Space Combat Kit User Guide

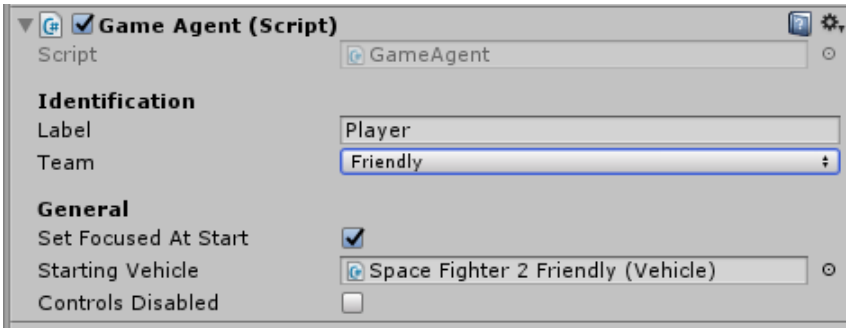
VSXGames

Contents

1. Game Agents

1.1 The Game Agent Component

A Game Agent is a component that represents a player or AI, and is separate from any kind of vehicle. This makes it possible for the player to maintain an identity while switching between vehicles in a game, and for the camera to maintain focus on a single player or AI through transitions between vehicles.



Label: The name of the player/AI that appears on the HUD when the game agent's vehicle is being targeted.

Team: The team this Game Agent belongs to.

Set Focused At Start: Whether this game agent should be focused on by the scene level components (such as the camera) when the game starts.

Starting Vehicle: the vehicle that this game agent should be in when the game starts.

Controls Disabled: Whether the game agent should start with the controls disabled (e.g. for intro cutscene).

1.2 The GameAgentManager Component

The GameAgentManager component stores references to all the Game Agents in the scene for easy access. Also, it stores a reference to the Focused Game Agent, which is the game agent focused on by the scene level components such as the camera, HUD etc (usually the Player). The GameAgentManager component is a singleton, meaning that there is only ever one of them in the scene, and it can be accessed globally by any script without a reference by calling *GameAgentManager.Instance*.

1.3 Creating a new Game Agent

1) Create a new gameobject in the scene.
2) Add a GameAgent component to it.
3) If you have a vehicle prepared, drag it into the 'Starting Vehicle' field in the inspector of the Game Agent component.

1.4 Switching Vehicles During Gameplay

To have a Game Agent switch between vehicles during gameplay, call the **EnterVehicle** method on the Game Agent component, passing the new vehicle's Vehicle component (or null to simply exit the current vehicle).

1.5 Adding Vehicle Input Scripts to a Game Agent

To enable a Game Agent to control a vehicle, it is necessary to add one or more vehicle input scripts as children of the transform that has the Game Agent component attached.

A vehicle input script is one which implements the IVehicleInput interface, and the kit provides the example **PlayerSpaceFighterControl** component (for the player to control a space fighter) and the **AISpaceFighterControl** component (for the AI to control a space fighter).

To make a new vehicle input script for a specific vehicle, it is necessary for the VehicleControlClass enum value on the input script and on the Vehicle component to match. When the GameAgent enters a new vehicle, it searches all its input scripts for one that has the VehicleControlClass value of the new vehicle, and uses the first one that is found.

2. Vehicles

2.1 What is a Vehicle

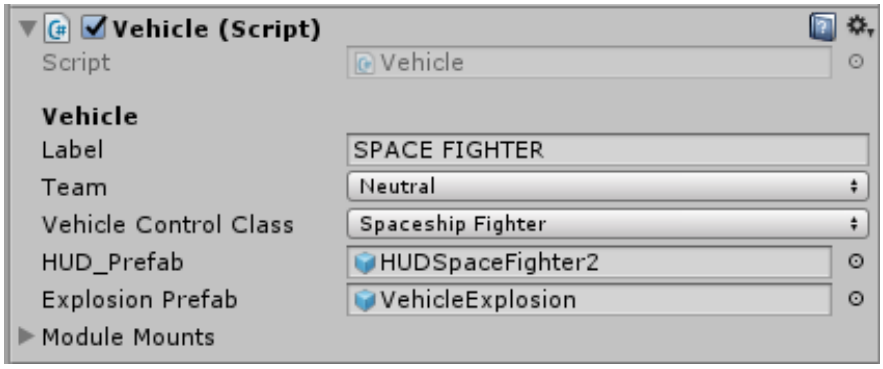
*NOTE: To see an example of a Vehicle that has been set up, look at the **Space Fighter 2 Friendly** prefab that has been provided in the kit.*

A vehicle is conceptually anything that the represents a player moving around and interacting with other things in the game world. A vehicle has three layers of functionality.

- The *Vehicle component* that is attached to the root transform;
- The *Subsystem components* that are attached alongside the Vehicle component to the root transform.
- The *module components* that are attached to child transforms, and which can be added either in the editor or during gameplay.

2.1.1 The Vehicle Component

The Vehicle component is a 'container' for all the components that make up the vehicle. It provides an identity for the vehicle, as well as an entry point for access to different parts of the vehicle. For example, it contains references to all the different subsystems that a vehicle has, so that a script can quickly determine what type of functionality the vehicle is implementing. Anything that needs to be accessed that is part of a vehicle should be accessed through this component.



Label: The name of vehicle that appears in the vehicle selection part of the loadout menu.

Team: The team that the vehicle belongs to when nobody is using it (on radar, when the vehicle is in use, this value is overridden by the team that the Game Agent piloting it belongs to).

Vehicle Control Class: The type of vehicle that this is (for matching an input script when a game agent enters it)

HUD_Prefab: the prefab of the HUD that this vehicle uses (used by HUDSceneManager to load the appropriate HUD when the player enters the vehicle).

Explosion Prefab: The explosion prefab that is instantiated when the vehicle is destroyed.

Module Mounts: A list of all the module mounts added to this vehicle.

2.1.2 Subsystem Components

The subsystem components, which inherit from the Subsystem base class, operate as managers for the different subsystems of the vehicle, including Engines, Weapons, Health, Radar and so on. The essential role of a subsystem component is to:

- Store references to all the modules related to that subsystem.
- Represent the state of a subsystem as a function of the modules associated with it.

What a subsystem does specifically varies according to its type. For example, the Weapons component stores references to all the weapon modules loaded on the vehicle and exposes methods for firing them. The Health subsystem checks if all the health generator modules loaded on the ship are depleted, and if so, destroys the vehicle.

The rule of thumb for what to put in a subsystem component is anything that will not change regardless of what type of vehicle is using that subsystem. For example, radar functionality (not HUD functionality) will not change between any kind of vehicle, so the code for it goes in the Radar subsystem component. But the implementation of a weapon is vehicle and context specific, so it would go in a module.

2.1.3 Module Components

Module components implement very specific functionality that is not shared across all vehicles. Anything that you as the developer want to add to a specific type of vehicle, or which the player will load/unload onto the vehicle during gameplay, should go into a module. See the Modules and Module Mounts section for more details on creating and loading modules.

2.2 Creating a New Vehicle

To create a new vehicle, create a new gameobject in the scene, and add a Vehicle component. That's all you need to do for the player to be able to enter and exit it. However, because the new vehicle has no subsystems, it cannot do anything in the game. Further along in this guide, you will see

sections on subsystems you can add to your vehicle including Engines, Weapons, Health, Radar, and more.

3. Modules and Module Mounts

3.1 What is a Module

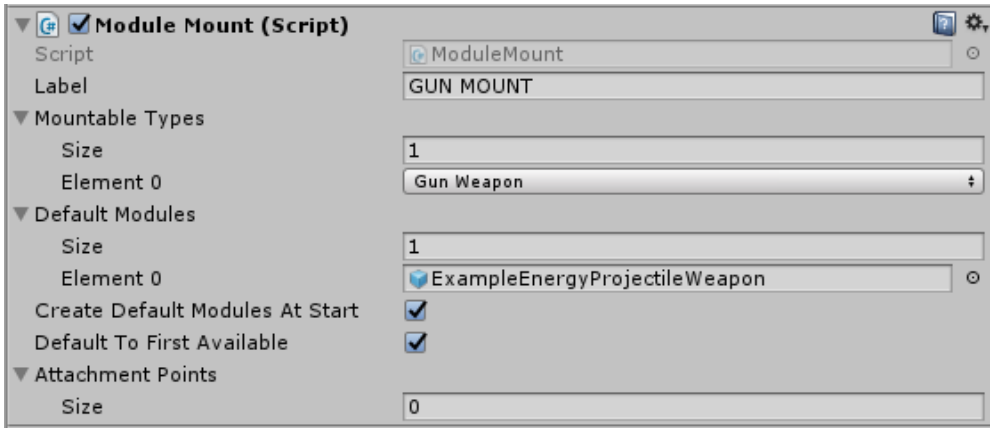
A module represents anything that you add to a vehicle to provide additional functionality in a specific area. Examples are weapons, powerplants, shield generators, and more.

Modules can be loaded, unloaded and cycled during gameplay, and provide the basis for a loadout menu, where the player can choose between different options for what to carry on their vehicle.

3.2 What is a Module Mount

A Module Mount simply represents a specific point on a vehicle where a module can be loaded.

- Multiple modules can be *created* on a module mount, but only one can be *loaded* at any time.
- Module Mounts can cycle modules during gameplay.
- Module Mounts can specify which types of modules can be loaded there, so that after the player chooses a loadout you can guarantee that a vehicle has certain types of modules attached.
- In the editor, you can set a list of Default Modules in the inspector of the Module Mount component with the option to create them and load the first available option as soon as the game starts.
- The Loadout scene demonstrates how to set it up so that player can choose which modules to load at a module mount, according to the module mount specifications.



Label: The label that appears when this module mount is selected in the loadout menu.

Mountable Types: The types of modules that can be mounted on this module mount.

Default Modules: The module prefabs to be created at the start on this module mount.

Create Default Modules At Start: Whether the default modules should be created at the start.

Default To First Available: Whether the first available default module should be selected (not just created).

Attachment Points: Passed to module for managing placement of a single module with multiple attachment points (for example twin wing-mounted weapons that are defined as a single weapon).

3.3 Creating a New Module

To create a new module prefab, it must have a component on the root transform that implements the IModule interface. The IModule interface allows all modules, whether they are weapons, shield generators or anything else, to be created, loaded and unloaded by a Module Mount.

3.4 Creating a Module Mount

To create a module mount for your vehicle, create a child gameobject of the vehicle and add a Module Mount component to it.

Also, you must add the module mount to the Vehicle component's module mounts list, by dragging the newly created Module Mount onto the Module Mounts list in the inspector of the Vehicle.

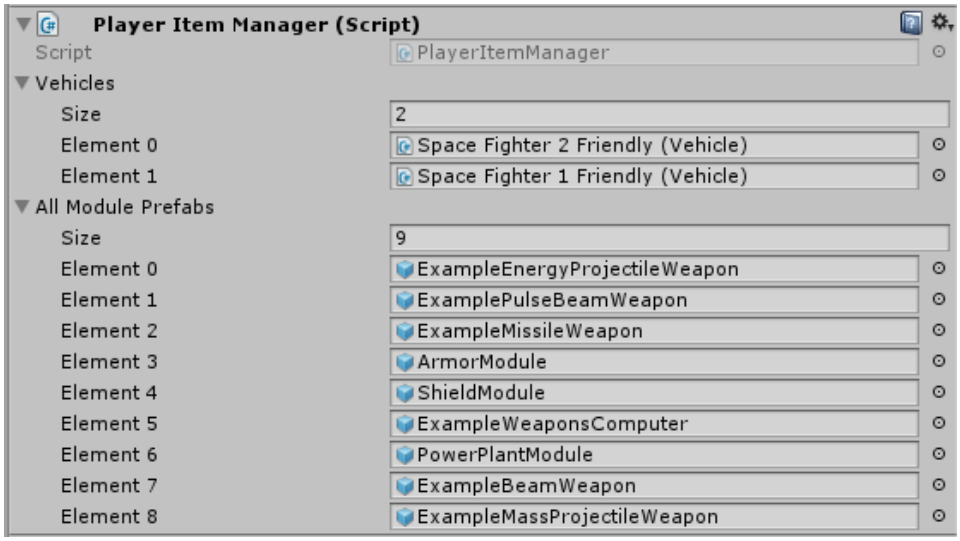
3.5 Adding Modules to the Loadout Menu

The loadout menu in the kit (in the *Loadout* scene) uses the PlayerItemsPrefab prefab to manage the ships and modules available in the loadout menu. This prefab (found in the Demos/Prefabs folder) has a PlayerItemManager script which contains references to all the ships and modules to be shown in the loadout menu.

To add a module to the Loadout menu:

1) Create a prefab for the module
2) Drag the PlayerItemsPrefab prefab into the scene

3) Drag the new module prefab into the All Module Prefabs list.
4) Apply the changes to the prefab.



Vehicles: The vehicles that can be selected on the loadout menu.

All Module Prefabs: The modules that can be selected in the loadout menu, depending on the vehicle.

4. Engines and Vehicle Controllers

4.1 The Engines Component

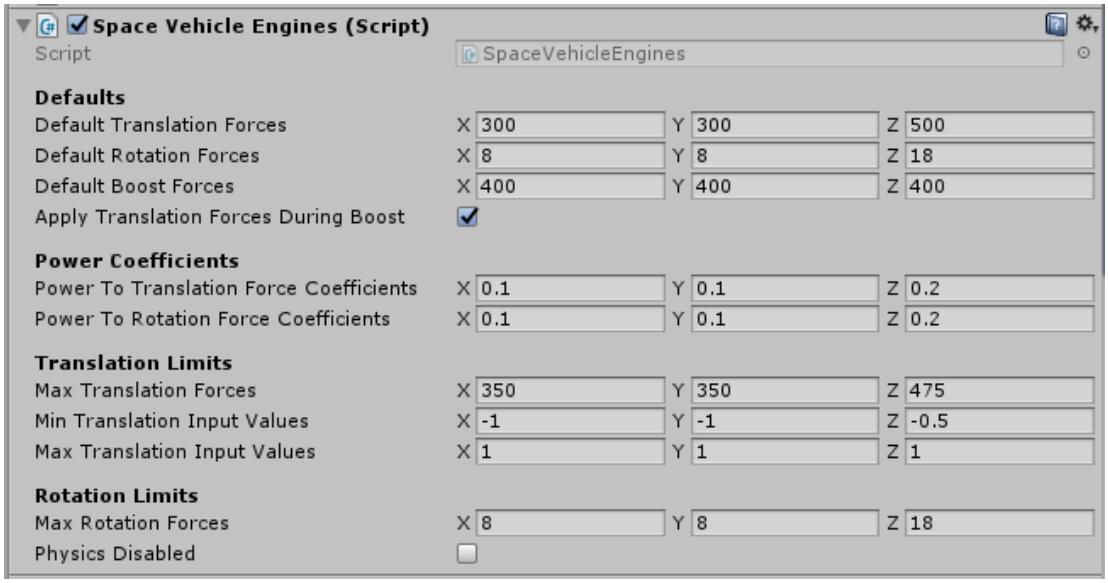
The Engines component is a Subsystem component that is basically a wrapper for the vehicle controller script (see below). It can be used to implement generalised code related to Engines across many different vehicles, but it does not contain any code for moving the vehicle around – that is what the vehicle controller script is for.

To add this subsystem, add an Engines component to the root transform of your vehicle.

4.2 Vehicle Controllers

A vehicle controller is a script that implements the IVehicleController interface and which contains code for controlling the movement of a vehicle. By using an interface, the vehicle controllers for many different vehicles can be accessed in a generalised way, such as for passing input values from control scripts.

The SpaceVehicleEngines script provided in the kit is an example of a vehicle controller for spaceships.


Space Vehicle Engines (Script)
Script
SpaceVehicleEngines
Defaults
Default Translation Forces
X 300 Y 300 Z 500
Default Rotation Forces
X 8 Y 8 Z 18
Default Boost Forces
X 400 Y 400 Z 400
Apply Translation Forces During Boost
Power Coefficients
Power To Translation Force Coefficients
X 0.1 Y 0.1 Z 0.2
Power To Rotation Force Coefficients
X 0.1 Y 0.1 Z 0.2
Translation Limits
Max Translation Forces
X 350 Y 350 Z 475
Min Translation Input Values
X -1 Y -1 Z -0.5
Max Translation Input Values
X 1 Y 1 Z 1
Rotation Limits
Max Rotation Forces
X 8 Y 8 Z 18
Physics Disabled

Default Translation Forces: The default forces that are multiplied by the translation input values (-1 to 1) along each axis to move the vehicle.

Default Rotation Forces: The default rotational forces (torques) that are multiplied by the rotation input values (-1 to 1) around each axis to move the vehicle.

Default Boost Forces: The default boost forces that are applied to the ship along each axis when boost mode is activated.

Apply Translation Forces During Boost: Whether full throttle is applied as well as boost forces when boost mode is activated.

Power To Translation Force Coefficients: The coefficients that are applied to the power available to the engines (in this case the Direct power) to obtain the maximum forces that can be applied to the vehicle. (Overrides the Default values above).

Power To Rotation Force Coefficients: The coefficients that are applied to the power available to the engines (in this case the Direct power) to obtain the maximum rotation forces (torques) that can be applied to the vehicle for steering. (Overrides the Default values above).

Max Translation Forces: The maximum force along each axis that can be applied to the vehicle, regardless of the amount of available power.

Min Translation Input Values: The minimum (i.e. the largest negative) translation input values, used to limit the translational speed along the negative x, y and z axes. In this case used to limit vehicle reversing speed.

Max Translation Input Values: The maximum translation input values, used to limit the translational speed along the positive x, y and z axes.

Max Rotation Forces: The maximum rotation forces (torques) that can be applied to the vehicle, regardless of the amount of available power.

4.3 Creating a New Vehicle Controller

To create a vehicle controller for a new type of vehicle, create the controller script and make sure it implements the IVehicleController interface. Then drag the script onto the root transform of the vehicle, alongside the Vehicle component.

5. Weapons

5.1 The Weapons Component

The Weapons component stores references to all weapons mounted on the vehicle, and exposes methods for an input script to start and stop firing weapons according to the weapon's trigger index.

The trigger index is an integer that represents a group that the weapon belongs to so that the group can all be fired simultaneously. This value can be set in the inspector of each weapon, or at runtime using the TriggerGroupsManager (see the Trigger Groups Manager section).

To add this subsystem, add a Weapons component to the root transform of your vehicle.

5.2 What is a Weapon Module

A weapon module is a module that fires a projectile or beam toward a target. In this kit, weapons are classified into Guns and Missiles. A Gun is a weapon whose projectile is unguided or 'dumb-fire' and requires lead target calculation, whereas a Missile is a guided weapon which requires target locking.

5.3 Creating a New Weapon

To create a new weapon, you must create a new module, with a component that implements the following interfaces:

- IModule (to be recognized as a loadable module)
- IWeaponModule (to be recognized as a weapon)
- IGunModule or IMissileModule (so that the weapon computer can implement lead target tracking or missile locking, respectively).

There are three example scripts provided that implement these interfaces, and which should cover most needs for projectile, beam and missile weapons:

- ProjectileWeapon.cs
- BeamWeapon.cs
- MissileWeapon.cs

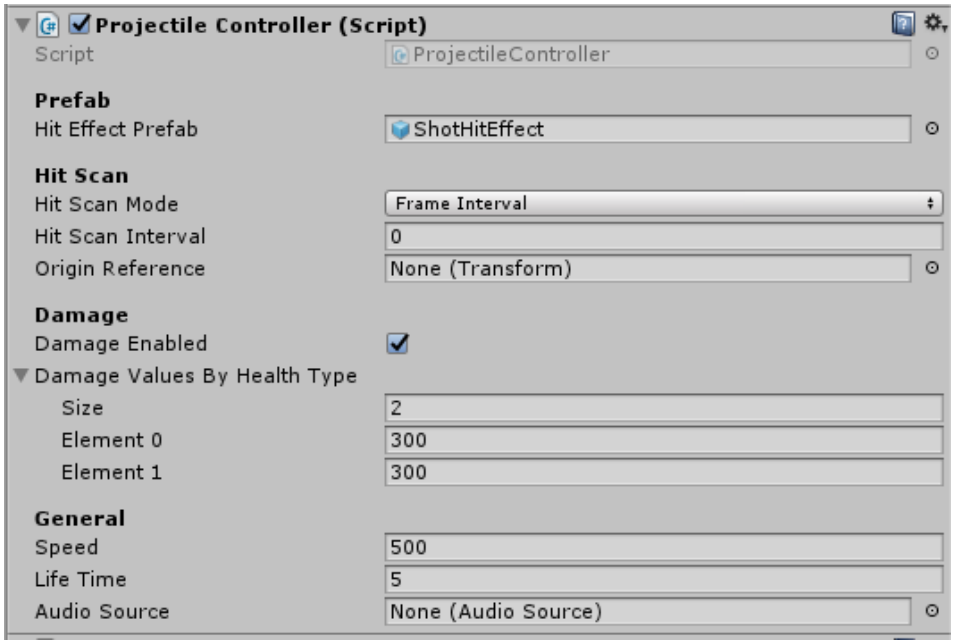
Using these scripts, the following prefabs are provided in the kit:

- Example Mass Projectile Weapon
- Example Energy Projectile Weapon
- Example Pulse Beam Weapon
- Example Beam Weapon
- Example Missile Weapon

Check out these weapons to see all the details of how you can set up your own.

5.4 Creating a New Projectile

To create a new projectile for a gun weapon, use the ProjectileController.cs script, which uses raycasts to determine a hit and can handle very fast projectiles.



Hit Effect Prefab: The prefab that is instantiated when this projectile hits something.

Hit Scan Mode: The hitscan mode – whether collision raycasts are made using a time interval or a frame interval.

Hit Scan Interval: The interval at which the collision raycasts are made (0 means every frame).

Origin Reference: The transform that represents the origin of the scene when a floating origin is being used (automatically set by an attached SceneOriginChild component).

Damage Enabled: Whether the projectile causes damage.

Damage Values By Health Type: The damage this projectile causes to any damageable object (IDamageable component) that it hits, according to the health type of the damageable object.

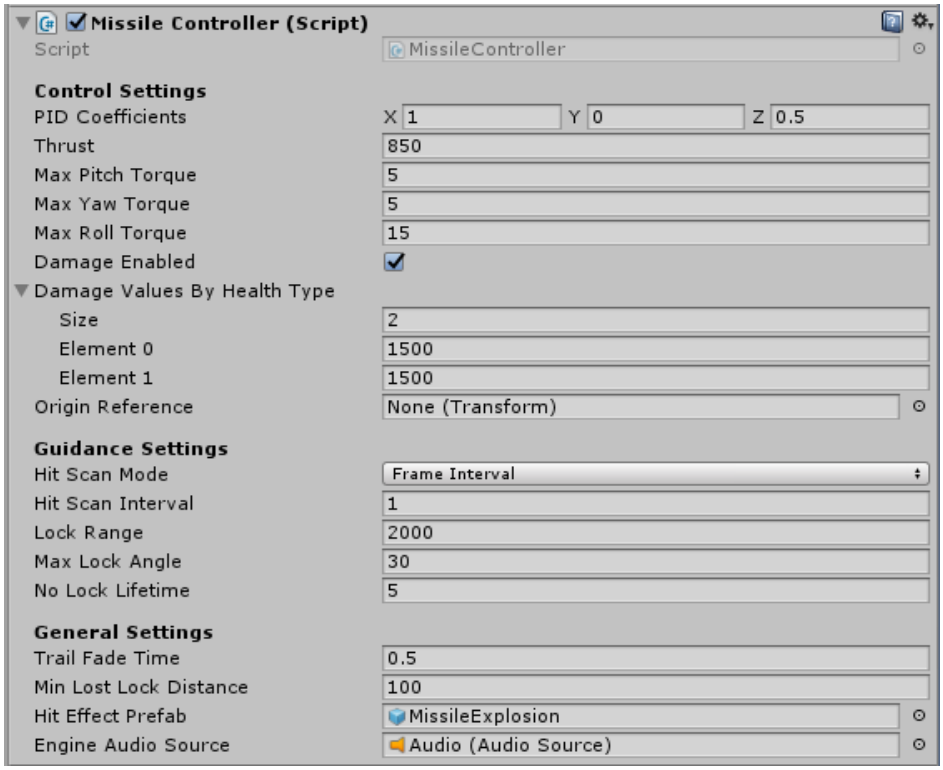
Speed: The constant speed of this projectile during its lifetime.

Lifetime: How long this projectile lasts before it is deactivated in the scene.

The ExampleEnergyProjectile prefab provided in the kit can be used as a starting point for creating your own bullets and projectiles.

5.5 Creating a New Missile

To create a new missile for a missile weapon, use the MissileController.cs script, which uses physics-based guidance to track and follow a target.



PID Coefficients: The steering coefficients for the missile's PID controller.

Thrust: The force used to propel the missile toward the target.

Max Pitch Torque: The maximum steering x-axis torque.

Max Yaw Torque: The maximum steering y-axis torque.

Max Roll Torque: The maximum steering z-axis torque.

Damage Enabled: Whether the missile causes damage.

Damage Values By Health Type: The damage this missile causes to any damageable object (IDamageable component), according to the damageable object's health type.

Origin Reference: The transform that represents the origin of the scene (automatically set by an attached SceneOriginChild component).

Hit Scan Mode: Whether collision raycasts are made by time or frame interval.

Hit Scan Interval: The collision raycast interval (0 means every frame).

Lock Range: The maximum locking range of this missile.

Max Lock Angle: The maximum angle to the target where the missile can stay locked.

No Lock Lifetime: How long after lock is lost that the missile self-destructs.

Trail Fade Time: How long it takes for the trail to fade after missile explodes.

Min Lost Lock Distance: Used to prevent losing lock due to angle when target is close.

Hit Effect Prefab: The explosion prefab that is instantiated when the missile explodes.

Engine Audio Source: The audio source for engine sound.

The ExampleMissile prefab provided in the kit can be used as a starting point for creating your own missiles.

5.6 Adding Weapons to a Vehicle

To add a new weapon to a vehicle, create a module mount (see Modules and Module Mounts section) on the vehicle and drag the prefab for your new weapon onto the Default Modules list in the inspector of the ModuleMount component.

Check the **Create Default Modules At Start** and **Default To First Available** checkboxes in the inspector of the ModuleMount component to have the weapon loaded and ready when the game starts.

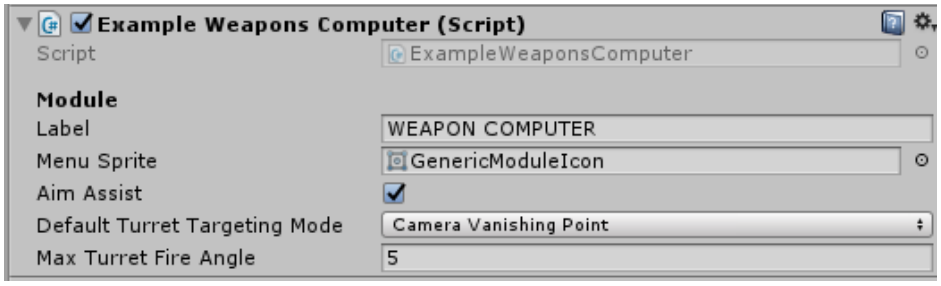
5.7 Adding a Weapon Computer

To have lead target calculation for guns and missile locking for missiles, it is necessary to add a weapon computer to the vehicle.

A weapon computer must implement the following interfaces:

- IModule (to be recognized as a module)
- IWeaponsComputer (to be recognized and loaded by the Weapons component)
- ILeadTargetComputer (so that the HUD can access lead target information from it)
- ILockingComputer (so that the HUD can access locking information from it)

The provided ExampleWeaponsComputer script shows you how to set up a script for a weapon computer, but you can create your own, for example to provide different classes of weapon computers in upgrade modules.



Label: The label of this module.

Menu Sprite: The sprite used for this module in the menu.

Aim Assist: Whether the weapons snap to target when within an angle range.

Default Turret Targeting Mode: The default targeting mode of turrets on the vehicle (automatically track targets, or track where the camera is pointing).

Max Turret Fire Angle: The angle range within which the turret will begin firing when set to automatic.

6. Health

In this kit, vehicle health basically consists of three different parts:

- The Health component;
- Health fixtures.
- Health generators.

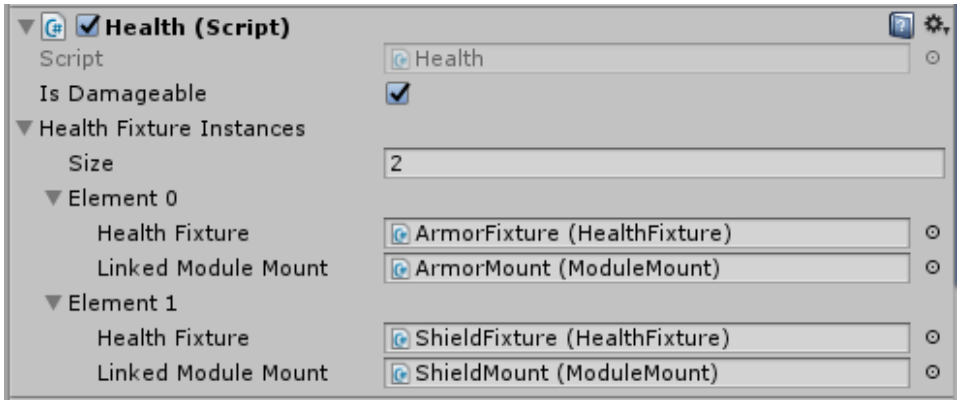
This setup makes it possible to equip different health generators in the loadout menu that offer different levels of damage protection for the player.

6.1 Health Component

The Health component is a subsystem component that adds health functionality to your vehicle.

- Stores references to all health fixtures on the vehicle.
- Stores references to all the health generator modules mounted on the vehicle.
- Links health fixtures with health generators.
- Manages embedded health fixtures (e.g. armor plating inside a shield).
- Sends an event when the vehicle is destroyed (no active health generators left).
- Expose health information for the HUD.

To add this subsystem, add a Health component to the root transform of your vehicle.



Is Damageable: Whether the vehicle is damageable.

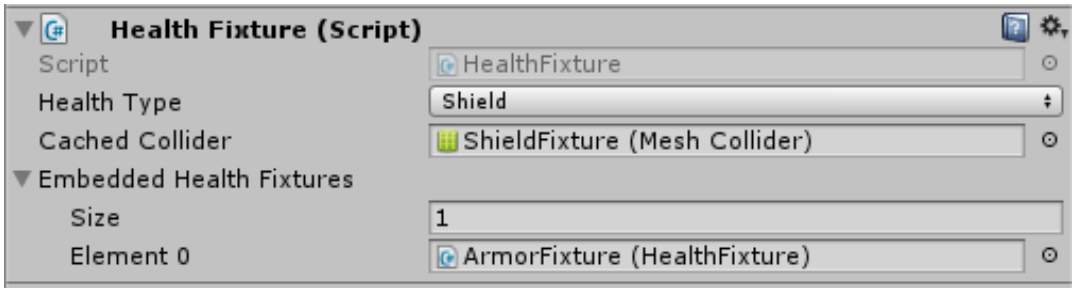
Health Fixture Instances: A list of all the health fixtures on the vehicle and their linked module mounts.

6.2 Health Fixtures

A Health Fixture basically represents a physical collider that can transmit damage and healing events to other components on the vehicle. It holds a reference to a delegate that other components (such as the HealthGenerator and the ShieldEffectsController components) can attach functions to, so that they can be alerted when a damage or heal event occurs.

A Health Fixture can be 'embedded' inside another one. This means that while the outer Health Fixture is still active, the 'embedded' Health fixture cannot receive any damage or healing events (the collider is disabled). In this way, you can implement, for example, a shield protecting the armored hull of a ship.

Each HealthFixture component, in the inspector, holds a list of references to all the other health fixtures that are 'embedded' inside it.



Health Type: The health type of this health fixture.

Cached Collider: The collider for this health fixture.

Embedded Health Fixtures: The health fixtures physically inside of this one, to be activated only when this one is destroyed.

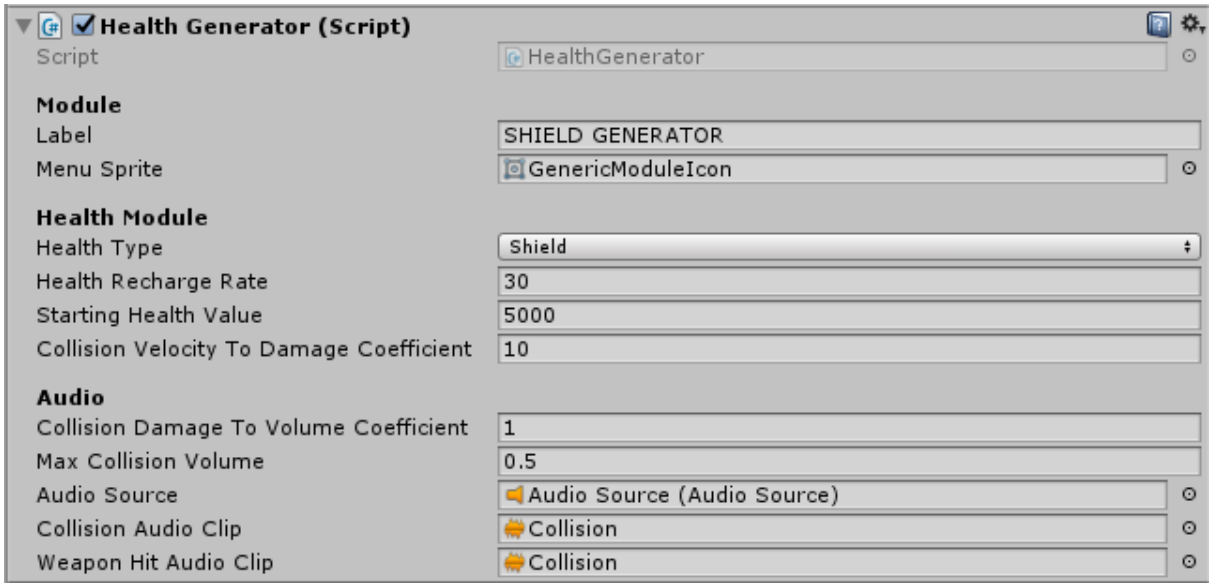
6.3 Creating a New Health Fixture

To create a new Health Fixture on a vehicle:

1. Create a child gameobject on the vehicle.
2. Attach a HealthFixture component.
3. Create a collider somewhere on the vehicle for this health fixture, and drag it into the Cached Collider field in the inspector of the HealthFixture component.

6.4 Health Generators

A health generator is a module that has a health value that can be decreased (due to damage) or increased (due to healing) within a specified range. A health generator module becomes destroyed when the health value reaches zero.



Label: The module label that appears in the loadout menu.

Menu Sprite: The sprite that appears in the loadout menu for this module.

Health Type: The health type of this health generator.

Health Recharge Rate: The recharge rate of the health value per second.

Starting Health Value: The starting health value of this health module.

Collision Velocity To Damage Coefficient: The number that is multiplied by the magnitude of the relative collision velocity to determine the damage value.

Collision Damage To Volume Coefficient: The number that is applied to the magnitude of the relative collision velocity to determine the volume of the impact sound effect.

Max Collision Volume: The maximum collision sound effect volume.

Audio Source: The damage sound effect audio source.

Collision Audio Clip: The audio clip to be played when a collision occurs.

WeaponHitAudioClip: The audio clip to be played when a weapon hit occurs.

6.5 Creating a New Health Generator

To create a new health generator module, create a prefab which has a component on the root transform that:

- Implements the IModule interface.
- Implements the IHealthGenerator interface.

The HealthGenerator.cs script provided in the kit is an example such a component and should cover most uses. It can be used as a template for implementing your own.

Note that the Health Type associated with a Health Generator (and Health Fixture) is an enum which can be found in the Health.cs script, and you can simply add values to it to create more health types in your game.

To associate a health generator module with a health fixture, link the module mount where the health generator module will be loaded to the health fixture in the inspector of the Health component. Then, when a health generator is loaded on the module mount, it will be linked to the health fixture (if the health types match).

See the Modules and Module Mounts section for more details on how to set up a module mount for your health generator module.

7. Radar

Radar target tracking in this package involves two main components:

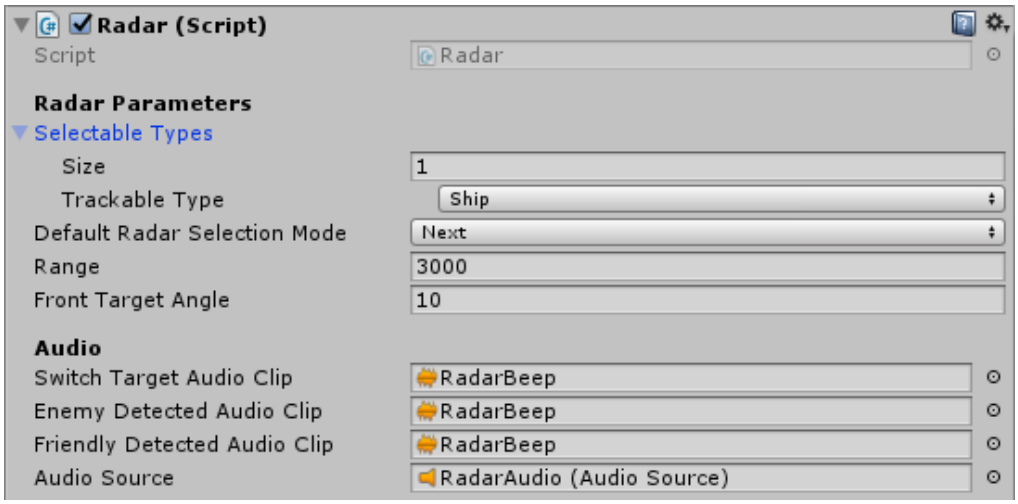
- The Radar component
- The Trackable component
- The RadarSceneManager component

7.1 The Radar Component

The Radar component is a vehicle subsystem component that enables the vehicle to track targets within a specified range and select a 'selected target'.

You can also specify which trackable types the radar is capable of tracking.

To add this subsystem, add a Radar component to the root transform of your vehicle.



Selectable Types: The trackable types that this radar can select as selected target.

Default Radar Selection Mode: The selection mode of the radar when it does not have a selected target.

Range: The tracking range of the radar.

Front Target Angle: The maximum angle for selecting a target using the Front selection mode.

Switch Target Audio Clip: The clip that is played when the selected target changes.

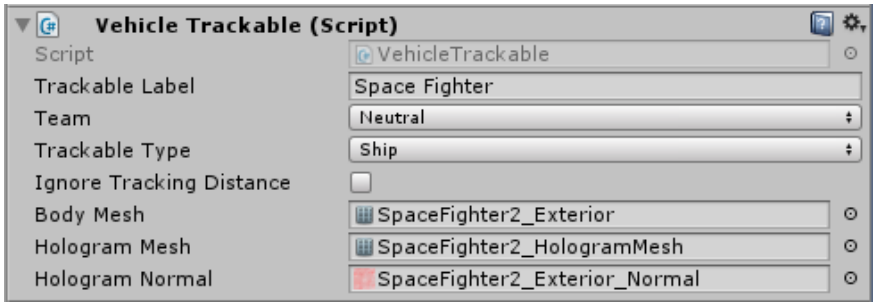
Enemy Detected Audio Clip: The clip that is played when an enemy is detected after no enemies have been present.

Friendly Detected Audio Clip: The clip that is played when a friendly is detected after no friendlies have been present.

Audio Source: The radar sound effect audio source.

7.2 The Trackable Component

The Trackable component designates an object to be trackable in the scene. The VehicleTrackable component provided in the kit inherits from Trackable, with the added functionality of passing the Game Agent information rather than the Vehicle information for tracking purposes when a Game Agent is in the vehicle.



Trackable Label: The label that appears when this trackable is being tracked

Team: The team of the empty vehicle (overridden by Game Agent when occupied).

Trackable Type: The trackable type of this trackable.

Ignore Tracking Distance: Whether distance is ignored when tracking this trackable.

Body Mesh: The mesh that is used for calculating size aspects of target tracking (e.g. expanding target boxes).

Hologram Mesh: The mesh that is displayed on the dashboard hologram when this trackable is the selected target.

Hologram Normal: The normal map that is applied to the dashboard hologram model when this is the selected target.

7.3 Creating a New Trackable Object

To create a new trackable object, add a Trackable component to the root transform.

7.4 The RadarSceneManager Component

The RadarSceneManager component is a singleton (meaning there should only ever be one of them in the scene) which registers all trackable objects and stores them in a way that makes it easy and efficient for the Radar component to access them.

To add a RadarSceneManager to your scene, add a new gameobject anywhere in your scene, and add a RadarSceneManager component to it.

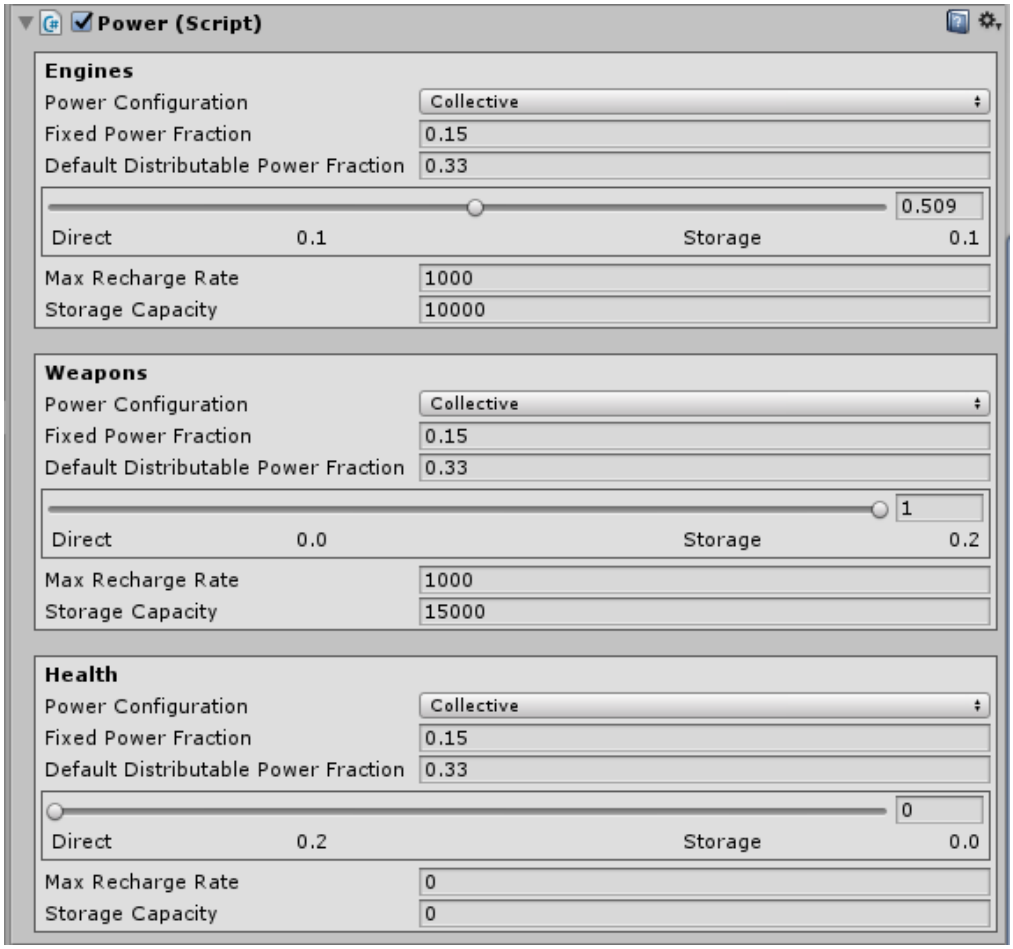
8. Power

This kit includes a way to create different types of powered systems for your vehicle. Whether you simply want to have a depleting energy bar for laser weapons, or full-on power management between engines, weapons and shields during gameplay, you can easily do it with this kit.

8.1 The Power Component

The Power component is a subsystem component that adds power to your vehicle. It is not necessary to use it if you don't want to have this in your game.

To add a power subsystem to a vehicle, add a Power component to the root transform of the vehicle.



Power Configuration: The power configuration for a powered subsystem.

Fixed Power Fraction: The fraction of the total collective power that is fixed to this subsystem (cannot be redirected).

Default Distributable Power Fraction: The fraction of the distributable power fraction (which is 1 – sum of fixed power fractions) that is directed to this subsystem by default, and can be changed at runtime.

Direct < > Storage: The amount of total available power that is available Direct (i.e. on-tap) vs the amount that goes to storage on the Power component.

Max Recharge Rate: The maximum recharge rate of the storage for this subsystem on the Power component.

Storage Capacity: The maximum storage capacity for this subsystem on the Power component.

8.1.1 Subsystem Power Configurations

In the inspector of the Power component, you can set the power of each subsystem of the vehicle to be:

- **Collective**: all subsystems set to this value draw power from a central power plant (which is loaded as a module).
- **Independent**: the subsystem has its own internal power source.

- **Unpowered**: The subsystem does not have any power functionality.

8.1.2 Direct and Stored Power

Additionally, on the Power component inspector, power for each subsystem can be provided in two different formats using a slider:

- **Direct**: The power is available directly each frame, and does not get stored if it is not used.
- **Storage**: The power is used to recharge the storage for the subsystem in the Power component and can be discharged from there.

8.1.3 Power Distribution

To manage power between subsystems that are powered collectively, you can set in the inspector of the Power component a value called Fixed Power Fraction for each subsystem, which is the fraction of the total collective power that is fixed to a subsystem and cannot be redirected to a different subsystem by the player during gameplay. This allows you to set a minimum value, for example, for the engines even when the player wants to divert all power to the shields.

Another value that can be set in the inspector for each subsystem is the Default Distributable Power Fraction, which is the fraction of distributable power (distributable power = total power – sum of fixed power fractions for all the subsystems) that is by default directed to this subsystem. This value can be changed by the player during runtime.

The GameplayDemo scene provided in the kit provides a power management menu for the player, using a triangle slider to distribute power between engines, weapons and shields.

8.1.4 Drawing Power

The Power component exposes functions for drawing power than you can call in your own scripts, as well as functions that get important values for displaying information on the HUD to the player. Check out the SCK code documentation online for more information.

8.2 Power Plants

A power plant is used by the Power component and determines the amount of power that is available to subsystems that are designated as Collectively powered. Power plants are implemented as modules so that you can, for example, create multiple power plants that the player can upgrade with throughout your game.

A power plant module must be a prefab that has a component on the root transform that implements:

- The IModule interface.
- The IPowerPlant interface.

For more information about creating and loading modules, see the Modules and Module Mounts section.

9. Trigger Groups

This kit provides the ability for the player to create and select between multiple trigger groups during gameplay.

9.1 The TriggerGroupsManager Component

To add trigger groups functionality to your vehicle, add a TriggerGroupsManager component to the root transform of your vehicle. This component exposes functions that can be called to add and remove trigger groups, select a trigger group, and bind triggerable modules on the vehicle to trigger indexes.

The GameplayDemo scene provided in the kit includes a trigger groups menu that allows the player to set up trigger groups during gameplay.

9.2 Creating a Triggerable Module

For a module to be part of trigger groups, the module prefab must have a component on the root transform that implements the ITriggerable interface. A triggerable module can be any module (for example a scanner) and is not limited to weapons.

For more information about creating and loading modules, see the Modules and Module Mounts section.

10. HUD (Head-Up-Display)

This kit includes a comprehensive spaceship HUD, including 3D radar, target tracking boxes, and dashboard target holograms.

First of all, to save memory and make it easier to implement screen-space HUD elements, there is **only one HUD loaded** in the game at any time, for the vehicle that the player is currently in.

10.1 Creating a New HUD Prefab

To create a new HUD prefab for a vehicle, create a new gameobject and add a HUDManager component to it. After saving the prefab, drag the prefab into the HUD Prefab field of the Vehicle component on the vehicle that the HUD is for. This means that when the player enters this vehicle, that prefab will be loaded into the game.

The basic rule when creating a HUD prefab is that the root transform should be at the same position and rotation as the vehicle's root transform. Then all parts of the HUD should be positioned as children in the appropriate position and rotation relative to that.

10.2 The HUDManager Component

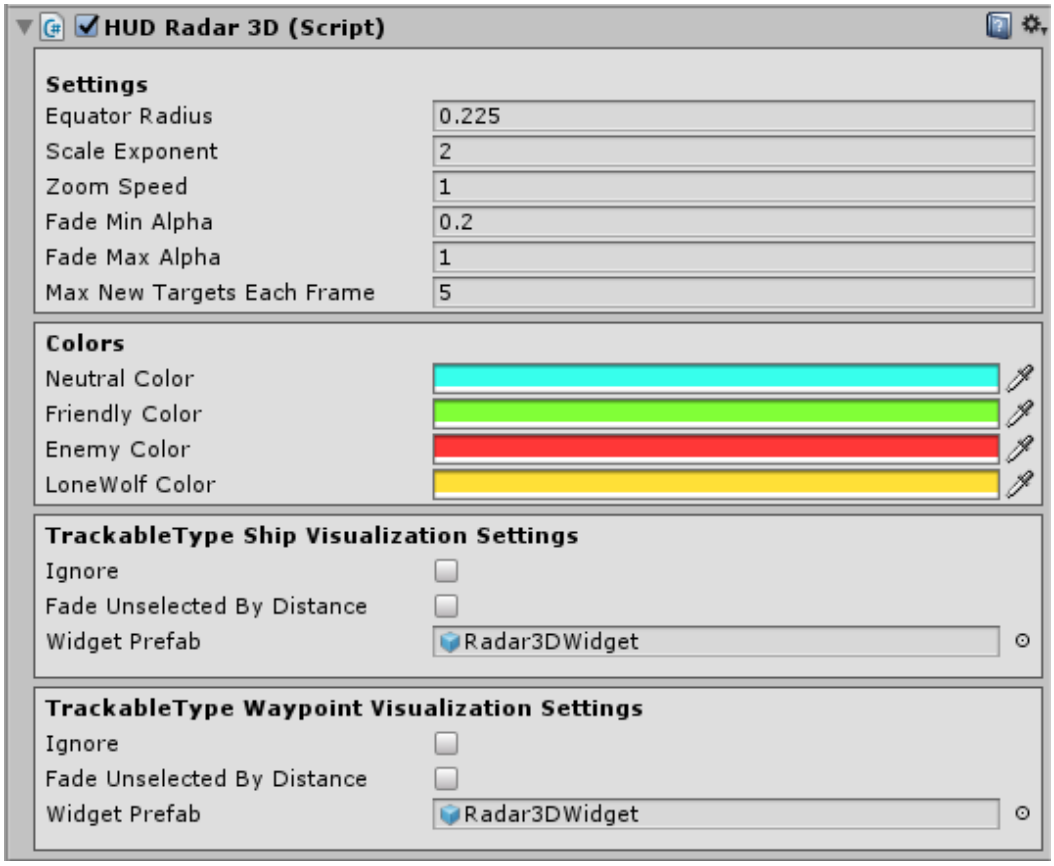
The HUDManager component, which is added to the root transform of every HUD prefab, manages the way that the parts of the HUD are arranged for different camera views, as well as providing a central source for each of the HUD components to access information about the vehicle.

Disable Hologram In Exterior View: Whether the hologram should be disabled in exterior view.

Disable 3D Radar In Exterior View: Whether the 3D radar should be disabled in exterior view.

10.3 Adding a 3D Radar

To add a 3D radar to your HUD prefab, it must have a child gameobject with a HUDRadar3D component. Look at the 3D radar on the HUD prefab already provided in the kit as a starting point for creating your own.



Equator Radius: The radius of the spherical viewing space.

Scale Exponent: The exponent used to scale the distance of the target widget from the center.

Zoom Speed: How fast the zoom changes when the zoom in or out button is pressed.

Fade Min Alpha: The minimum alpha of the radar widget when it is at the edge of tracking range.

Fade Max Alpha: The maximum alpha of the radar widget at zero distance from the tracker.

Max New Targets Each Frame: The maximum number of new target widgets that can be added each frame (for performance management).

Colors: The team colors for target widgets.

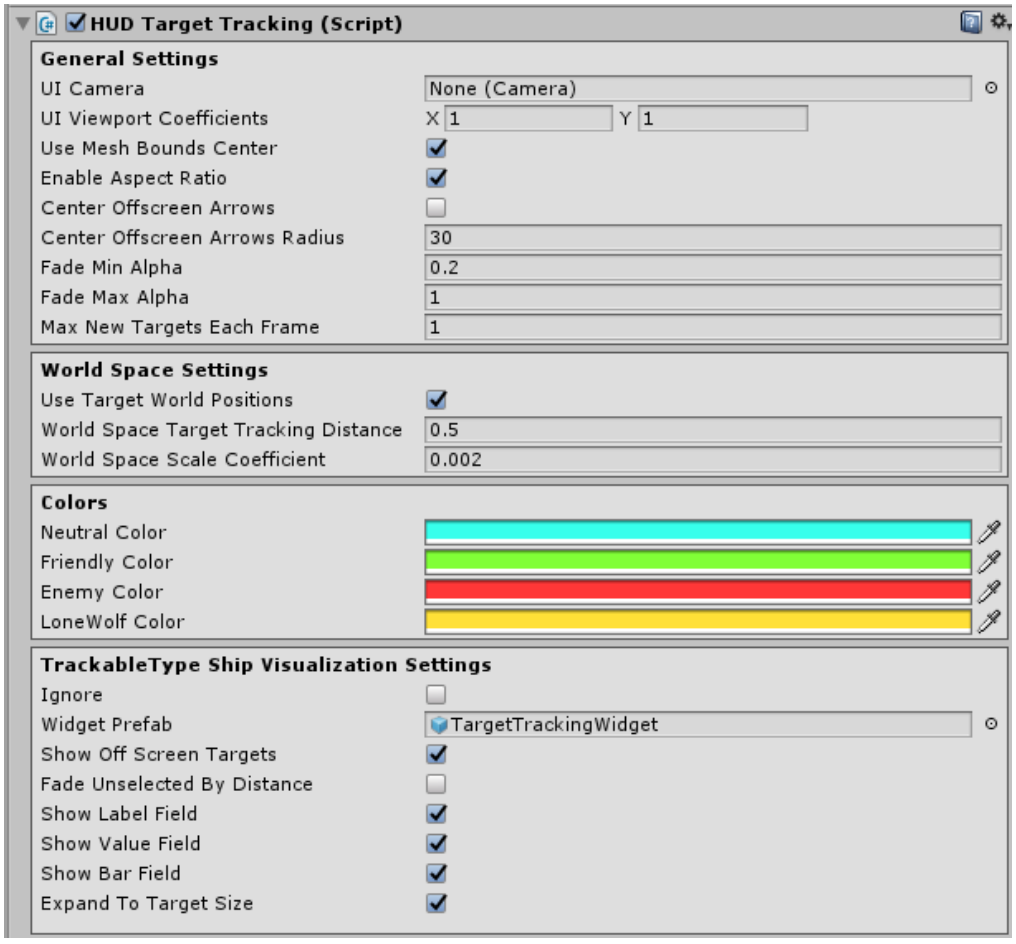
Ignore: Whether to ignore this trackable type on the 3D radar.

Fade Unselected By Distance: Whether to fade unselected targets by distance.

Widget Prefab: The widget prefab for this trackable type.

10.4 Adding Target Tracking

To add Target Tracking to your HUD prefab, it must have a child gameobject with a HUDTargetTracking component. Look at the target tracking on the HUD prefab already provided in the kit as a starting point for creating your own.



UI Camera: The camera used to calculate the target tracking UI (uses camera with MainCamera tag by default).

UI Viewport Coefficients: The amount of the screen width and height used to show the target boxes (useful for keeping everything in view when using VR).

Use Mesh Bounds Center: Whether to use the center of the target mesh bounds to center the target box (as opposed to the target's transform position).

Enable Aspect Ratio: Whether to adjust the aspect ratio of target boxes depending on the target's profile on the screen.

Center Offscreen Arrows: Whether the offscreen arrows should be displayed in a circular pattern around the screen center (as opposed to using the rectangular screen border).

Center Offscreen Arrows Radius: The radius of the offscreen arrows circular pattern (in Unity Units when in World Space, and in number of pixels when in Screen Space).

Fade Min Alpha: The alpha of a target box when the target is at the edge of tracking range.

Fade Max Alpha: The alpha of the target box when the target is at zero distance.

Max New Targets Each Frame: The maximum number of new target widgets that can be added each frame (for performance management).

Use Target World Positions: Whether the target boxes are displayed at the world position of the target (and scaled up) as opposed to projecting it a specified distance along the camera-to-target vector.

World Space Target Tracking Distance: The distance that the target boxes are projected along the camera-to-target vector (when **Use Target World Positions** is unchecked).

World Space Scale Coefficient: The coefficient multiplied to the distance of the target box from the camera to determine the target box scale.

Colors: The team colors for target widgets.

Ignore: Whether to ignore this trackable type on the HUD Target Tracking.

Widget Prefab: The widget prefab for this trackable type.

Show Offscreen Targets: Whether to show offscreen targets for this trackable type.

Fade Unselected By Distance: Whether to fade unselected targets of this type by distance.

Show Label Field: Whether to show the label of the target box for this trackable type.

Show Value Field: Whether to show the value field of the target box for this trackable type.

Show Bar Field: Whether to show the bar field of the target box for this trackable type.

Expand To Target Size: Whether the target box should expand to the on-screen target size for this trackable type.

10.4.1 Setting Up Target Tracking for VR

To set up the HUDTargetTracking for VR, go through the following steps:

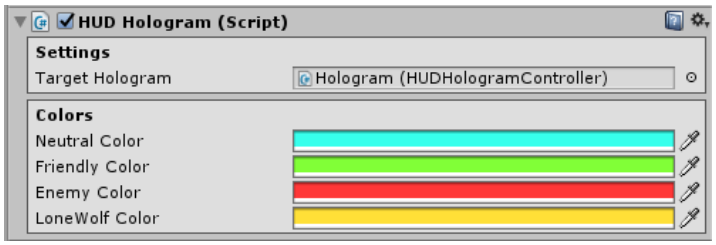
1. Set the Render Mode of the Canvas component on the HUDTargetTracking gameobject to World Space
2. Make sure the canvas scale is (1,1,1) – it often becomes changed after being in Screen Space.
3. Check the **World Space UI** checkbox in the inspector of the HUDTargetTracking component.

Adjust the properties under **World Space Settings** in the inspector of the HUDTargetTracking component according to your needs. **The UI Viewport Coefficients** setting is also useful for bringing in the off-screen arrows from the screen border where they are not visible in VR.

10.5 Adding a Target Hologram

To add a dashboard target hologram to your HUD prefab, it must have a child gameobject with a HUDHologram component. Look at the hologram on the HUD prefab already provided in the kit as a starting point for creating your own.

Note that the hologram mesh and hologram normal map for each target is provided to the HUD via that target's Trackable component (see the Radar section).



Target Hologram: The hologram controller for the selected target.

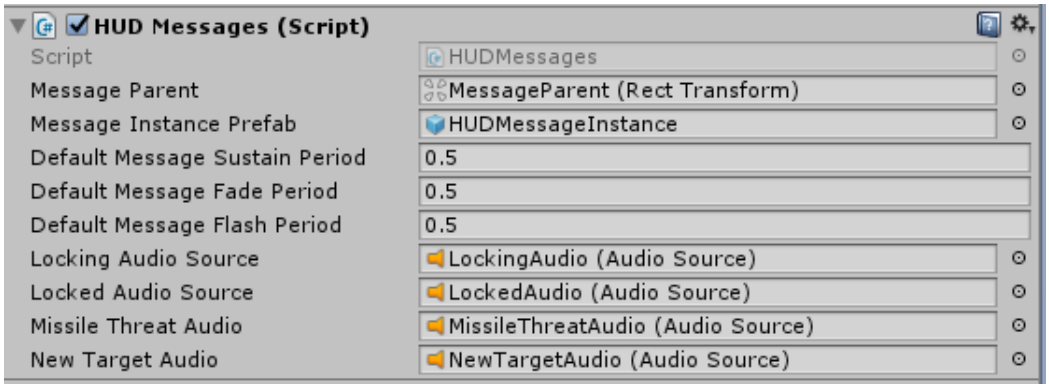
Colors: The team colors for the hologram.

10.6 Adding HUD Messages

HUD Messages is a component that allows you to show messages on the HUD that fade out and expire after a time limit, or flash on and off on the screen, by calling functions on this component via code.

The HUD Messages component also includes sound effects for certain vehicle functions.

To add HUD messages to your HUD prefab, it must have a child gameobject with a HUDMessages component.



Message Parent: The parent to instantiate the HUD message prefabs.

Message Instance Prefab: The prefab for an instance of a HUD message.

Default Message Sustain Period: The default amount of time the message is shown without fading.

Default Message Fade Period: The default amount of time taken for the message to fade.

Default Message Flash Period: The default amount of time taken for the message to flash on and off, when it is a flashing message.

Locking Audio Source: The audio source to be played when the weapon computer is locking onto a target.

Locked Audio Source: The audio source to be played when the weapon computer has locked onto a target.

Missile Threat Audio: The audio to be played when there is a missile threat.

New Target Audio: The audio to be played when there is a new target.

10.7 The HUDSceneManager Component

To manage the loading and unloading of HUDs in the game, there is a component called the HUDSceneManager. This component is a singleton, meaning there should only ever be one of them in the scene.

To add a HUDSceneManager component to your scene, create a new gameobject and add a HUDSceneManager component to it.

11. Input

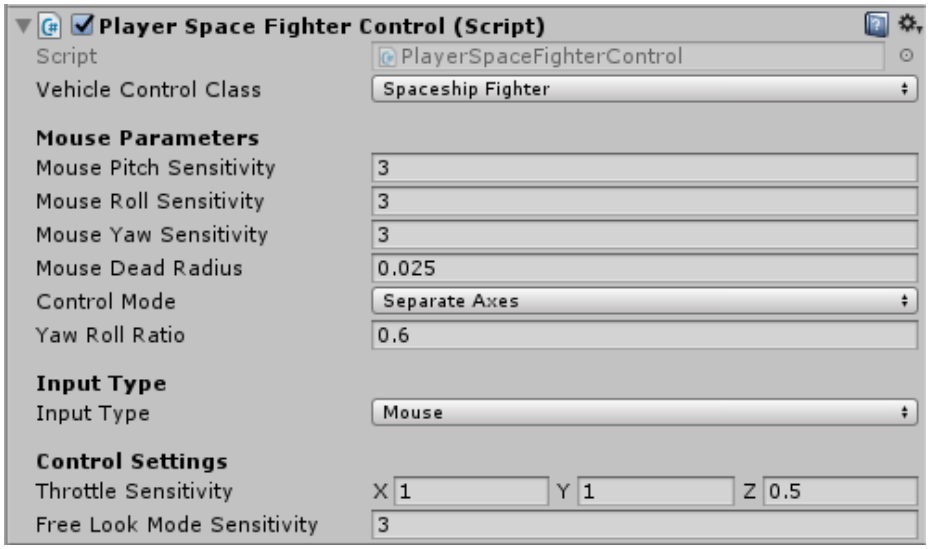
To support the possibility of a player using many different vehicles during the game, this kit separates input entirely from the player and the vehicle. Any number of input scripts can be created for different vehicles, and using a VehicleControlClass enum, the correct input script can be run automatically when the player enters a new vehicle.

11.1 Creating a New Input Script

To create an input script for a vehicle, the script must implement the IVehicleInput interface. This goes for both player and AI scripts. By implementing this interface, each script will have a value for

the VehicleControlClass enum, which can be matched to the VehicleControlClass of a new vehicle when the player enters it.

This kit includes two example input scripts: PlayerSpaceFighterControl.cs (for the player) and AISpaceFighterControl (for the AI). It is recommended to use these scripts as templates for writing your own input scripts.



Vehicle Control Class: The control class of the vehicle that this input script is for.

Mouse Pitch Sensitivity: How fast the ship pitches when mouse is positioned above or below the screen center.

Mouse Roll Sensitivity: How fast the ship rolls when the vehicle is using linked yaw/roll with the mouse.

Mouse Yaw Sensitivity: How fast the ship yaws when mouse is positioned left or right of the screen center.

Mouse Dead Radius: The viewport radius around the center where the mouse position does not affect the vehicle steering.

Control Mode: Whether the yaw and roll are linked (e.g. Starlancer style) or the pitch, yaw and roll are on separate axes.

Yaw Roll Ratio: The ratio of yaw to roll when the control mode is set to Linked Yaw Roll.

Input Type: Mouse/Keyboard or just Keyboard.

Throttle Sensitivity: How fast the throttle changes when the throttle up/down buttons are pressed.

Free Look Mode Sensitivity: How fast the camera looks around based on mouse movement in Free Look Mode.

11.2 Loading an Input Script

To add an input script to a player, simply add the script to a child transform of the player (which has a Game Agent component on the root transform. Set the VehicleControlClass field in the inspector to match the type of vehicle that the input script is for.

When the player enters a new vehicle, it will search through all the input scripts in its hierarchy for one that matches the VehicleControlClass value on the vehicle and run the first one that is found.

11.3 Rewired Integration

This kit includes a comprehensive integration with the Rewired input system on the Asset Store, which makes it easy to use all kinds of controllers in your game. This integration includes maps for the gamepad and HOTAS templates, which means that you can plug-and-play any of the controllers supported by these templates.

This integration is not included in the package and can be downloaded here:

https://www.dropbox.com/s/dy5irgkdmqbwj4v/RewiredIntegration1_21.unitypackage?dl=0

12. Vehicle Camera

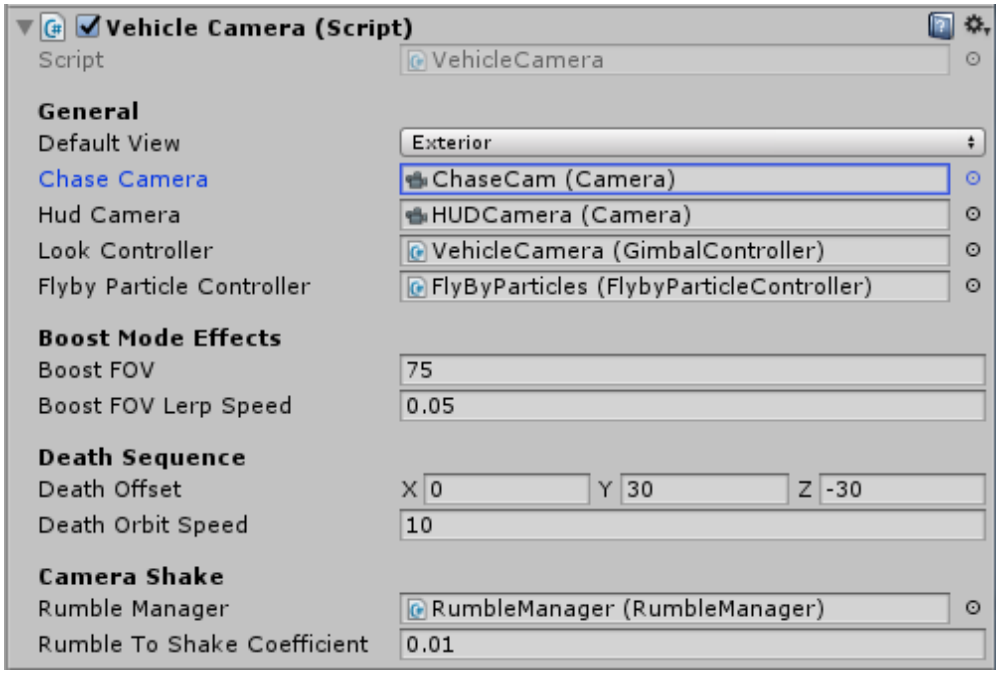
This kit includes a versatile vehicle camera that makes it easy for you to create different camera views for your vehicle and switch between them.

12.1 The Vehicle Camera Component

The VehicleCamera component provides the ability to:

- Automatically switch to a new vehicle when the player enters it.
- Switch between camera views.
- Look around 360 degrees (e.g. for cockpits) using a Gimbal Controller (see the Gimbals section)
- Change the FOV during boost mode.

A VehicleCamera prefab is included in the kit, and is used in all of the example scenes.



Default View: The camera view that is selected by default when the player enters a new vehicle.

Chase Camera: The main gameplay camera.

HUD Camera: The camera that shows the HUD overlaid on top of the scene.

Look Controller: The gimbal controller that is used to look around when the camera is in free look mode.

Flyby Particle Controller: the particles that appear when the vehicle is moving to give a sense of speed (stretched when the player is moving fast).

Boost FOV: The field of view of the camera when the player boosts.

Boost FOV Lerp Speed: How fast the camera field of view changes when the boost is started/stopped.

Death Offset: The orbital position offset of the camera from the vehicle when the player dies.

Death Orbit Speed: How fast the camera orbits around the player's vehicle when the player dies.

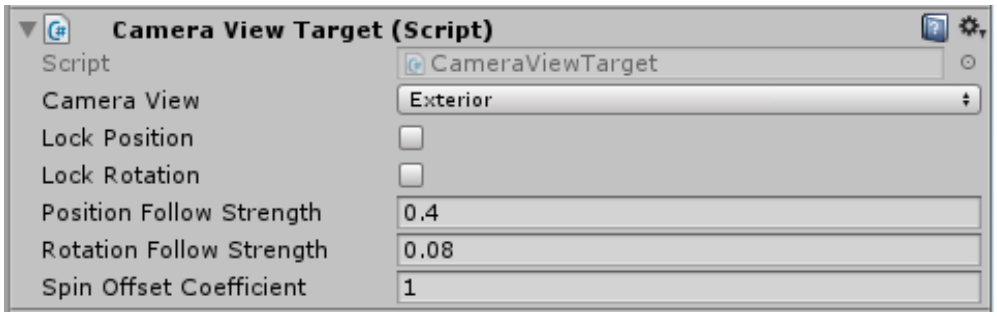
Rumble Manager: The rumble manager in the scene that drives camera vibration effects.

Rumble To Shake Coefficient: How much the camera shakes according to the rumble level.

12.2 Creating a New Camera View

To create a new camera view for your vehicle, create a child gameobject and add a CameraViewTarget component to it. This transform is what the vehicle camera will try to follow when this view is selected, in terms of both position and rotation.

In the inspector, you can set the type of view (so that you can call it on the VehicleCamera script via code), as well as parameters controlling how fast the camera will move to try to match the transform.



Camera View: The camera view that this transform represents in terms of position and rotation.

Lock Position: Whether the camera should be locked positionally to this transform when in this camera view.

Lock Rotation: Whether the camera should be locked rotationally to this transform when in this camera view.

Position Follow Strength: How fast the camera 'catches up' to the position of this transform when in this camera view.

Rotation Follow Strength: How fast the camera 'catches up' to the rotation of this transform when in this camera view.

Spin Offset Coefficient: How much the camera rotates relative to this transform when the vehicle is rotating (e.g. Starlancer style chase view).

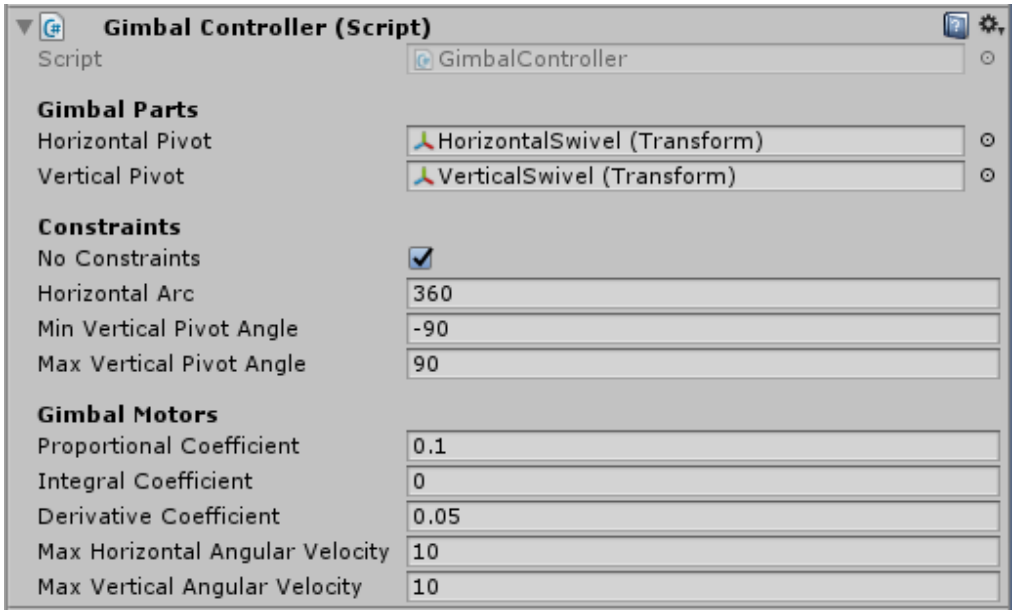
13. Gimbals

This kit includes a component called GimbalController, which is a very versatile component allowing full 360 degree rotation of a gimbal (using a horizontal pivot and a vertical pivot). This component provides the ability to:

- Track a world position in space and adjust how fast the gimbal moves to orient itself.
- Incrementally rotate the horizontal and/or vertical pivots (e.g. for mouse-controlled looking around).

- Set the rotation of the horizontal and vertical pivots.
- Constrain the horizontal and vertical pivots to particular angle ranges.

This component is used by the Vehicle Camera to look around, and can also be used to create turrets in your game.



Horizontal Pivot: The transform that rotates horizontally when orienting this gimbal.

Vertical Pivot: The transform that rotates vertically when orienting this gimbal.

No Constraints: Whether the horizontal and vertical pivots should be unconstrained (allowed to rotate through 360 degrees).

Horizontal Arc: The angle range around zero rotation (gimbal oriented forward) that the horizontal pivot can rotate.

Min Vertical Pivot Angle: The minimum angle that the vertical pivot can rotate relative to zero rotation (when the gimbal is oriented directly forward).

Max Vertical Pivot Angle: The minimum angle that the vertical pivot can rotate relative to zero rotation (when the gimbal is oriented directly forward).

Proportional Coefficient: The Proportional coefficient of the PID controller that drives an automatic gimbal.

Integral Coefficient: The Integral coefficient of the PID controller that drives an automatic gimbal.

Derivative Coefficient: The Derivative coefficient of the PID controller that drives an automatic gimbal.

Max Horizontal Angular Velocity: The maximum velocity of the horizontal pivot of the gimbal controller.

Max Vertical Angular Velocity: The maximum velocity of the vertical pivot of the gimbal controller.

14. AI

This kit includes a versatile physics-driven AI for space games that includes:

- Physics-based steering.
- Obstacle avoidance.
- Combat.
- Patrolling.
- Formation flying.
- Group behavior.

14.1 AI Behaviours

The way that the AI operates is that the main input script holds references to a collection of 'behaviours' such as Patrolling, Combat, Obstacle Avoidance and so on. Each behaviour is a separate component that is instantiated from a prefab by the input script when the game starts.

According to the context of the game, a specific behaviour is run, and it writes its results to something called a Blackboard, which is a component that stores the results of the behaviours (e.g. throttle and steering input values) so that the main AI input script can implement it. Behaviours can be blended by blending the results with the values already on the Blackboard, rather than overwriting them.

This allows the AI behaviour to be built modularly from smaller, more specific behaviours, and adapt to the context of the game.

14.2 Creating a New AI Behaviour

To create a new behaviour for your AI:

1. Create a script that implements the IVehicleControlBehaviour interface.
2. Create a new gameobject, add the script and save it as a prefab.
3. Instantiate the behaviour in your input script and call it's Tick method when you want the behaviour to be run.

14.3 Physics-Based Vehicle Steering

This kit includes a versatile script called **Maneuvring.cs** with several methods for calculating physics-based vehicle steering using a PID controller. These methods are static (meaning the script doesn't have to be instantiated in the scene to call them) and they can be used inside of behaviour scripts to make the vehicle steer toward a point in space.

You can also restrict the maximum pitch (up/down rotation) and roll (left/right rotation) for example to make capital ships move in a more reasonable way.

Using this functionality, an AI can operate all the same vehicles as the player, in the same way.

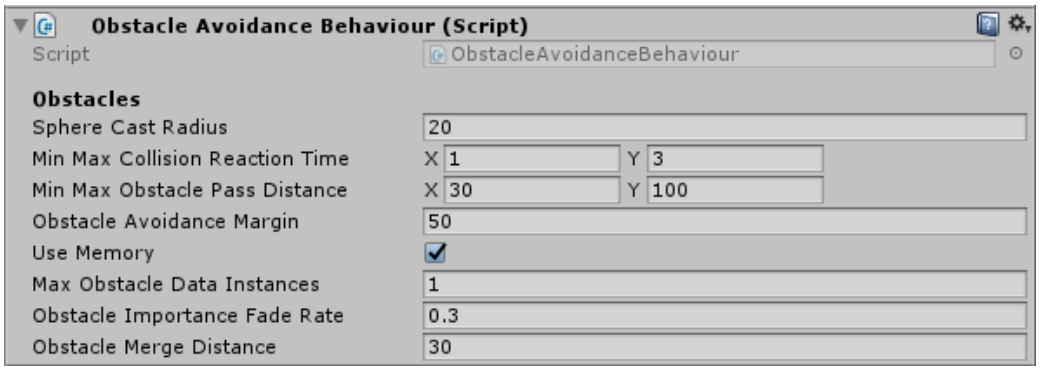
These methods include:

- **TurnToward**: make the ship turn toward a point in space, while setting limits on the pitch and roll angles (e.g. for large capital ships).
- **Formation**: Make the ship follow another vehicle (e.g. a formation leader) using both steering and throttle, and with a designated position offset.

All these methods generate input values for throttle and steering which can then be sent to the Engines component of the vehicle in your input script.

14.4 Obstacle Avoidance Behaviour

This kit includes physics-based obstacle avoidance using SphereCasts in the ObstacleAvoidance behaviour. This behaviour can store information for multiple obstacles, 'remember' them for a specified time, and blend them to calculate the optimum path to follow through multiple obstacles.



Sphere Cast Radius: The radius of the sphere cast used to detect approaching collisions in obstacle avoidance.

Min Max Collision Reaction Time: The time-to-impact range in which the proximity risk factor increases from 0 to 1.

Min Max Obstacle Pass Distance: The range of passing distances where the proximity risk factor increases from 0 to 1.

Obstacle Avoidance Margin: The distance from the impact point that the obstacle avoidance waypoint is calculated at.

Use Memory: Whether this component should 'remember' obstacles even after they are not detected anymore by the spherecast (useful to reduce zig-zag movement).

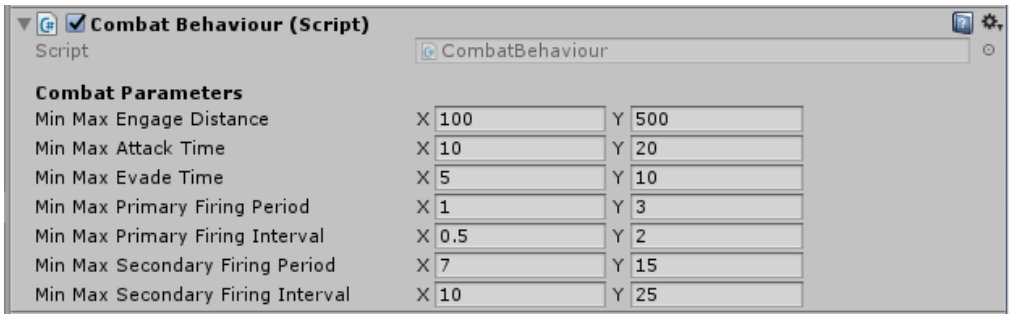
Max Obstacle Data Instances: The maximum number of detected collisions that can be used when determining an avoidance route.

Obstacle Importance Fade Rate: How fast the importance of 'remembered' obstacles fades after they are no longer detected.

Obstacle Merge Distance: The minimum distance between two obstacles to be considered as separate obstacles.

14.5 Combat Behaviour

The Combat behaviour alternates between attacking and evading to provied varied and exciting combat.



Min Max Engage Distance: The minimum and maximum distance to the target in which the AI will fire at the target.

Min Max Attack Time: The random range of time that the AI will spend in the attack state before switching to evasion.

Min Max Evade Time: The random range of time that the AI will spend in the evade state before switching to attack .

Min Max Primary Firing Period: The random range of the length of the primary weapon firing bursts when the AI is attacking the target.

Min Max Primary Firing Interval: The random range of the interval between the primary weapon firing bursts when the AI is attacking the target.

Min Max Secondary Firing Period: The random range of the length of the secondary weapon firing bursts when the AI is attacking the target.

Min Max Secondary Firing Interval: The random range of the interval between the secondary weapon firing bursts when the AI is attacking the target.

Base Stamina Change Rate: How fast the AI runs out of or recuperates stamina when attacking or evading the target.

14.6 Patrol Behaviour

The Patrol behaviour implements looped patrolling between a series of patrol targets that serve as waypoints. When the AI is part of a group, patrolling involves formation flying with the patrol leader, who is the one that follows the patrol route.

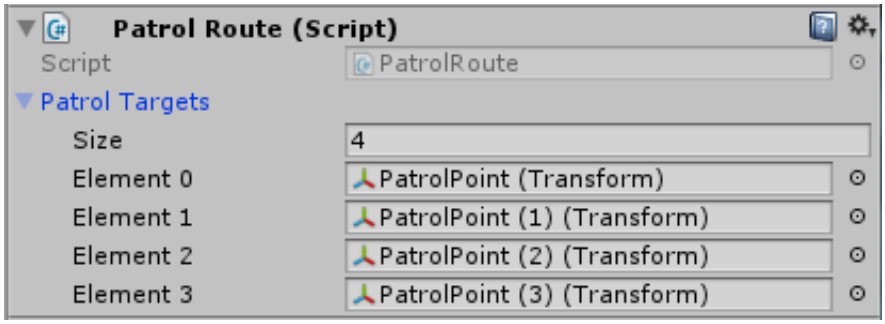
Patrol Speed Factor: The coefficient applied to the speed of the AI vehicle when patrolling.

Patrol Steering Factor: The coefficient applied to the steering speed of the AI vehicle when patrolling.

14.7 Creating a Patrol Route

To create a new patrol route in your scene, create a new gameobject and add the PatrolRoute component. Then, create transforms in your scene for all the waypoints and add them to the Patrol Targets list in the inspector of the PatrolRoute component.

Then, add the PatrolRoute to the inspector of the GroupMember script (see below) on your AI.



Patrol Targets: The transforms representing the waypoints that make up this patrol route.

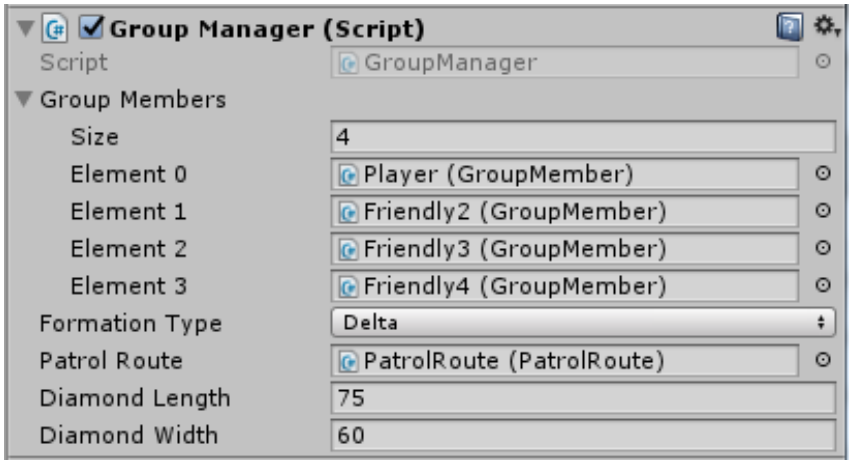
14.8 Group Behaviour

Group behaviour can be implemented with the GroupMember component, which can be added to the root transform of an AI game agent to designate it as a member of a group.

Patrol Route: The patrol route that this group member should follow when in Patrol mode.

A GroupManager script is provided in the kit to which group members can be added, in order to implement group behaviour such as:

- Distributing targets evenly amongst members, to avoid ‘bunching’ of ships on one target.
- Designation of a formation leader.
- Designation of a formation shape.
- Anything else you’d like AI ships to do as a group.



Group Members: The group members that are assigned to this group manager.

Formation Type: The type of formation that this group flies in when patrolling.

Patrol Route: The patrol route for this group to be followed when patrolling.

Diamond Length: The length of the diamond shape of the formation when it is set to Delta.

Diamond Width: The width of the diamond shape of the formation when it is set to Delta.

14.9 Behaviour Designer Integration

This kit comes with a basic integration for the Behaviour Designer behaviour tree package on the asset store. Behaviours have been implemented as Behaviour Designer Actions, and a behaviour tree built for an AI space fighter. The integration package can be downloaded here:

https://www.dropbox.com/s/fz9i4tpyguymmr1/BehaviorDesignerIntegration.unitypackage?dl=0

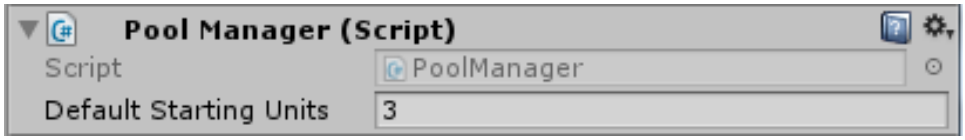
15. Object Pooling

This kit features an easy-to-use object pooling system to increase the efficiency of often-used items such as bullets.

To enable object pooling, simply add a new gameobject anywhere in the scene and add a PoolManager component to it.

15.1 The PoolManager Component

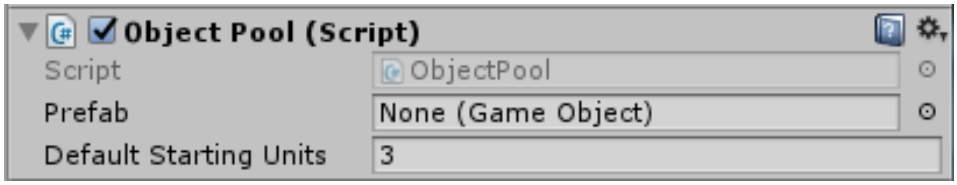
The PoolManager component provides a way to easily get pooled objects from any script. It is a singleton (meaning it can be accessed without a reference, and there's only ever one of them in the scene). To get a pooled object, call the PoolManager.Instance.Get method, passing the object's prefab reference as a parameter, along with the desired position and rotation where it should be spawned, as well as the transform it should be parented to (which can be null).



Default Starting Units: The default number of pooled items that are created when a new object pool is created.

15.2 The ObjectPool Component

The ObjectPool component represents a pool for a single object, and a new one is created by the PoolManager when an object is requested that does not yet have a pool.



Prefab: The prefab of the item to be pooled.

Default Starting Units: The default number of pooled items that are created for this object pool when the game starts.

16. Rumble Manager

The RumbleManager component allows you to create 'rumbles' in your game (for example to drive controller vibration and camera shaking when the player is hit). This component is a singleton (which means that it can be accessed statically and there is only ever one of them in the scene).

To create a new rumble, call the RumbleManager.Instance.AddRumble method. This method allows you to specify the:

- Maximum level of the rumble (0 – 1)
- The attack time (how long the rumble takes to build up).
- The sustain time (how long the rumble is held at its maximum value).
- The decay time (how long the rumble takes to fade out).

Every frame, the RumbleManager searches through all the rumbles that currently exist, and find the maximum level of all of them. This level can be accessed by calling the RumbleManager.Instance.CurrentLevel, which specifies the current rumble level from 0-1 and can be used to drive effects in the game.

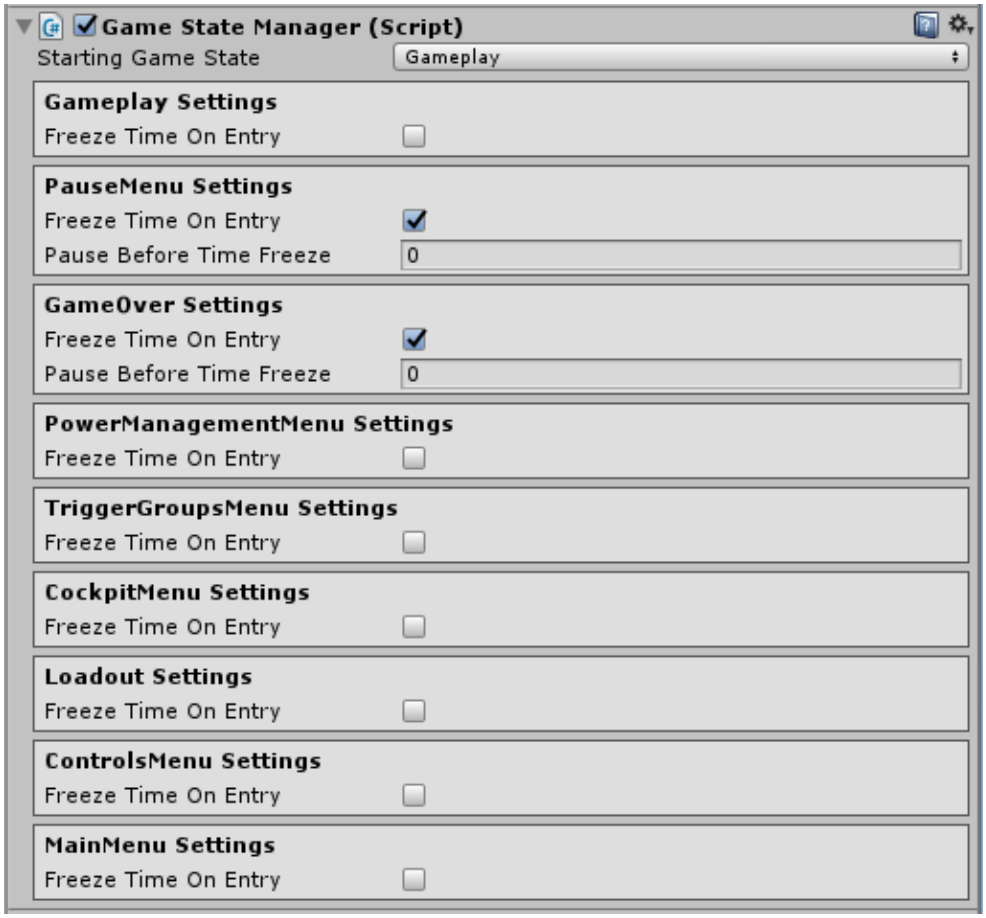
17. Game State

In this kit, the game is in a single state at any given time. Entering a new game state can be used to enable and disable specific aspects of the game, such as showing menus, enabling different input, or time freezing. Examples in this kit include Gameplay, PowerManagementMenu, PauseMenu, GameOver and more.

17.1 The GameStateManager Component

The GameStateManager component provides a reference point for the current game state. It is a singleton, meaning that it can be accessed statically and there is only ever one of them in the scene. There must be a GameStateManager in every scene that uses this kit.

To add the GameStateManager to your scene, create a new gameobject and add a GameStateManager component to it.



Starting Game State: The game state that the game should start in.

Freeze Time On Entry: Whether time should be frozen on entering this game state.

Pause Before Time Freeze: How long to pause before freezing time when entering this game state.

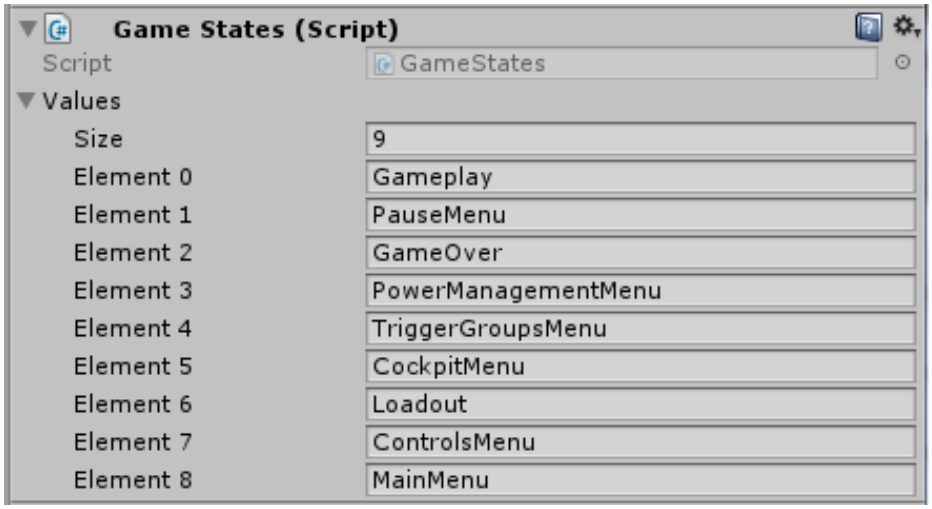
17.2 Entering a New Game State

To enter a new game state, call the GameStateManager.Instance.EnterGameState method, passing the new game state value by string.

17.3 Adding New Game States

To make it easy to add new game states to your game, there is a prefab in a Resources folder called GameStates. This has a component called GameStates.cs that allows you to edit your game states as

strings in a list. When you edit these strings, the inspector of the GameStateManager component in your scene automatically updates, allowing you to specify whether to freeze time when the game enters each of the game states.



18. Floating Scene Origin

When gameplay occurs more than approximately 5000 units from the center of the scene, due to the nature of floating point precision, games can experience jittery, shaky movement. To prevent this, it's necessary to keep the player near the center of the scene, but this can result in game spaces feeling very small, especially with fast ships.

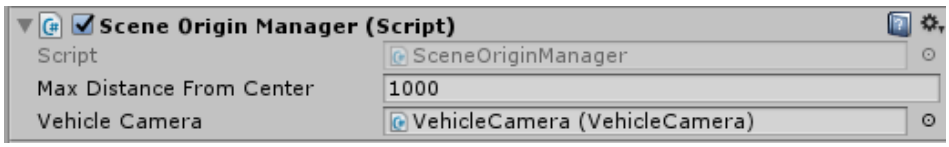
To solve this problem, this kit provides a framework that allows the scene to periodically re-arrange itself by putting the player at the center and positioning everything else in the scene relative to the player. This way, the player can cover very large distances in the game world without ever getting too far from the center.

18.1 The SceneOriginManager Component

To add a floating scene origin to your game, create a new gameobject in your scene and add a SceneOriginManager component to it. Everything that is part of the scene that is shifted around the player must be parented to this transform.

This component periodically places the player at the center of the scene, and shifts everything else (that is parented to it) relative to the player, according to the player's 'real' position in the game world.

Note that Unity's TrailRenderer component is not compatible with this way of doing things, as it stores the positions of the trail vertices in world space, causing the trail to stretch when it is moved.



Max Distance From Center: The maximum distance that the player is allowed to get from the center.

Vehicle Camera: The vehicle camera used in the scene.

18.2 The SceneOriginChild Component

Sometimes you would like to instantiate something in the scene that is part of the scene that is shifted around the player. To make sure that the object is parented to the SceneOriginManager transform, add a SceneOriginChild component to the root transform of the prefab. This component will then parent the object to the SceneOriginManager transform when the object is created in the scene.

19. The UVCEventManager

To make it easier to create events that propagate through the scene, the UVCEventManager component has been provided. This component is a singleton (meaning that it can be accessed statically and there is only ever one of them in the scene).

19.1 Triggering An Event

Events can be triggered in the scene by calling the Trigger method in the UVCEventManager script, and passing a set of arguments that match one of the methods in the script.

19.2 Listening For An Event

To listen for an event, a script must call the StartListening method on the UVCEventManager, and the UVCEventType value as well as the rest of the arguments list must match that of the triggered event.

To stop listening for an event, the script must call the StopListening method on the UVCEventManager, and again the UVCEventType value as well as the rest of the arguments list must match that of the triggered event.

19.3 Adding a New Event

To add a new event that can be triggered and listened for in the scene, create new methods in the UVCEventManager script, with a unique set of arguments that pass relevant information about that event. You must create a new Trigger, StartListening and StopListening method, each with the same arguments list.