Space RTS & Arcade Kit

User Manual

This asset provides mechanics for seamless transitions from the RTS view to the chase camera in the third person view – this allows players to take direct control of one of their ships and jump into the action immediately, with one press of a button. Combined with the RTS view, this system offers an intuitive and dexterous way of controlling multiple ships in combat while still keeping the player right in the middle of the battle.

1. Introduction

This document outlines the main components and features of the Space RTS and Arcade Kit. It is an asset which encompasses features of both an RTS-style tactical game and an arcade space sim in a seamless package. This is a fork of the **Space Sim Framework** project and expands upon the tactical aspect of the game by adding an intuitive way to quickly and easily command AI units and keep track of the battle. These features are bound together into a complete gameplay experience, along with progression and save/load functionality, and a seamless Universe to Sector transition.

Special attention was given to making the codebase self documenting and easy to understand, as well as allowing for easy customization and expansion – whichever way you wish to take your project. I believe I've made some good design choices while developing this project – and it becomes apparent as new features often blend in perfectly, like pieces in a puzzle. These design choices are a result of a long iterative process which eventually brought you this asset.

Some of the features include:

- Unity3D Physics based **flight model**
- **Saving and loading** using binary files
- **Sector editor** and sector saving using binary files
- A full feature **tactical/RTS mode** with isometric camera
- **Campaign mode** with player progress tracking
- Interactive **Universe map** with sectors to be selected as a part of campaign mode
- Seamless transitioning from third person to tactical mode and universe map
- Multiple ship ownership
- **AI** with an intricate command system
- Multiple types of **weapons**, including beam and projectile weapons
- AI controlled **turrets** with target prediction
- **User interface** with a 3D heads up display

- Ingame **menu system** with window menus
- Fully keyboard or/and mouse controllable interface and controls
- Sector editor with options for **random sector generation**
- **Ship equipment** such as energy boosters, additional armor and other
- **Player progression**: experience, levels, kill counting
- **Asteroid fields** and dynamic sectors easily expandable with other features
- Modular design allows for decoupling of components and easy addition of new ones
- Maintainable and **scalable codebase**
- Tried and tested design principles and two years of development iterations

If there are features you are curious about, bugs you've encountered or any other questions you may have, feel free to send me an email. Thank you for the purchase of **Space RTS/Arcade Framework**, it would be greatly appreciated if you would leave a rating on the Unity Asset Store.

2. Component overview

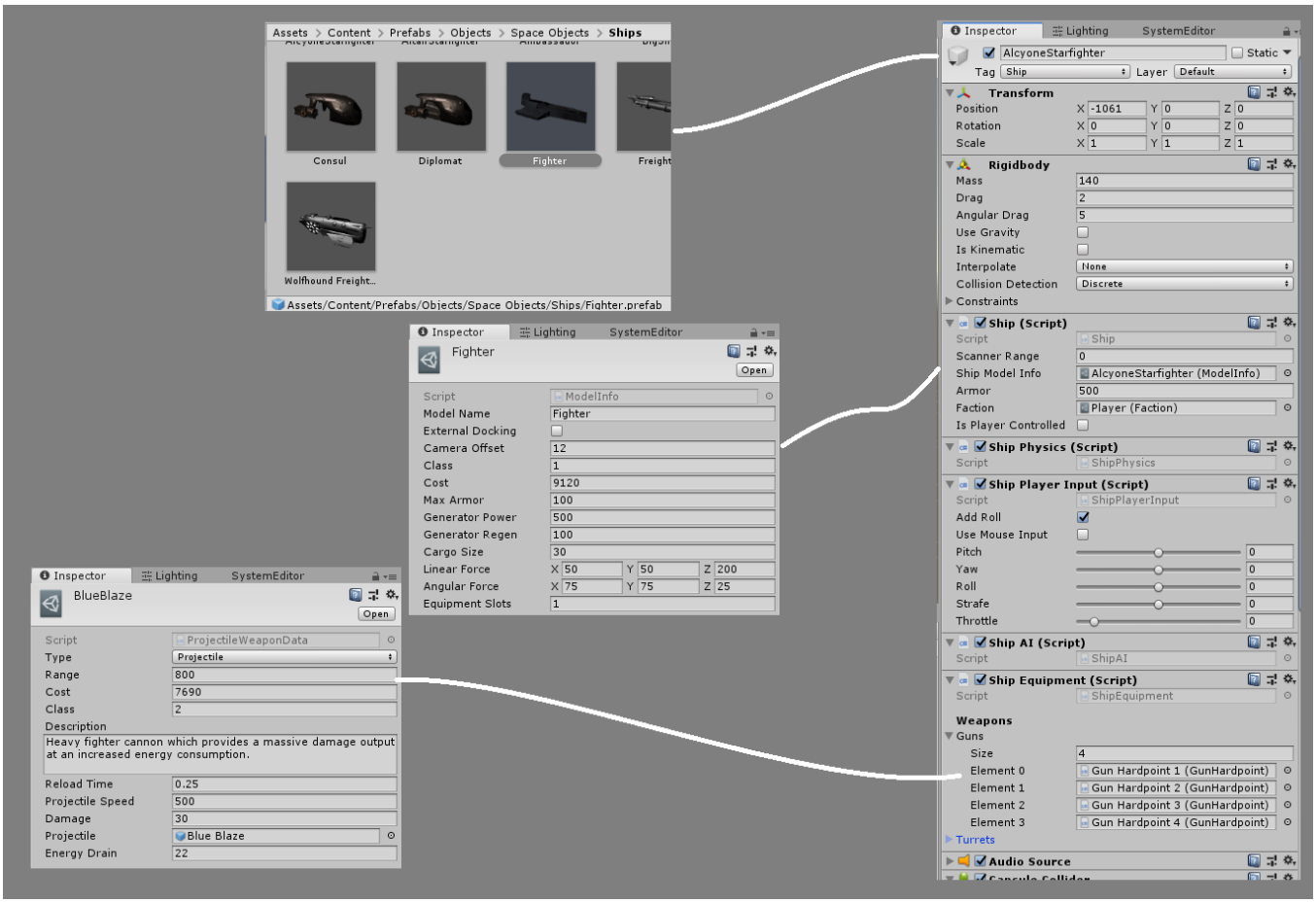
2.1 Ships

The main class which holds the ship properties is **Ship.** It references a **ModelInfo** scriptable object which stores all the ship information, such as agility, maximum speed, cargo space, number of equipment slots and other. Only one ship can have **isPlayerControlled = true.** This is the ship which is currently flown by the player. Since all ships have this flag, it's not difficult to switch ships.

The Ship component holds references to all the main components of the ship, these being:

- **ShipEquipment –** equipment and weapons mounted on the vessel, and their firing mechanic
- **ShipPlayerInput –** player input handling for applying throttle and other ship controls
- **ShipAI –** AI order processing when the ship is not player controller (NPCs)
- **ShipPhysics –** applies linear and angular forces to the ship to move it, depending on AI or player input

Adding ship models is very easy - simply add new models, set up their colliders and weapon hardpoints and attach a *ShipModelInfo* which gives the ship all its characteristics.



Ship prefab configuration and attached components

2.2 Scene setup

Universe scene loads right after the Main Menu. It's the starting point for a player's campaign, where he can see all the sectors and assemble his fleet. Yellow sectors have previously been conquered by the player, and blue ones are not yet successfully taken. The **fleet assembly** screen allows for purchasing and selling of ships. The ships that eventually end up in the player's fleet are spawned in every sector that he wishes to play in.

EmptyFlight scene loads when you select Continue Game on the main menu. It has basic components on top of which the sector and the ships are loaded. All the scriptable objects which hold various ingame objects are exposed by singletons, thus being easily accessible from any class. The **ObjectFactory** can spawn objects at runtime using their names. **UIElements** scriptableObject holds all UI prefabs used by the menu system, similarly. **SectorNavigation** exposes all sector objects such as stations, asteroid fields and jumpgates to all scripts which use them. **MusicController** is also attached to the GameController, and provides 2D music (ambiental music, sound effects and etc.) playback functionality. 3D sounds are attached to individual objects.

2.3 Sectors

Sectors are loaded into the **EmptyFlight** scene using the **SectorLoader** script, from specially written binary files. This makes it easy to have a huge number of sectors without having a separate scene for each of them, as well as to procedurally generate sectors at runtime. Sectors have a difficulty setting which determines how many ships will spawn once the game starts. This parameter should ideally also account for enemy ship types and amount of ships spawned at once – that is left up to you. Sectors are serialized using **SerializableSectorData** which contains:

- Sector size
- Index of the star flare from the ObjectFactory
- Index of the skybox from the ObjectFactory
- List of **SerializableSpaceObjectData** for other types of sector objects
- List of **SerializableGateData** for jumpgates
- List of **SerializableFieldData** for asteroid fields

Each of the serializable data classes contain all necessary information to spawn the object into the sector using the SectorLoader. To create a sector, you will use the **SectorEditor** window which can be opened via Unity's **Windows > Sector Editor** menu. This menu offers options which enable you to save and load sector binary files, while editing them or adding new sectors.

The editor window will automatically switch between the Universe scene for adding or removing sectors and the SectorEditor scene for editing a sector.

Sectors are connected into a **Universe** using a separate Universe file which contains sector position (used to compute adjacent sectors and render the sector in the Universe map), sector sizes and difficulty. The universe file is updated automatically when a sector is exported via the SectorEditor.

2.4 Saving and loading

The system used for saving and loading of player's progress is quite alike the sector saving logic. The **SerializablePlayerData** model contains data relevant to the player's progress. Several main components need to be saved when to accurately keep track of player progression:

- Name, rank, credits
- Owned ships
- List of sectors already taken

The player data is loaded when the Universe Scene is opened and when the EmptyFlight scene is opened (a battle has begun). Player data is saved when the player selects a sector in the Universe and once a battle has been finished. Check out the **SaveGame and LoadGame** scripts for more details. As each profile has one save file to keep all its data, it is easy to keep more profiles separated.

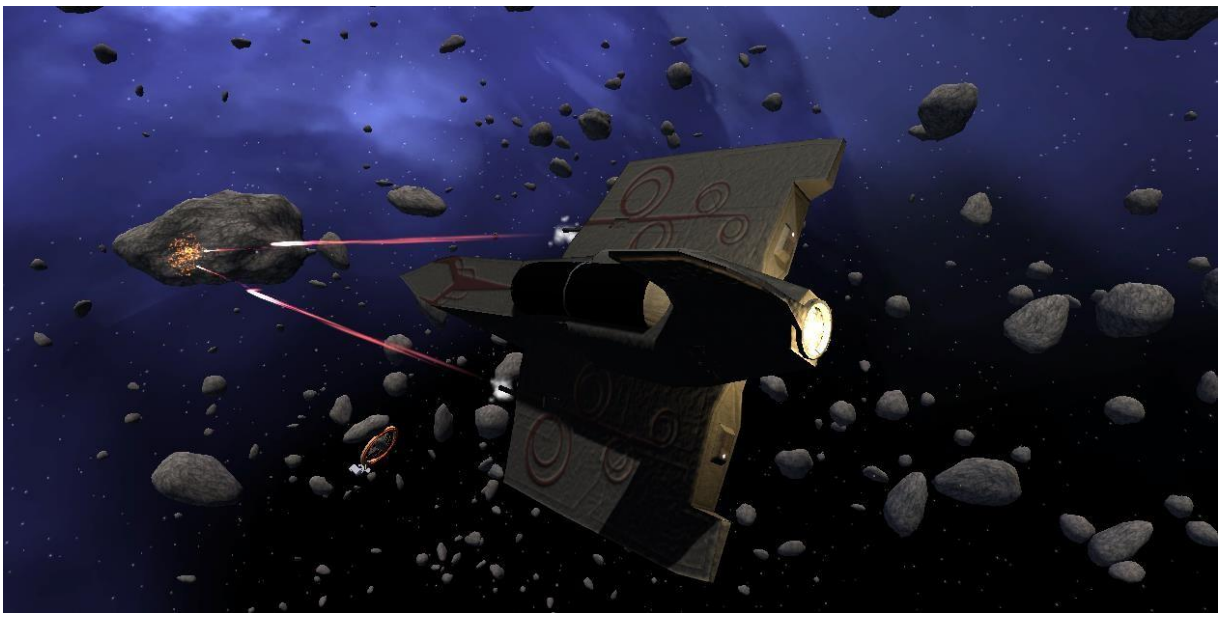
2.5 Weapon systems

There are several weapon systems available, each mounting to a **Hardpoint**. Fixed hardpoints are **GunHardpoints** and revolving hardpoints are **TurretHardpoints.** Each ship with armaments has its weapon hardpoints connected to **ShipEquipment** discussed above. Once a trigger has been pressed the fire command is relayed via the hardpoint to the mounted weapon, which does its own processing depending on the type. Available weapon types:

- Projectile weapons
- Beam weapons

For details on how each weapon performs its firing and preparation, consult their appropriate scripts. Using this system, it is easy to add new weapon types such as area of effect or ammunition based weapons.

Ship turrets are a particular point of interest. A turret has a firing arc and an elevation limit which restricts its firing angles. Turrets have a separate command structure to ships but function as individual units. This means you can issue the following actions to a turret: *none*, *attack target*, *attack all enemies* and *manual control*. For any AI ship (player or other) the default command is *attack all enemies.* Turrets will, when set to *attack all enemies* or *attack target*, aim automatically while computing the required target lead depending on the mounted weapon's projectile speed.



2.6 AI and physics

Ships are using Unity's built in physics by having Rigidbodies and Colliders attached to them. Therefore, they are driven by applying forces instead of by means of direct rotation and translation. As mentioned previously, these forces are applied to each ship by the **ShipPhysics** script. This is rather simple when the ship is player-controlled: angular input (steering) and linear input (throttle & strafing) is applied according to keyboard and mouse input. It's important to

notice that the framework has been built so that the game can be played entirely by mouse or by keyboard only, although it's optimal to combine them. When AI ships are concerned the situation is a bit more complex.

Each AI ship is issued an order; either by the player or by the **AIShipController,** which is executed by the ship's **ShipAI** component. Most of the orders primarily control the throttle (linear input to the ships physics) but all of them use the **ShipSteeringAction** script which uses a PID (Proportional Integral Derivative) controller to rotate the ship towards its destination. The PID controller ensures that the ship will turn towards the target without overrotating and corrects the steering input accordingly. If the ship's steering seems unsatisfactory when on autopilot, play around with the values of the PID controller to see how they affect steering. The implemented orders which come with the framework are:

- Move to position
- Attack target
- Attack all enemies
- Patrol waypoints
- Follow target

2.7 Menu system

This framework features an improved version of the **Ingame Menu Framework** which implements a system of menu windows which are used to interact with various ingame entities. These are controlled by the **CanvasController** which tracks the number of open menus and handles opening/closing of menu windows. There are several basic menus from which specific use-case menus (such as station trade menu, ship purchase menu, sector map, etc…) can be derived from. These building blocks are **SimpleMenu, ScrollMenu and ScrollText,** along with small popup windows for text input, confirmation and amount slider popup. Dive into the CanvasController if you wish to know how the menus are opened, dig through the abovementioned menus to start making your own specific menus.

Apart from the ingame menus, the Canvas is shared by some HUD components, such as speed and throttle indicators, aiming reticule, health indicator and the console output. Some of these components are on a 3D HUD which will lag behind the camera, giving the feeling of spatial presence. Feel free to take it apart and customize it, as well as all the other menus.

2.8 Ship equipment

Along with weapons, the ship mounted equipment items are listed in the **ShipEquipment** and are used to enhance or alter the ship's performance. Equipment items are initialized whenever a ship is spawned or undocked and are updated as long as it is active. This allows you to add additional armor, energy or speed, but also to add unique abilities to ships. The framework features some examples of mountable ship equipment.

This is also interesting due to the ability to use **ship loadouts**, which are sets of equipment and weapons mounted to a certain ship model at its instantiation. One example of use for these loadouts is faction-specific ship loadouts: for example a military vessel would have higher grade equipment compared to a civilian one.

2.9 Factions

Factions are defined in a scriptableObject placed with other Resources. Each faction has a name, a list of friendly and hostile factions and lists of loadouts. By default, if a faction is not friendly or hostile to another, they are neutral. There are only two factions in that come with the kit, the Player and Enemy factions – but more factions are will be very easy to add. This is how it's possible to, for example, add allied non-player controlled units.

2.10 RTS view

The RTS view has a separate input system which primarily uses mouse controls for selecting units and issuing orders. There is a classic drag-n-drop selection rectangle for multiple units and a single unit click selection method. The right mouse button issues orders to selected units. It's also possible to move the camera via keyboard or mouse, adjust its height, and lock it onto a ship for tracking. If one vessel is selected, its status is displayed on the lower left corner of the UI.

A somewhat complex camera script system enables seamless transitions from the RTS view to the chase camera in the third person view – this allows players to take direct control of one of their ships and jump into the action immediately, with one press of a button. Combined with the RTS view, this system offers an intuitive and dexterous way of controlling multiple ships in combat while still keeping the player right in the middle of the battle.

3. Final word

This asset was meant to become a game but there is only so much one person can do in a finite amount of time. Having devoted two years to the development of this game in my free time, I've decided to release this asset to other developers and studios hoping it would shorten

their development to a more manageable span and to allow you to materialize your projects. Developing and releasing a game takes much more than time and skill; it takes passion.

The codebase is well commented and the code is simplified wherever that is possible, however If there are questions regarding some components, feel free to contact me anytime you wish. I will hopefully reply promptly. Thank you once again for the purchase of this asset and I wish you an enjoyable experience with Unity and Space RTS & Arcade Kit!

Kind regards,

Sincress