Space Combat Kit

User Guide

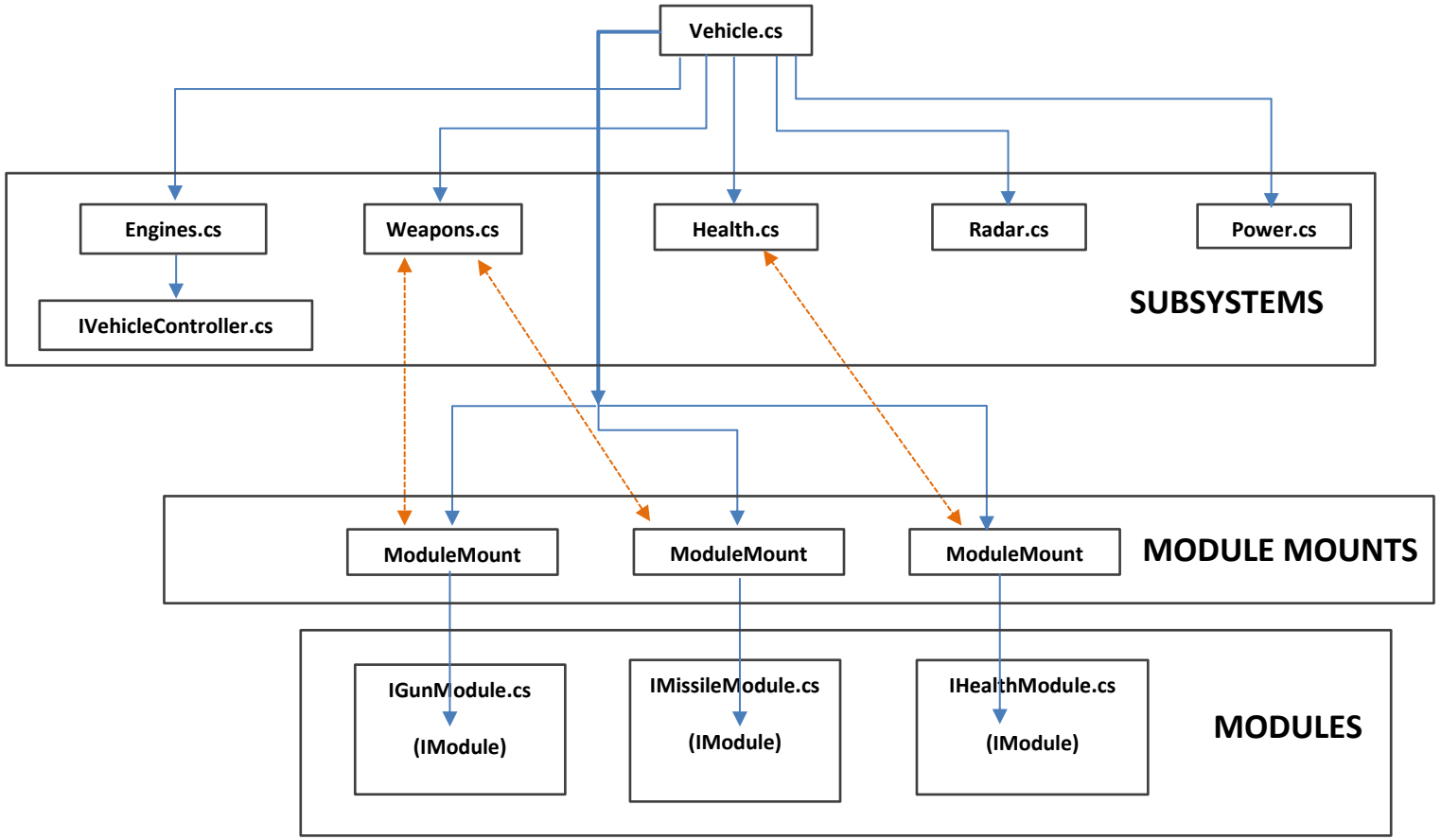
Publisher: VSXGames

Contact: contact@vsxgames.com

Contents

1. Introduction 5
2. Vehicles 5
   2.1 Overview 5
   2.2 Getting Started 6
3. Subsystems Overview 6
4. Engines 7
   4.1 Overview 7
   4.2 Quick Start 8
   4.3 Power, Thrust and Torque 8
5. Weapons 9
   5.1 Overview 9
   5.2 Quick Start 9
   5.3 Weapons Computer 9
   5.4 Weapons Modules 9
   5.5 Projectile Controller 10
   5.6 Missile Controller 10
6. Radar 10
   6.1 Overview 10
   6.2 Quick Start 11
7. Health 11
   7.1 Overview 11
   7.2 Quick Start 12
   7.3 Health Fixtures 12
   7.4 Health Generators 13
8. Power 14
   8.1 Overview 14
   8.2 Configuring Power 14
   8.3 Power Management During Gameplay 15
9. Modules 15
   9.1 Creating a Module 16
   9.2 Creating a Module Mount 16
   9.3 Creating a Mountable Module 16

9.4 Mounting a Module 17
9.5 Gimbal Controllers 17
10. Trigger Groups Manager 18
10.1 Quick Start 18
10.2 Making a Module Triggerable 18
10.3 Creating/Changing Trigger Groups 19
11. Game Agents 19
11.1 Overview 19
11.2 GameAgent 19
11.3 Game Agent Manager 20
11.4 Entering/Exiting Vehicles 20
12. AI 20
12.1 IVehicleInput 20
12.2 AI Behaviours 21
13. HUDs 21
13.1 Overview 21
13.2 The HUD component 21
13.3 Visor Target Tracking 22
13.4 3D Radar 23
13.5 Target Hologram 23
14. Event Manager 23
15. Integrations 24
15.1 Behavior Designer 24
15.2 Rewired 24
16. That's it! 24


Vehicle.cs
Engines.cs
Weapons.cs
Health.cs
Radar.cs
Power.cs
IVehicleController.cs
SUBSYSTEMS
ModuleMount
ModuleMount
ModuleMount
MODULE MOUNTS
IGunModule.cs
(IModule)
IMissileModule.cs
(IModule)
IHealthModule.cs
(IModule)
MODULES

1. Introduction

The Space Combat Kit for Unity is designed to provide a complete and flexible foundation for creating space games of all kinds, from simple arcade shooters to sophisticated, immersive space simulators.

The kit is built around a Vehicle/Subsystem/Module framework, which is primarily to enable functionality to be added in a flexible way to a vehicle via modules. This means that the basic rules that form the basis of this kit are:

- A **Vehicle** is made up of **Subsystems**;
- **Subsystems** manage **Modules**;
- **Modules** provide concrete functionality;

It is not necessary to follow these rules always strictly, but if you are trying to understand how the kit works or thinking about the best way to add something of your own, it's a good starting point.

Although this kit at the moment only provides an implementation for space combat, the code and framework is designed to be able to represent many other types of vehicles, even to the point of something like a sci-fi suit. For anything that can be described as *a form that the player takes to move around in the game world, load/unload equipment, give/receive damage and generally interact with the game world around them*, this kit is potentially suitable for the job. It is intended to expand this kit in future to explicitly cover these other types of vehicles.

Game Agents

Game agents represent either a player or an AI control script - essentially a component which can assume and relinquish control of any vehicle. A game agent also provides a fixed reference for a player or AI in the game even when they may be transitioning between different types of vehicle during gameplay – allowing scene-level scripts (such as camera, HUD manager, menus, etc) to maintain the correct focus.

2. Vehicles

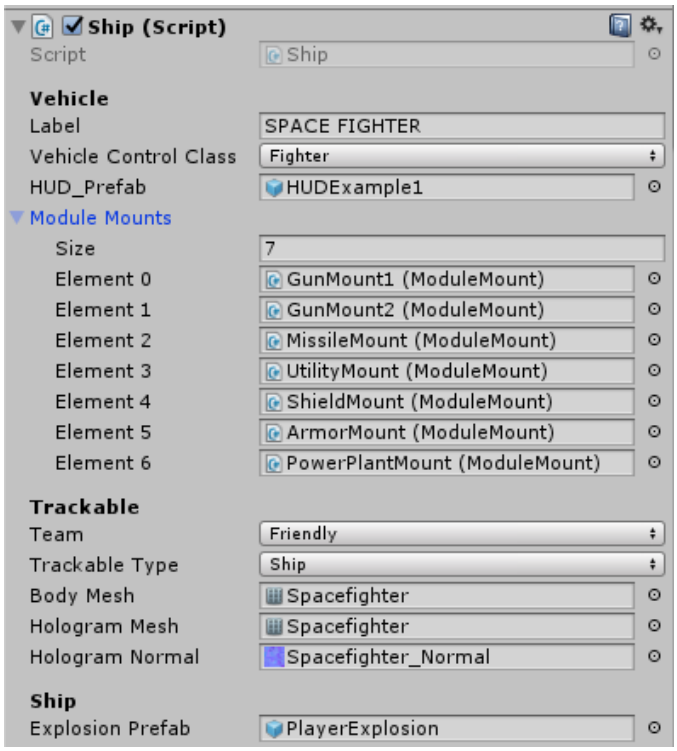
2.1 Overview

The Vehicle component is a base class that forms the foundation of this kit. It is designed to be an abstract framework for representing the form with which a player or AI can physically move around in the game world and interact with it.

Although the kit currently extends the Vehicle class only in the form of a space vehicle, this class is potentially suitable as a base for mechs, buggies, and other vehicle types – even something like a sci-fi suit – providing a very straightforward and easy solution for creating games where the player can assume control of many different vehicles.

The role of the Vehicle component is to provide an interface for control scripts to access subsystem-level components (which themselves provide an interface with modules). The Vehicle class also deals with vehicle-level events such as entering, exiting, and destroying a vehicle in a clean and organised way.

Here is a image of the inspector of the Ship component (which extends the Vehicle component and implements the ITrackable interface so that it can be tracked by radar).



2.2 Getting Started

To begin creating a vehicle, simply add a component extending the Vehicle base class to the root transform of the vehicle. The provided Ship component can be used for space vehicles. The next sections will discuss the subsystems (and the modules associated with these subsystems) that give a vehicle its capabilities.

3. Subsystems Overview

Subsystems are components which manage groups of modules that relate to a subset of vehicle functionality, such as:

- Engines
- Weapons
- Health
- Radar
- Power

These components, in most cases, do not directly implement functionality but instead manage the different types of Modules attached to the vehicle. This design enables much of a vehicles capability to be transferred into a system of upgrade modules, making it easy to create upgrade and loadout menu mechanics in your game.

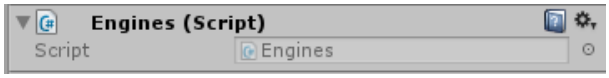
Subsystems extend the Subsystem base class, which is a class that listens for events related to loading and unloading of modules from the vehicle's module mounts. To add a subsystem component to a vehicle, simply add the component to the root transform of the vehicle (where the Vehicle or Vehicle-extending component is).

More subsystems can be added to the kit if required – all that's needed is to update the Vehicle component to look for and hold the appropriate references at runtime.

Each of the subsystems, how to add it to a vehicle, and the way that it manages its modules, are explained in further detail in dedicated sections below.

4. Engines

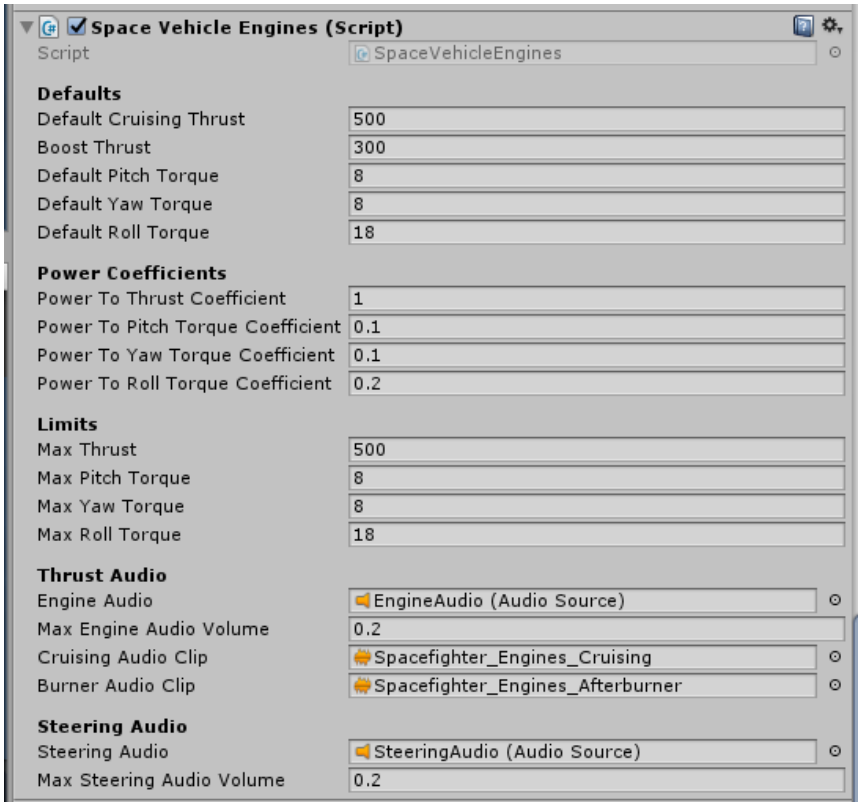
4.1 Overview



The Engines component manages engine related functions on the vehicle. Since vehicles might have very different types of control, the Engines script is quite simple and essentially just a link to the actual vehicle controller script (a component extending the IVehicleController interface). Its basic role is to pass translation (movement) and rotation (steering) values to the vehicle controller script, which can then take these values and perform whatever type of control behaviour the vehicle is designed for. These values are from -1 to 1 for each of the x, y, and z axes. This way, the framework of the kit can be used for many kinds of vehicles, or for games of different levels of control sophistication.

To implement vehicle control, add a vehicle controller component (which is a component extending the IVehicleController interface) to the root transform of the vehicle. Input is passed to this interface in the form of a -1 to 1 value on each axis for translation and rotation, as well as a boolean value for boost capability.

For a standard space vehicle, the provided SpaceVehicleEngines component can be used:



4.2 Quick Start

To add engines to a vehicle, add an Engines component to the root transform of the vehicle and set the values in the inspector. You must also add a specialized control component – one that extends the IVehicleController interface – to the root transform of the vehicle. The Engines component will automatically grab the first one that it finds.

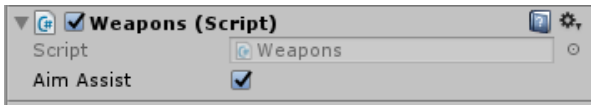
4.3 Power, Thrust and Torque

If the vehicle has a Power subsystem (which provides varying levels of power to different subsystems), the SpaceVehicleEngines component converts power to thrust (movement forces) and torque (steering forces) according to coefficients specified in the inspector.

Note that this component uses the *Direct* fraction of power provided to the engines for thrust and torque, and uses the *Recharge* fraction of power provided to the engine to recharge the boost capability. Please see the Power section for more information.

5. Weapons

5.1 Overview



The Weapons subsystem component manages all the weapons that the vehicle is carrying, providing a link between the weapons modules and various other components, such as input, weapons computer, HUD and more.

5.2 Quick Start

To add weapons capability to a space vehicle, add the Weapons component to the root transform of the vehicle, and set the values in the inspector. This does not implement any actual weapons – weapons must be added as modules (see Weapons Modules section below).

5.3 Weapons Computer

The weapons computer (which is a component extending the IWeaponsComputer interface) provides lead target calculation (calculating the position to aim to hit a fast-moving target) for weapons implementing the IGunModule interface, and missile locking capability for weapons implementing the IMissileModule interface.

Lead target information is available from this component in the form of the LeadTargetData class, while missile locking information is available in the form of the LockingData class.

The weapons computer has been provided in this kit in the form of a module, in case it is desired to create different weapon computers as upgrade modules.

5.4 Weapons Modules

Weapons are implemented in this kit in the form of modules that are mounted onto the vehicle's module mounts. A weapon module component must:

- Be a module (implement the IModule interface);
- Be a weapon module (implement the IWeaponModule interface, this is mainly for menu purposes);
- Be a gun module or a missile module (implement either the IGunModule or IMissileModule interface);
- To be triggered via the TriggerGroupsManager, it must implement the ITriggerable interface;

In this kit, the main types of weapons have been provided as base classes (ProjectileWeapon, BeamWeapon and MissileWeapon). Using these classes, it should be possible to create a variety of weapons with different characteristics and effects.

Weapons modules (like any components implementing the IModule interface) can be either fixed or gimballed, which means they can rotate toward a target independent of the Vehicle's orientation. See the Module section for more details.

5.5 Projectile Controller

The ProjectileController component is a control script for a projectile fired by a projectile weapon module. It uses raycasting rather than colliders to reliably detect collisions at any speed, and the raycasting frequency can be adjusted for performance. It also carries a reference to the Game Agent who fired it, to keep track of scores and enable reactive AI behaviour on a ship that is hit.

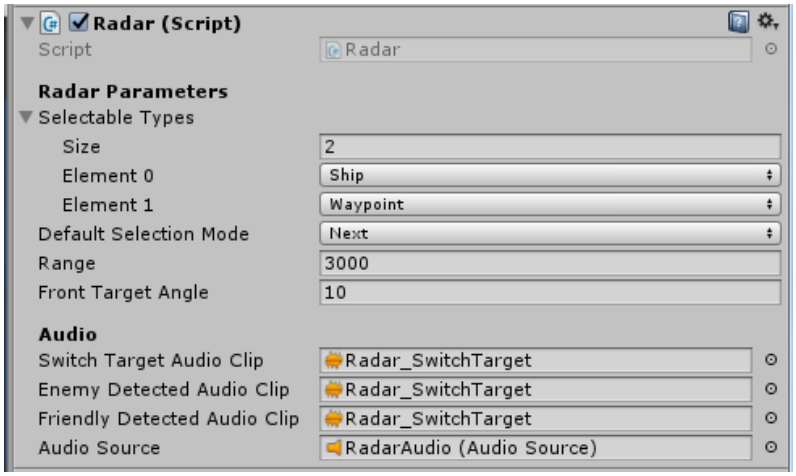
5.6 Missile Controller

The MissileController is a control script for a missile fired by a missile module. Like the projectile controller, it uses raycasting to detect collisions, the frequency of which can be adjusted for performance, and carries a reference to the Game Agent who fired it.

The MissileController script is physics-driven and uses a PID controller to steer toward its target. It can lose lock due to exceeding the missile's locking angle or the locking distance to the target, and contains a detonation timer to explode and remove it from the active scene when it loses lock.

6. Radar

6.1 Overview



The Radar component provides the ability to track multiple targets and select between them according to Next, Nearest and Front selection modes, as well as specifying selection of hostile (enemy) or non-hostile (friendly) targets.

To get target information, the Radar component links to a scene-level, singleton RadarSceneManager component. All trackable objects are required to register/unregister with the

RadarSceneManager so that it does not have to scan the scene every frame, and so that many different ships can access the data efficiently.

Note that missile lock and lead target calculation is carried out by the WeaponComputer component, not by the Radar. However, the Radar is responsible for enabling the selection of a target, which the weapons computer then uses for its calculations.

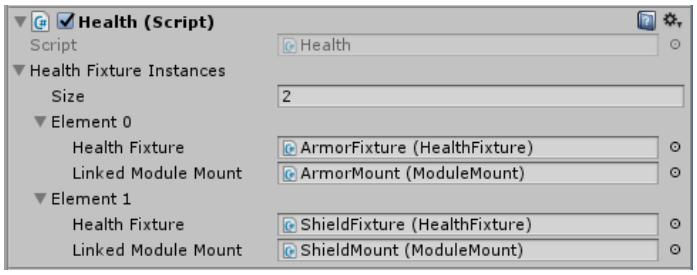
The Radar component does not visualize any target information – please see the HUD section for that.

6.2 Quick Start

To add radar capability to a vehicle, add a Radar component to the Vehicle's root transform. Also, it is necessary to add a RadarSceneManager component to the scene, if one has not been added already.

7. Health

7.1 Overview



The health subsystem manages:

- References to health fixtures/generators and their values/settings;
- Pathways for damage and destruction events to be passed higher up the vehicle hierarchy;
- Power distrubution to rechargeable health modules;

The Health subsystem does not directly implement health functionality. Instead, the implementation of a vehicle's health is made up of two parts:

1. Health fixtures;
2. Health generators;

Health fixtures are basically colliders/mesh renderers that are specific to a ship, which are linked to health generators (such as shield generators) that are mounted onto the ship. This link is achieved by associating a specific module mount with a health fixture, meaning that whenever the module

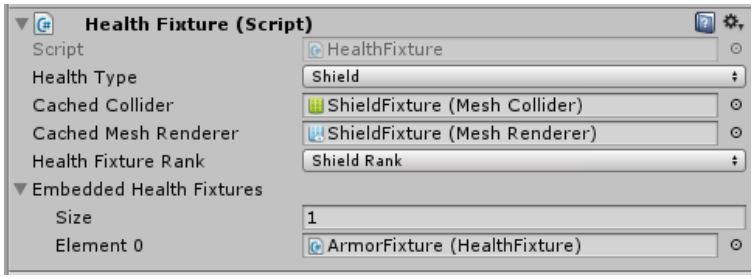
mount loads a health generator module, the subsystem will attempt to link it with the health fixture that has been associated with that module mount.

Health fixtures and health generators are discussed in greater detail below.

7.2 Quick Start

To add a Health subsystem to a vehicle, simply add a Health component to the vehicle's root transform.

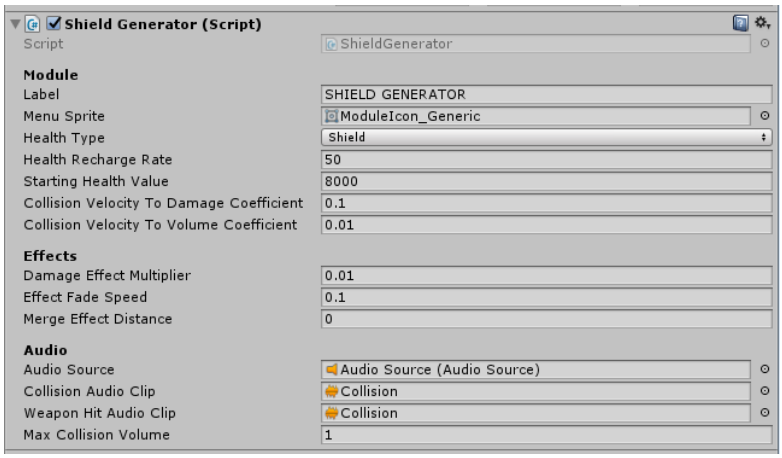
7.3 Health Fixtures



A health fixture is a component which basically contains a reference to a collider and/or a renderer, implements the IDamageable interface, and is linked to a health generator (a module implementing the IHealthGenerator interface). This way, a ship can contain a pre-determined set of colliders/mesh renderers to which health generators can be linked during loadout or in the inspector, giving the option of multiple damageable components in a ship (such as the ability to destroy specific subsystems in a large capital ship).

Additionally, health fixtures can be embedded inside other health fixtures, such as armor being surrounded by shields. To prevent accidental damage of 'inner' health fixtures, such as during high-speed collisions where the physics engine cannot prevent the obstacle from intersecting with the inner collider, 'outer' health fixtures automatically disable 'inner' health fixtures at the start, activating them only when the 'outer' one is destroyed. Also, to prevent recursive function calling, health fixtures can only be embedded inside other health fixtures of a higher HealthFixtureRank (enum) value.

7.4 Health Generators



Health generators are modules which implement the IHealthGenerator interface. This interface basically provides a way to:

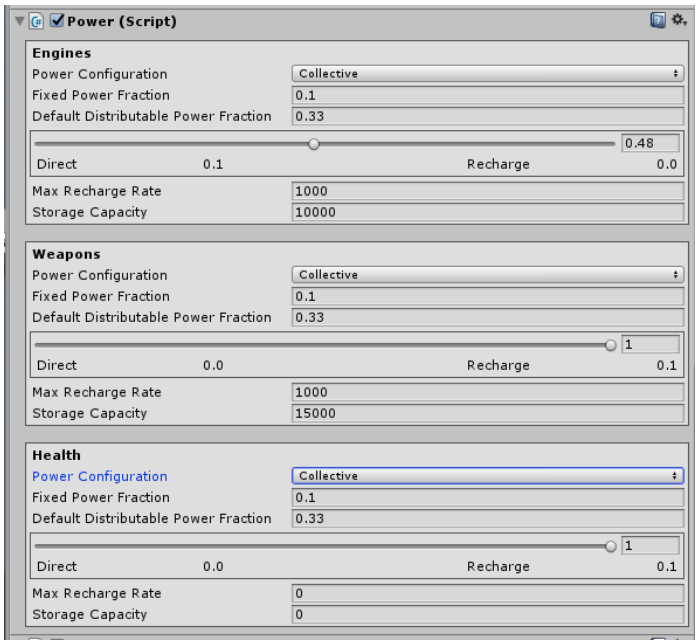
- Determine the health type of the health generator;
- Determine the starting and current health values of the generator;
- Recharge the generator.

Health modules can be of different types – basic armor and shield health types are provided but, by extending the HealthType enum, different health types can be added to the kit.

Health generators can also perform their own visual effects. The example shield generator in this kit communicates with an energy shield shader (that has been provided), showing shield effects where there has been an impact.

8. Power

8.1 Overview



The Power subsystem component provides a way to create power discharge and recharge mechanics for modules attached to the ship, and the opportunity to distribute a finite amount of power among different subsystems.

1. Power is generated via a powerplant (a module implementing the IPowerPlant interface);
2. Power is divided/accessed according to the PoweredSubsystemType enum. For each of the values in this enum, the Power component holds a reference to an instance of the PoweredSubsystem class, each of which can be configured individually (see below);
3. Each subsystem or module can then draw power and access relevant power information by calling methods on the Power component and passing its PoweredSubsystemType as an argument.

Except for when a subsystem power configuration is set to Independent, the Power component does not provide power on its own, but instead contains a reference to a component extending the IPowerPlant interface. The PowerPlant class is provided in the form of a module so that different kinds of powerplants can be created as upgrades.

8.2 Configuring Power

Each Powered Subsystem can have any of the following configurations: *Collective*, *Independent* or *Unpowered*. This setting can be changed on the Power component in the inspector (see image above).

In **Collective** mode, subsystems source power from a collectively referenced powerplant module (a component implementing the IPowerPlant interface). In this mode, it is possible to configure a fixed

fraction of the powerplant's output for each of the subsystems, and to distribute the remainder between subsystems either in the inspector, or dynamically during gameplay (for which the the **power management menu** is provided, see Section 8.3 below);

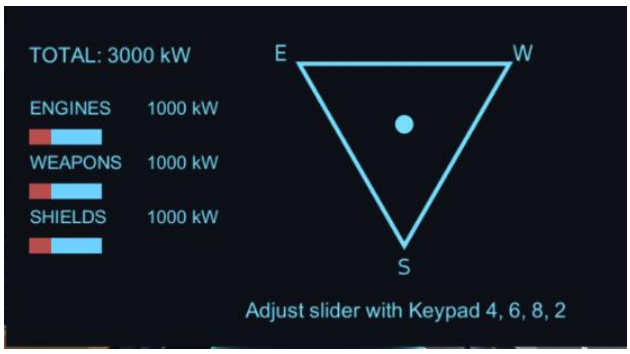
In **Independent** mode, subsystems are not affected by powerplants or the power characteristics of other subsystems, and are for cases where a subsystem should have an 'internal' power source and power storage capacity.

In **Unpowered** mode, a subsystem is not considered in power distribution, and does not interact with the Power subsystem component at all. Any attempt to draw power from a subsystem configured this way will return True.

In both the **Collective** and **Independent** configurations, it is possible to split the total power allocated toward a subsystem between *direct* and *recharge* power. Direct power can be accessed directly 'on tap' but recharge power diverts power to a storage capacity, which can then be discharged in specific situations.

The storage capacity is provided for each subsystem within the Power component, which can then be accessed and discharged externally.

8.3 Power Management During Gameplay



A power management menu has been provided in this kit for allowing a player to dynamically set the power distribution between different subsystems during gameplay. This is through a single PowerManagementMenuController script which resides in the scene and interacts with any Power component found on the vehicle that the scene's GameAgentManager component is currently focused on.

The red bar shows the power which is fixed for a given subsystem, and the blue bar shows the power that can be dynamically distributed among subsystems.

9. Modules

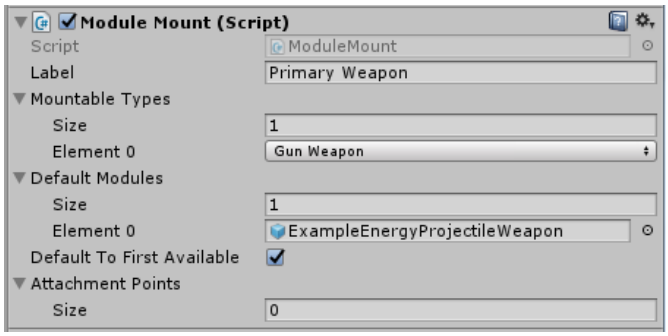
The module system is what provides concrete functionality to a Vehicle, in the form of attachable modules which are managed by the vehicle's subsystem components. Armor, shields, weapons, weapons computer, etc. are all implemented as modules.

There are two parts to the module system – the module itself, and the module mount onto which it is mounted to be used by the vehicle.

9.1 Creating a Module

To create a module that can be mounted on a vehicle, it is necessary to create a script component which implements the IModule interface, and attach it to the root transform of the module prefab. This allows it to be recognized by the loadout menu and to be linked to the vehicle's subsystems. There are several module examples provided in the kit, including weapons modules, health modules, a powerplant module and a weapons computer module. Many more types of modules can be created.

9.2 Creating a Module Mount



To attach a module to a vehicle, it is necessary to attach it to a module mount. Create one on the vehicle by following these steps:

1. Add a ModuleMount component to a transform somewhere in the Vehicle's hierarchy.
2. Go to the Vehicle component (on the vehicle's root transform) and add the module mount component to the Module Mounts list in the Vehicle's inspector.

These ModuleMount components can then be accessed through the Vehicle component's public ModuleMounts list variable.

9.3 Creating a Mountable Module

To allow multiple modules to be stored and dynamically equipped at a module mount during gameplay, a module mount (ModuleMount component) can store multiple modules at its location as instances of the MountableModule class, and contains methods for cycling and selecting between them during gameplay. The (non-MonoBehaviour) MountableModule class represents a module that has been instantiated at the module mount's position, but may or may not be mounted/active in the scene.

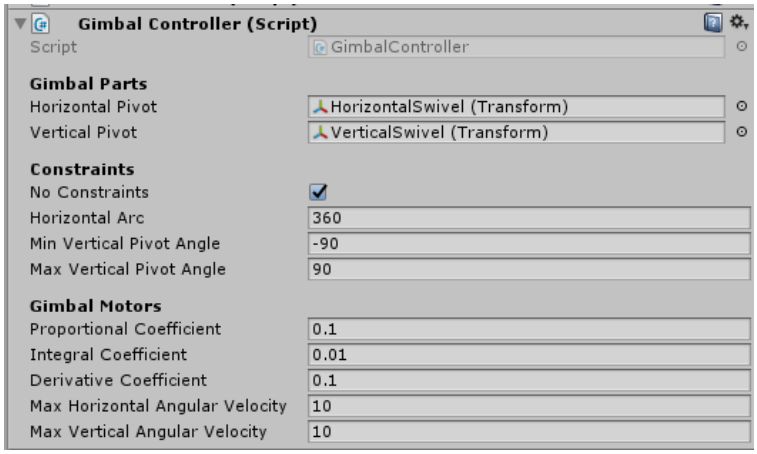
To create an instance of a MountableModule class for a module (which is necessary before a module can be mounted), it is necessary to call the ModuleMount's *AddMountableModule* method, passing the prefab gameobject as a parameter. It is optional to mount the module simultaneously, by setting the parameter *select* to true when calling this method.

9.4 Mounting a Module

To mount a module on a ModuleMount component, it is necessary to have already added it to the mount as an instance of the MountableModule class (see above), which adds it to the ModuleMount's *mountableModules* list (which is a public variable that can be accessed and searched).

To mount a specific module, call the ModuleMount's *OnMountModule* method, passing the index of the module within the ModuleMount's *mountableModules* list as a parameter.

9.5 Gimbal Controllers



Gimbal controllers provide a way for modules (or anything else, including the camera) to rotate and orient themselves toward a location in space, independent of the ship's orientation. The GimbalController component in this kit allows gimbals to rotate in restricted arcs, and exposes a method to automatically rotate the gimbals toward a specified target point in space, which can be either a position that represents the aim of the player, or a target location for automatic turrets and AI.

The IModule interface holds a reference to a GimbalController, which may be left null if the module is not gimballed. Through this reference, subsystem components can control the module's orientation.

To create a gimballed module, first check out the gimbal used by the VehicleCamera object as a reference. To create one from scratch, it is necessary to:

1. **Create a base gameobject and add the GimbalController component to it.** This represents the base transform (which does not move and is always fixed relative to the attachment point of the vehicle).
2. **Add another gameobject as the child of the base gameobject (local position and rotation of zero).** This controls the local horizontal rotation of the gimbal, and is rotated around its local Y axis by the GimbalController. Add this transform to the Horizontal Pivot reference in the GimbalController's inspector.

3. **Add another object, this time as a child of the transform added in Step 2.** This controls the vertical rotation of the turret, and is rotated on its local X axis by the GimbalController. Add this transform to the Vertical Pivot reference in the GimbalController's inspector.

To make a module gimballed:

1. **Add the module component** (a component implementing the IModule interface) to the same transform as the GimbalController component – which must be the root transform of the module prefab.
2. **Perform the module's functionality from the vertically rotating transform (Step 3 above)** – this is the 'end point' of the gimbal machanism. Weapon spawn points, for example, should be added as children of the vertically rotating transform of the gimbal.
3. **Save the gimballed module as a prefab**. It can now be added to the vehicle as a module.

10. Trigger Groups Manager

10.1 Quick Start

The TriggerGroupsManager allows the player to bind different triggerable modules (such as weapon modules) to different triggers, creating more tactical and customizable gameplay. To add this capability, add a TriggerGroupsManager component to the vehicle's root transform.

10.2 Making a Module Triggerable

To make a module triggerable, and to be interacted with on the trigger groups menu, it must inherit the ITriggerable interface. The TriggerGroupsManager will automatically reference it when it is mounted on a module mount.

10.3 Creating/Changing Trigger Groups

An interactive menu has been provided in this kit for allowing the player to dynamically bind triggerable modules to input triggers at runtime, and to create several different trigger groups which can be switched depending on the combat situation, allowing for more tactical combat. This has been added in a single TriggerGroupsMenuController component, which resides in the scene and interacts with any TriggerGroupsManager component found on the vehicle that the scene's GameAgentManager component is currently focused on.

11. Game Agents

11.1 Overview

Game agents represent agents which can enter and exit different vehicles. They consist of player and AI control scripts that derive from the GameAgent base class. Game agents are scene-level components which can assume control of vehicles in the game, and which can be focused on by scene-level components such as the VehicleCamera, HUDManager, PowerManagementMenuController and others.

11.2 GameAgent

The GameAgent base class provides a way to:

- Get the vehicle the agent is in, if any;
- Check if agent is AI;
- Make agent enter a new vehicle;
- Call a 'Death' event, for example by a vehicle component when it is destroyed with the agent inside.

Note that all game agents MUST register with the scene's Game Agent Manager (see below), and must also call the GameAgentManager's OnChangedVehicle method whenever it changes vehicles.

11.3 Game Agent Manager

The GameAgentManager component is a scene-level singleton component for keeping track of all the game agents in the scene. It provides:

- A list of game agents in the scene, which can be accessed and searched;
- A reference to a focused game agent, enabling all of the scene-level components, such as the camera and HUD manager, to maintain focus on a single agent reference.

11.4 Entering/Exiting Vehicles

The GameAgent class contains a public method which can be called to make it enter and exit a vehicle. To exit a vehicle, simply call this method, passing a null reference as the new vehicle reference.

12. AI

This kit features AI for space combat. Included are behaviours for combat (attacking/evading), formation, patrolling and obstacle avoidance. Behaviour Designer has been integrated to facilitate building the space combat AI in your game.

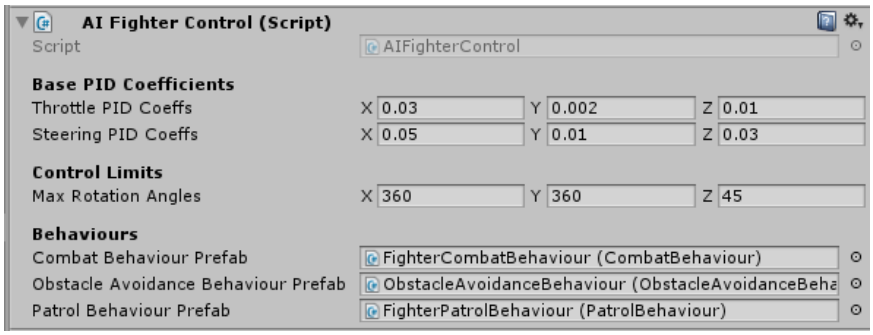
This kit provides a static MoveToward function which takes the space vehicle's SpaceVehicleEngines component and the target destination, and returns control values (-1 to 1 for x, y and z rotation) as well as thrust values (0 – 1) in order to reach the target.

This function is a modified PID controller which enables restriction of pitch and roll for larger ships, and uses 100% physics to control the vehicle.

12.1 IVehicleInput

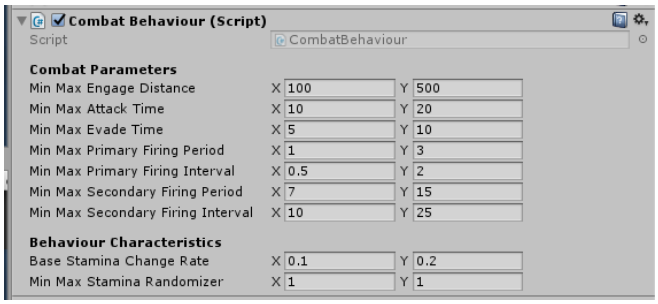
To separate the control of a particular vehicle from the game agent itself, the IVehicleInput interface provides a way to create a control script separately, and run it when the agent enters a vehicle that matches it – each IVehicleInput script contains a reference to a VehicleControlClass enum value, which can be matched to the VehicleControlClass on a Vehicle component.

Here is an example of the AIFighterControl script for controlling a space fighter:



12.2 AI Behaviours

Each vehicle control script (a script implementing the IVehicleInput interface) references a single 'blackboard' (an instance of the non-MonoBehaviour BehaviourBlackboard class) as well as several AI Behaviours that will share this blackboard. These AI Behaviours are components that extend the VehicleControlBehaviour base class, and are stored as prefabs that are instantiated by the vehicle control script and run according to its code. Here is an example of the fighter combat behaviour:



13. HUDs

13.1 Overview

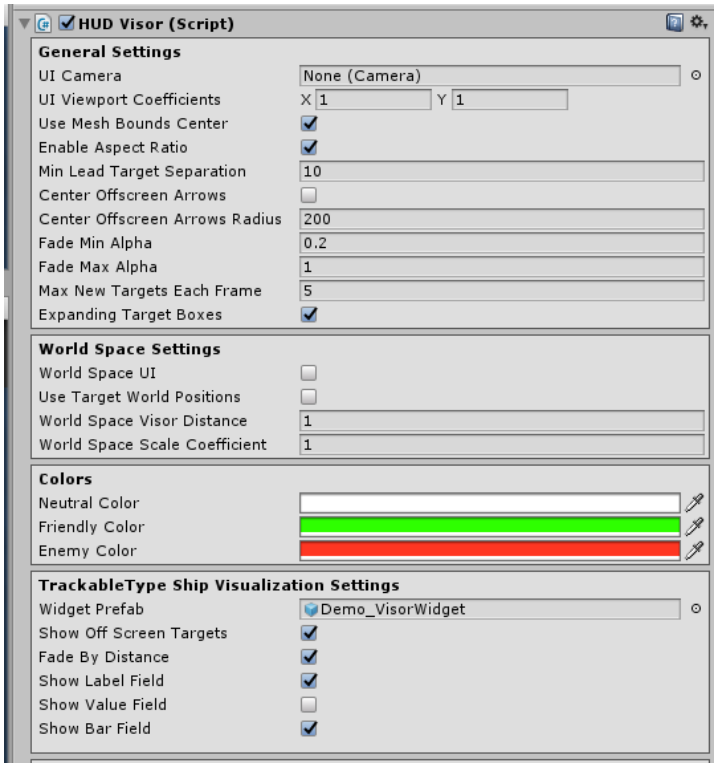
Each Vehicle component contains a reference to a HUD prefab. To achieve better efficiency, only one instance of each HUD prefab is created in the scene, and the scene's HUDManager component is responsible for enabling and positioning the appropriate HUD when the focused game agent (e.g. player) enters a new vehicle (that is, when the OnFocusedVehicleChanged event is called on the Event Manager - see the Event Manager section below).

13.2 The HUD component

The HUD component, which exists on the root transform of every HUD prefab, stores a reference to each of the different HUD-related components (the visor (HUDVisor), 3D radar (Radar3D), dash hologram (HUDHologram) and others, which are present on child transforms in the HUD prefab's hierarchy, and which make up the HUD.

Additionally, it provides a single reference to a Vehicle component that all the HUD-related components can access for displaying information, making the code cleaner and changing vehicles easier.

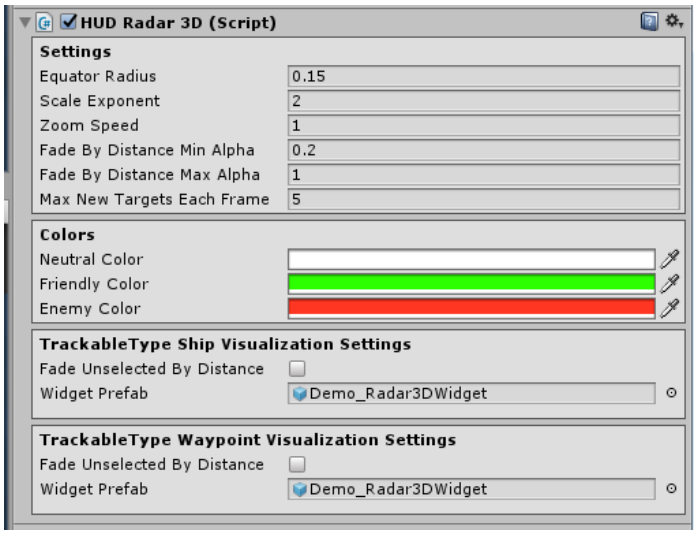
13.3 Visor Target Tracking



The HUDVisor component implements visual target tracking. Essentially, its role is to create new target boxes for each of the targets in radar range, and position these boxes around the camera in the direction of the targets. It also passes data to each target box (an instance of the DemoVisorWidget class) for displaying information about the targets, such as:

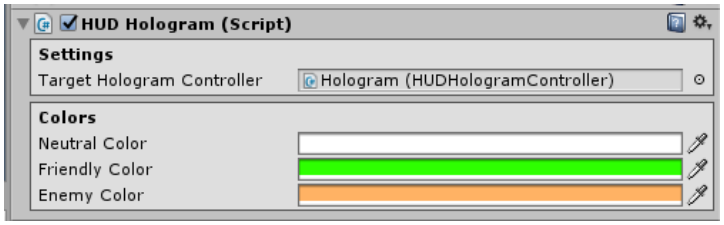
- Label;
- Distance to target;
- Health bars;
- Locking animations for selected target;
- Lead target aiming reticle for selected target;
- Etc.

13.4 3D Radar



The Radar3D component implements a holographic 3D radar for ship cockpits.

13.5 Target Hologram



The HUDHologram component allows a 3D hologram of a target to be displayed. A holographic shader is provided.

14. Event Manager

This kit provides a basic event manager, using Unity Events, which helps to coordinate events that would be difficult to control directly from the point in the code that triggers them. Events are stored as members of the UVCEventType enum:

- **OnFocusedGameAgentChanged**: When the game focuses on a different agent, such as when implementing a spectator mode.
- **OnFocusedVehicleChanged**: When the game is still focused on the same agent, but this agent has entered a different vehicle.
- **OnCameraViewChanged**: Used, for example, for adjusting the HUD display when switching between third person and cockpit view.
- **OnVehicleHit**: Coordinate scoring etc
- **OnVehicleDestroyed**: Coordinate scoring, game mode transition, etc

Events are created and stored in Dictionaries with strings used as the key, but this enum is used to prevent two events being created for the same thing because of a typo in the string.

Any kind of event can be added to this script, as a way to coordinate game flow and scene-wide effects.

15. Integrations

15.1 Behavior Designer

This kit includes a UnityPackage with a behaviour tree that has been created for Behavior Designer. Make sure that you have installed Behavior Designer first, to avoid errors. Then simply import the BehaviorDesignerIntegrationPackage.unitypackage into your project.

Inside the folder that appears, you will find a prefab called 'BehaviorDesignerFighterAI'. This prefab is to be added as a child of the transform where the AI GameAgent component that you wish to use this behavior tree is attached to. The GameAgent will automatically load it upon entering a vehicle of the appropriate VehicleControlClass (a Fighter in this case).

15.2 Rewired

This kit includes an integration with Rewired (in the form of a unitypackage) for mouse and keyboard control. Make sure you have installed Rewired (or the free trial) into your project, then simply import the RewiredIntegrationPackage.unitypackage into your project.

Inside the folder that appears, you will find a prefab called Rewired. This is to be added as a child of the transform that the Player GameAgent component is attached to. The Player GameAgent component will automatically load it upon entering a vehicle of the appropriate VehicleControlClass (a Fighter in this case).

16. That's it!

Thank you for purchasing this asset!

If you have any suggestions, bugs that you've found, or any issues whatsoever with this package, don't hesitate to send an email to contact@vsxgames.com and I will get back to you as soon as possible.

Have fun and good luck with your project!