set the settings in the inspector.
2. Add a **FirstPersonCharacterController** component to the root transform of the character, and set the settings in the inspector.
3. Add a **GimbalController** component to the root transform of the character, and drag it into the *Gimbal Controller* field in the inspector of the **FirstPersonCharacterController** component.



Inspector Settings:

- **Gimbal Controller:** The gimbal controller used to rotate the horizontal and vertical axes when looking around.
- **Rotation Sensitivity X, Y and Z:** How fast the rotation of the camera is when looking around.
- **Walk Speed:** How fast the player moves when walking.
- **Run Speed:** How fast the player moves when running.
- **Runstep Lengthen:** Reduce this value to take longer steps when running.
- **Jump Speed:** Affects the speed and height of the player's jump.
- **Stick To Ground Force:** The force keeping the player on the ground when the player has feet on the ground (e.g. to prevent 'jumps' when reaching the top of a ramp).
- **Gravity Multiplier:** How much gravity affects the player.
- **Enable Vertical Movement:** Enable the player to move on the vertical axis (e.g. in space).
- **Step Interval:** The interval between each time the player's foot hits the ground.
- **Footstep Sounds:** An array of footstep sounds that will be randomly selected from.
- **Jump Sound:** The sound effect for when the player leaves the ground.
- **Land Sound:** The sound effect when the player lands on the ground.

For the inspector of the **Gimbal Controller**, make sure that the *Horizontal Pivot* is the root transform of the character. The *Vertical Pivot* should be an empty child transform of the character, and the camera view target is added under here. The Hierarchy view for the character should look similar to this:



This means that looking left/right will rotate the entire character, but looking up/down will rotate the Vertical Pivot.

Also, make sure that the *Parent Camera* field is set to *True* on the **Camera View Target** component, so that the camera position and rotation are directly controlled by the gimbal controller and there is no lag.

Setting Up The Input

To set up the input for controlling the First Person Character:

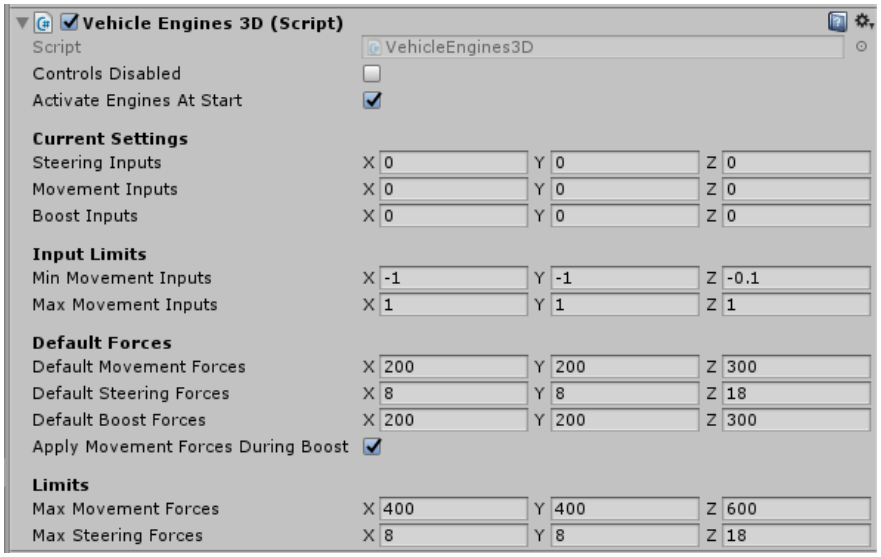
1. Create a new gameobject as a child of the Player's GameAgent component.
2. Add a **PlayerFirstPersonCharacterInput** component to the new gameobject.
3. Customize the controls in the inspector.

Controlling A Spaceship

Setting Up The Controller

To set up the controller on a spaceship, add a **VehicleEngines3D** component to the root transform (or **PoweredVehicleEngines3D** if you are using Power Management as a mechanic in your game).

The VehicleEngines3D component is a generic controller for steering and moving an object around in 3D. It is used for the space fighter, capital ship, missiles and anything else you may want to create for your game that moves and rotates in 3D.



Inspector Settings:

- **Controls Disabled**: Enable and disable the controller.
- **Activate Engines At Start**: whether the engines should be 'on' when the game starts. Engine activation enables the possibility of an idling visual and sound effects.
- **Steering Inputs**: -1 to 1 input values for steering (rotation) around each axis (local to the vehicle). These values are modified by the input script.
- **Movement Inputs**: -1 to 1 input values for movement (translation) along each axis (local to vehicle). These values are modified by the input script.
- **Boost Inputs**: -1 to 1 input values for boost (fast movement) along each axis (local to vehicle). These values are modified by the input script.
- **Min Movement Inputs**: The minimum input values for movement along each axis. For example, to limit reverse speed.
- **Max Movement Inputs**: The maximum input values for movement along each axis.
- **Default Movement Forces**: The forces for each axis that are multiplied by the *Movement Inputs* values and applied to the vehicle's Rigidbody to move it. These 'default' forces are used when not using Power Management in your game.
- **Default Steering Forces**: The turning forces (torques) for each axis that are multiplied by the *Steering Inputs* values and applied to the vehicle's Rigidbody to rotate it. These 'default' forces are used when not using Power Management in your game.
- **Default Boost Forces**: The forces for each axis that are multiplied by the Boost Inputs values and applied to the vehicle's Rigidbody to move it faster when 'boosting'.
- **Apply Movement Forces During Boost**: Whether to add the boost forces to the movement forces during boost (or just use the boost forces alone).
- **Max Movement Forces**: The maximum movement forces that can be applied to the vehicle's Rigidbody (regardless of the Default Movement Forces settings or any amount of power applied to the engines when using Power Management).

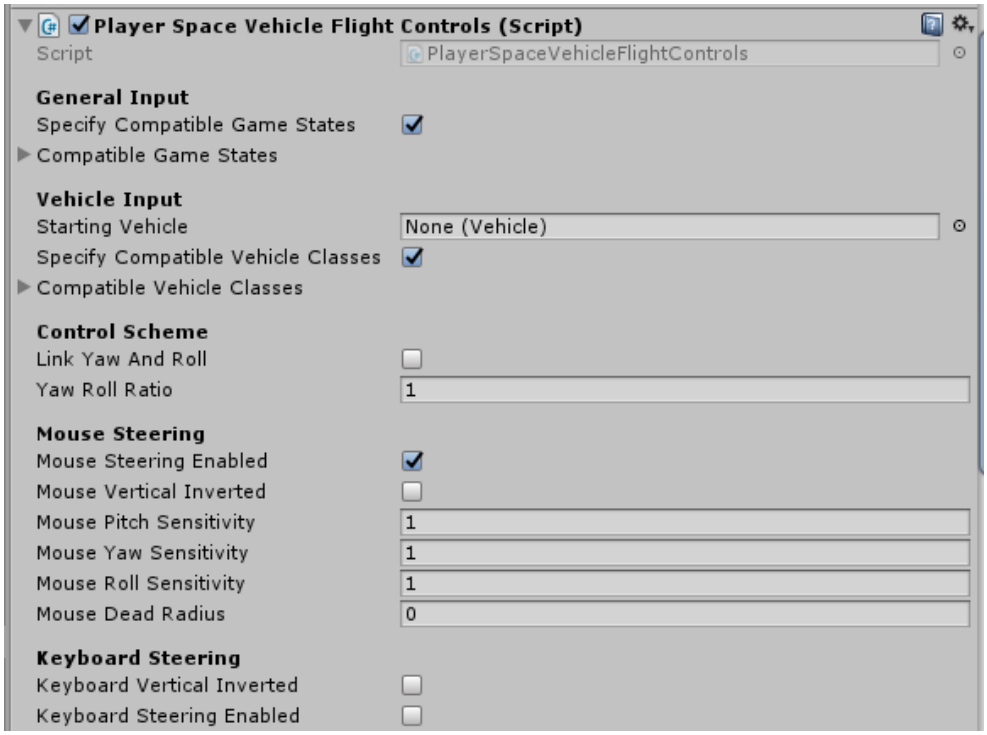
Setting Up The Input

Because the space fighter and capital ship are controlled in a different manner, they use different input scripts, even though both use the **VehicleEngines3D** component to steer and move around in the game world.

Space Fighter Controller Input

To set up the input for controlling a space fighter (or similar type of spaceship):

1. Create a new gameobject under the Player's Game Agent component.
2. Add a **PlayerSpaceVehicleFlightControls** component to it.



Inspector Settings:

- See the Vehicle Input section for the *General* and *Vehicle Input* settings.
- **Link Yaw And Roll**: Apply Roll (rotation around local Z axis) as a function of the Yaw (rotation around local Y axis). Enables 3-axis vehicle control using 2 inputs. Ship's nose will trace out a circle when rolling (rather than rolling on a point). The game Starlancer uses this method.
- **Yaw Roll Ratio**: The amount that the ship yaws (rotates around the Y axis) when rolling (rotating around the local Z axis).
- **Mouse Steering Enabled**: Enables pitch (local X axis rotation) and yaw (local Y axis rotation) to be controlled by the position of the mouse cursor on the screen.
- **Mouse Vertical Inverted**: Invert the pitch (nose up/down) of the ship based on the mouse cursor position.
- **Mouse Pitch Sensitivity**: How quickly the ship reaches full rotation speed around the local X axis based on the vertical position of the mouse from the screen center.
- **Mouse Yaw Sensitivity**: How quickly the ship reaches full rotation speed around the local Y axis based on the horizontal position of the mouse from the screen center.
- **Mouse Roll Sensitivity**: How quickly the ship reaches full rotation speed around the local Z axis based on the vertical position of the mouse from the screen center. This is used when Linked Yaw And Roll is set to True, as the mouse controls the Roll, and the Yaw is applied as a function of Roll.
- **Mouse Dead Radius**: The radius of the area around the screen center where the mouse position does not affect the steering of the vehicle. This value is the fraction of the viewport width (which goes from 0-1).
- **Keyboard Vertical Inverted**: Whether the pitch (local X axis rotation) is inverted when using keyboard controls.
- **Keyboard Steering Enabled**: Whether keyboard input will steer the vehicle.
- **Pitch Axis Input**: Input settings for controlling the pitch (local X axis rotation).
- **Yaw Axis Input**: Input settings for controlling the yaw (local Y axis rotation).

- **Roll Axis Input**: Input settings for controlling the roll (local Z axis rotation).
- **Throttle Up Input**: Input settings for increasing throttle.
- **Throttle Down Input**: Input settings for decreasing throttle.
- **Throttle Sensitivity**: How fast the throttle moves when throttling up or down.
- **Set Throttle**: Whether to set the throttle value using the Throttle Axis Input value;
- **Throttle Axis Input**: Input settings for setting the throttle using an Input Axis.
- **Strafe Vertical Input**: Input settings for strafing vertically (moving up and down while pointing forward).
- **Strafe Horizontal Input**: Input settings for strafing horizontally (moving left and right while pointing forward).
- **Boost Input**: Input settings for boosting (fast movement).

Setting Up The Capital Ship Input

To set up the input for controlling a capital ship:

1. Create a new gameobject under the Player's Game Agent component.
2. Add a **PlayerCapitalShipFlightControls** component to it.

Inspector Settings:

- See the Vehicle Input section for the *General* and *Vehicle Input* settings.
- **Yaw Input**: Input settings for controlling the yaw (local Y axis rotation).
- **Throttle Sensitivity**: How fast the throttle moves when throttling up or down.
- **Throttle Up Input**: Input settings for increasing throttle.
- **Throttle Down Input**: Input settings for decreasing throttle.

- **Strafe Vertical Input**: Input settings for strafing vertically (moving up and down while pointing forward).
- **Strafe Horizontal Input**: Input settings for strafing horizontally (moving left and right while pointing forward).
- **Boost Input**: Input settings for boosting (fast movement).
- **Ship PID Controller**: The PID controller for correcting the pitch and roll of the ship if it is knocked out of the horizontal plane.

Setting Up The Weapons

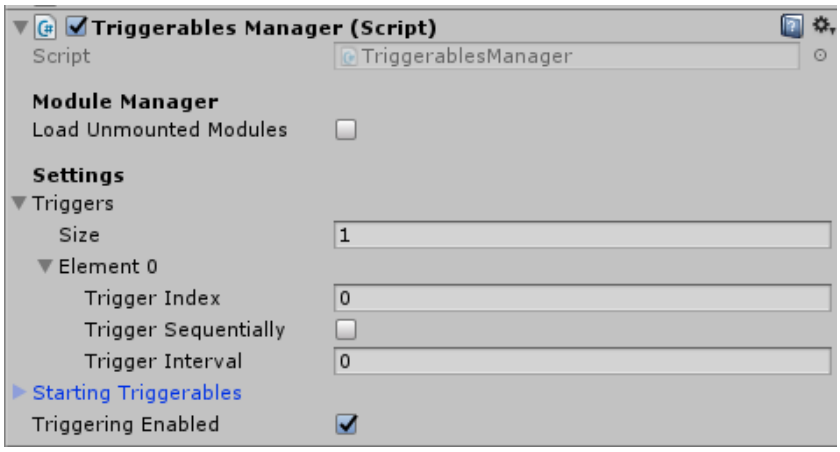
The way that weapons are set up in this kit is using a Triggerables system. The **TriggerablesManager** component stores all the **Triggerable** components found on the modules loaded onto the vehicle, each of which have a trigger index (integer) assigned in the inspector.

On the **PlayerTriggerablesInput** script, you can set up inputs (buttons) and assign them to a trigger index as well. When the input is pressed, all the Triggerable modules assigned to that trigger index are triggered.

Add A Triggerables Manager

To enable weapons or any other triggerables to be fired, add a **TriggerablesManager** component to the root transform of the vehicle.

The **TriggerablesManager** component enables the player or AI to fire triggerable modules such as weapons (see *Creating A Triggerable Module* below) that are loaded on the vehicle.



Inspector Settings:

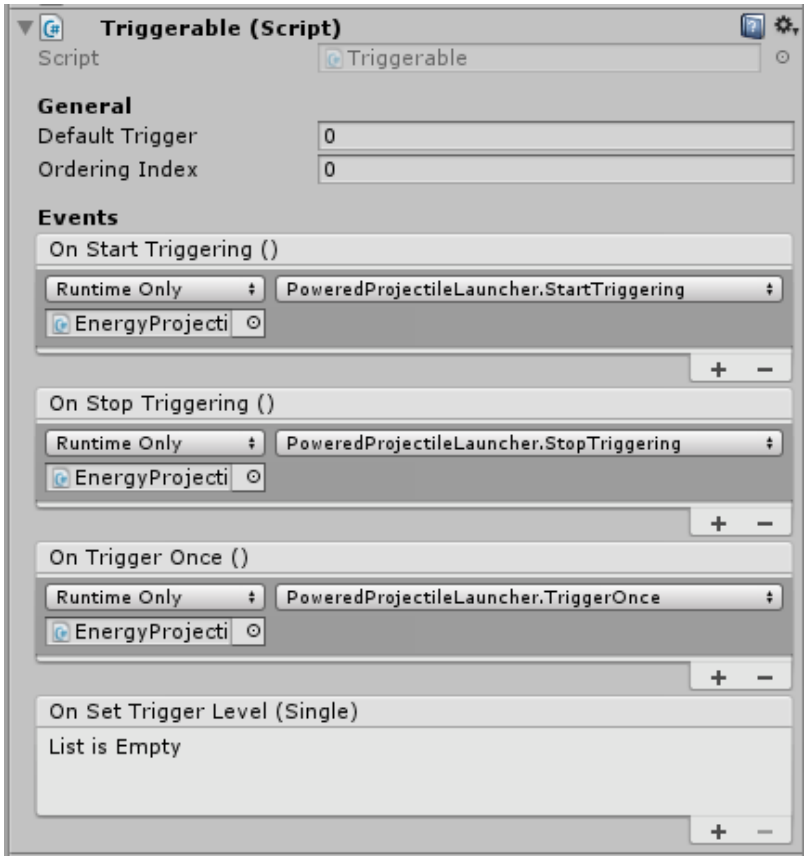
- **Load Unmounted Modules**: Whether to manage Triggerable modules that are part of the vehicle's hierarchy without being mounted at a module mount.
- **Triggers**: An array of Triggers, which are containers of data that describe how to fire Triggerable modules at a specific Trigger Index.
  - **Trigger Index**: The trigger index that this Trigger is for.
  - **Trigger Sequentially**: Whether to trigger the weapons at the trigger index one after the other (rather than all at the same time).
  - **Trigger Interval**: The interval between firing each weapon at the Trigger Index.
- **Starting Triggerables**: Any Triggerable components on the vehicle that are not part of any module.

- **Triggering Enabled**: Enable/disable the triggering of triggerable modules on the vehicle.

Creating Triggerable Modules

See the Modules section for more information on how to create a module.

A triggerable module is a module that has a Triggerable component on the root transform. The Triggerable component enables you to create any kind of triggerable item in your game, and provides Unity Events that you can use to call functions in your own scripts, or functions on scripts provided in the kit.



Inspector Settings:

- **Default Trigger**: The default trigger index for this triggerable module (may be changed in the Triggerables Manager).
- **Ordering Index**: Enables the ability to order the triggerable in any menu or list.
- **On Start Triggering**: Unity Event to call functions when triggering starts (when the *StartTriggering* method is called).
- **On Stop Triggering**: Unity Event to call functions when triggering stops (when the *StopTriggering* method is called).
- **On Trigger Once**: Unity Event to call functions when this Triggerable is triggered once.
- **On Set Trigger Level**: Unity Event to call functions when the Trigger Level is set (enables precise control of a weapon that can go anywhere between 0-1, not just on or off).

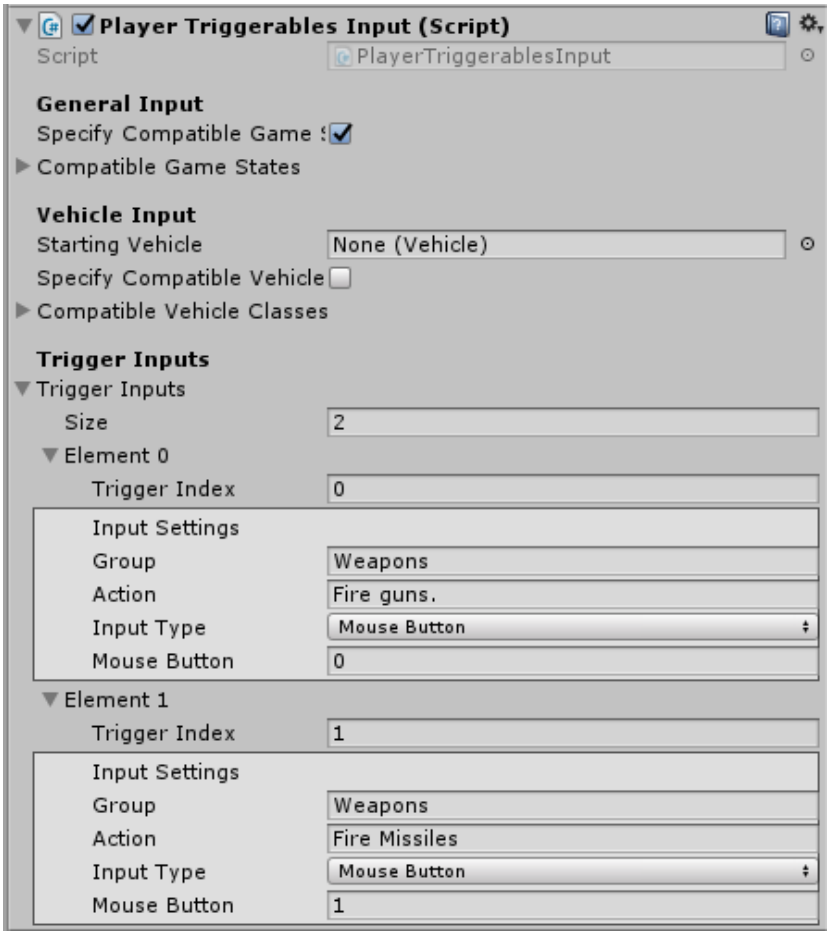
Loading A Weapon Module

See the Modules section for how to load a module onto a vehicle.

Setting Up The Input

To create input to fire triggerable modules (e.g. weapons) on a vehicle:

1. Add a new gameobject as a child of the Player's **GameAgent** component.
2. Add a **PlayerTriggerablesInput** component to it.
3. Customize the controls in the inspector.



Inspector Settings:

- See the Input section for the *General Input* and *Vehicle Input* base class settings.
- **Trigger Inputs**: An array of inputs that each fire the triggerable modules on a specified *Trigger Index*.
  - **Trigger Index**: The trigger index that this input will trigger (see above Creating Triggerable Modules for how to assign a triggerable module to a trigger index).
  - **Input Settings**: The input for firing the specified trigger index.

Set Up The Radar

Add a Trackables Scene Manager

Add a TrackablesSceneManager component anywhere in the scene.

This component is necessary to enable efficient tracking of targets in the scene, and is required for any target tracking to occur.

Add a Tracker To The Vehicle

To add target tracking capability to a vehicle.

1. Add a Tracker component to the root transform of the vehicle.
2. Customise the tracking parameters in the inspector.

That's all that is necessary to begin tracking targets in the scene. The targets tracked by the vehicle are stored in the *Targets* list on the Tracker component.

However, the targets are not yet visualized on the HUD (which includes target boxes, 3D holographic radar, and target hologram). That will be set up in the next section.

First, let's enable the player or AI to select targets from the Targets list.

Add Target Selectors To The Vehicle

To select a target:

1. Add a **TrackerTargetSelector** component anywhere on the vehicle.
2. Drag the **Tracker** component you previously added into the *Tracker* field in the inspector.
3. Customize the selection parameters in the inspector.

The selected target can be accessed via code in the *SelectedTarget* property of the TrackerTargetSelector component.

Multiple target selectors can be added to a vehicle as needed.

Link Weapons To Selected Target

To link weapons on a vehicle to the selected target:

1. Open the inspector of the Weapons component on the root transform of the vehicle.
2. Drag the TrackerTargetSelector component you just added into the *Weapons Target Selector* field.

This enables the guns to perform lead target calculation on the selected target, and the missiles to lock onto the selected target. Only one target selector can be linked to the weapons at any time.

Set Up Health

Making a Vehicle Damageble

To make a vehicle damageable:

1. Add one or more **Damageable** components anywhere in the hierarchy.

2. Customize the values in the inspector.

Receiving Damage

To receive damage, for every collider gameobject that represents something damageable:

1. Add a **DamageReceiver** component.
2. Open the inspector and drag the **Damageable** component into the *Damageable* field.

Now, when a hit occurs on one of the colliders, the DamageReceiver component on that collider's gameobject will send damage information to its assigned Damageable component.

Receiving Collision Damage

To receive collision damage on a damageable object (such as a vehicle):

1. Add a **Health** component to the root transform of the damageable object.
2. Make sure the damageable object has a Rigidbody on the root transform.

When a collision occurs, the **Health** component will receive it in its OnCollisionEnter function, and pass the damage to the **DamageReceiver** on the collider involved, which then passes it to the **Damageable**.

Causing Damage

To cause damage to a damageable object through one of its colliders:

1. Get the **DamageReceiver** component that is on the collider.
2. Call its *Damage* function with the necessary damage parameters.

This kit provides a **DamageEmitter** component that serves as a customizable projectile, beam and area damage emitter for all kinds of weapons. It will be discussed in detail in later sections.

Destruction of Damageable Objects

To enable a vehicle to be destroyed:

1. Add the vehicle's Destroy function to the Damageable's OnDestroyed event.

To deactivate the damageable object's gameobject when it is destroyed:

1. Open the inspector of the Vehicle component on the root transform and add a new entry to the OnDestroyed event.
2. Drag the vehicle's root transform from the Hierarchy view into the object field of the new event entry.
3. Select GameObject -> SetActive from the function popup menu and leave the checkbox that appears unchecked.

You can add other functions to the event as desired. There is an **ExplosionGenerator** component for creating an explosion via event.

NOTE: for damageable objects that are not vehicles, simply use the events in the **Damageable** component to drive what happens when it is destroyed.

Set Up HUD Target Boxes

Before proceeding, make sure you have set up your radar by following the steps in the 'Set Up The Radar' section.

To enable any part of the HUD to work, add a HUDManager component to the root transform of the vehicle.

The first steps for setting up target boxes for a vehicle:

1. Create a new gameobject as a child of your vehicle.
2. Add a HUDTargetBoxes component (which will add canvas components automatically) and open the inspector.
3. Add the Tracker component you added in the 'Set Up The Radar' section to the Trackers list.
4. Add any TrackerTargetSelector components you added in the 'Set Up The Radar' section to the TargetSelectors list.
5. In the TargetBoxContainers list, add a new entry by adding 1 to to the Size.
6. Under the new entry you just added, set the Prefab to the target box you wish to display for your target. Use the TargetBoxGeneric prefab provided if you haven't created your own yet.
7. Under the new entry you just added, add the type(s) of target(s) you wish to track to the TrackableTypes list.

Run the game. You will see a target box appear around your target (if it is on screen) and a directional arrow appear at the edge of the screen (if it is off screen).

Creating a Target Box

To create a target box, create a new gameobject and add a HUDTargetBox script.

Importing Into An Existing Project

If you're looking to import the Space Combat Kit into an existing project, you might not want to overwrite the Project Settings. These steps will ensure that you import the SCK correctly into an existing project without affecting existing project settings.

1. First create a new empty project and install SCK from the asset store, including project settings.

2. Export the 'Space Combat Kit' folder as a Unity Package (right click -> Export Package..)
3. Import the Unity Package into your existing project.
4. In your existing project and add the following:

Input Axes:

- PowerBallMoveHorizontal
- PowerBallMoveVertical
- Roll
- Strafe Vertical
- Strafe Horizontal

Layers:

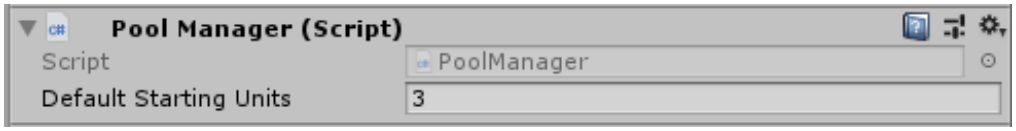
- UIManagedFOV
- UINoFOV

Object Pooling

Setting Up The Pool Manager

The first thing to do to set up pooling in your scene is to add a Pool Manager component, anywhere in the scene.

The Pool Manager component is a singleton, meaning there should only be one of them in your scene. It provides a reference point for accessing pooled objects, and also stores information about all object pools in the scene.

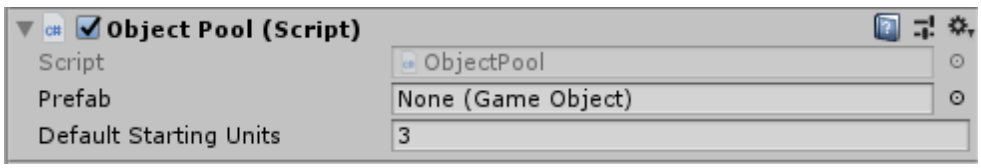


Creating An Object Pool

Next, you may want to create an object pool in your scene for a specific object. Doing so gives you the ability to set parameters for the object pool to help you manage the way objects are created and used from within them.

It is not necessary to create an object pool for every item in your game, as a new one will be created (if one doesn't exist already) when you get a pooled object (see next section below).

To create an object pool, simply add an Object Pool component to your scene.



Settings

- **Prefab**: This is a reference to the prefab of the object you are pooling.

- **Default Starting Units**: This is the number of units that are created immediately after the scene starts, which helps prevent performance issues when a pool starts being used

Getting A Pooled Object

Pooled objects are accessed through the Pool Manager, which is a singleton and can be accessed via 'PoolManager.Instance'.

There are several different ways you can get a pooled object in code. All of them return a GameObject reference to the object.

GameObject myPooledObject = PoolManager.Instance.Get(prefab);

This code passes either a reference to the prefab itself (GameObject), or the name of a prefab in a Resources folder in the project (string), and receives a pooled object at position (0,0,0) with no rotation and no parent.

GameObject myPooledObject = PoolManager.Instance.Get(prefab, position, rotation);

This code passes either a reference to the prefab itself (GameObject), or the name of a prefab in a Resources folder in the project (string), and receives a pooled object at the provided position (Vector3) and rotation (Quaternion).

GameObject myPooledObject = PoolManager.Instance.Get(prefab, position, rotation, parentTransform);

This code passes either a reference to the prefab itself (GameObject), or the name of a prefab in a Resources folder in the project (string), and receives a pooled object at the provided (world space) position (Vector3) and rotation (Quaternion), and parents it to the parentTransform (Transform).

Returning An Object To The Pool

When you don't need a pooled object anymore, you can return it to the pool simply by deactivating the gameobject.

This can be done by calling the code:

pooledObject.SetActive(false);