Space Sim Framework

Documentation

1. Introduction

Welcome to the Space Sim Framework documentation. This passion project of mine has been in development for a year and a half, and is still currently being worked on. I truly believe I've put all my knowledge and experience into this project (which has been inspired by true space classics such as Freelancer and the X series), and as such hope it will be a good base for your own space simulation. The current features of the framework include:

- Unity3D Physics based flight model
- **Saving and loading** using binary files
- **Sector editor** and sector saving using binary files
- **Faction** system with faction relations
- Multiple ship ownership
- **AI** with an intricate command system
- Multiple types of **weapons**, including beam, projectile and missile weapons
- AI controlled **turrets** with target prediction
- **User interface** with a 3D heads up display
- Ship **cockpit** support (interior view)
- Ingame **menu system** with window menus
- Fully keyboard or/and mouse controllable interface and controls
- Full feature **space stations** with animated docks and cargo/ship/equipment dealers
- Distinction between small craft (**fighters**) and **capital ships** (transports and warships)
- **Sector and universe maps** accessible and interactable through menus
- Sector editor with options for **random sector generation** (which could be done at runtime!)
- **Ship equipment** such as energy boosters, additional armor and other
- **Jumpgates** and a full-featured player persistence (autosave and autoload)
- **Player progression**: experience, levels, kill counting, dynamic faction reputation
- **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**

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

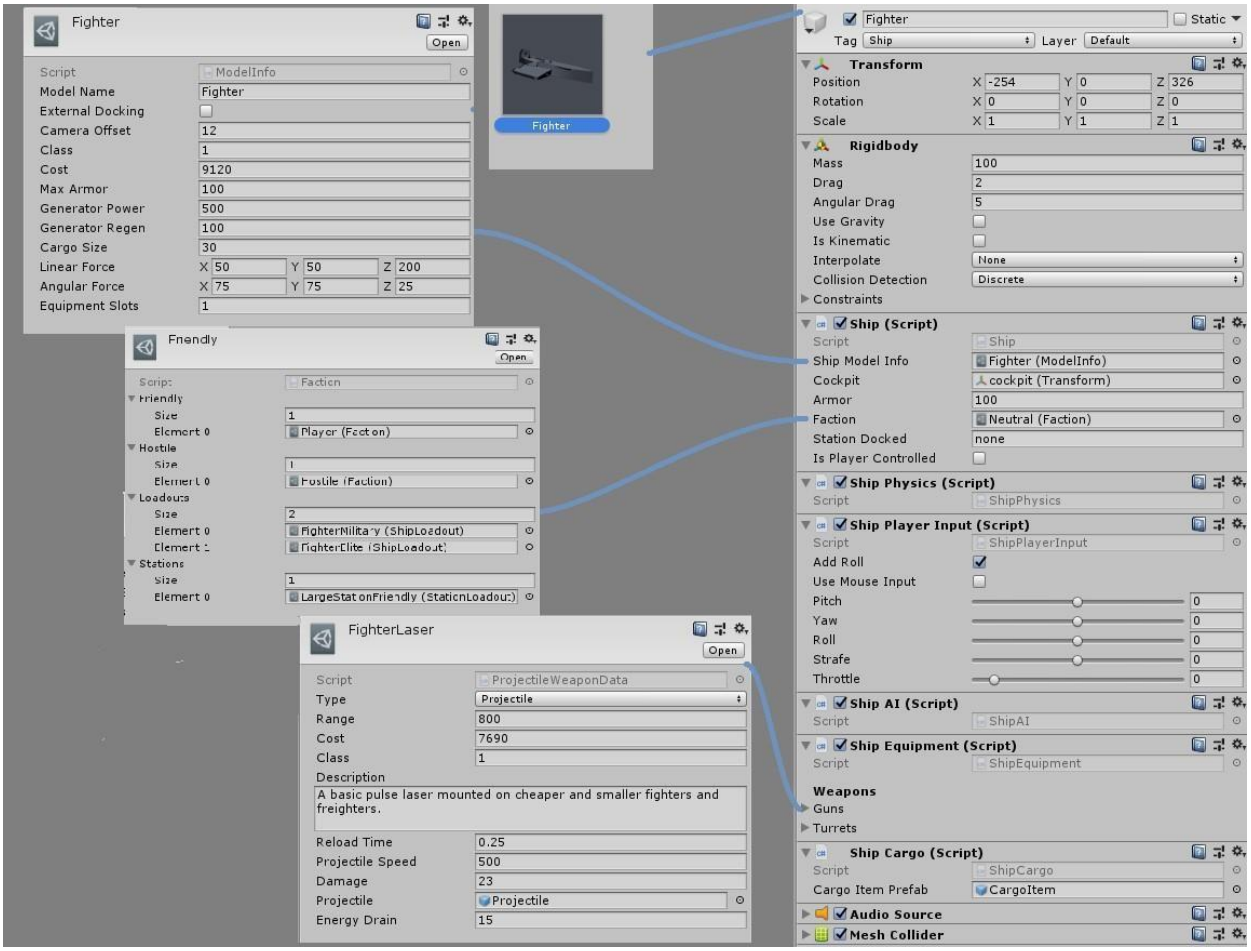
2. Component overview

2.1 Ships

Every ship prefab has similar components attached to it, for reasons of uniformity. 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:

- **ShipCargo –** keeps track of the cargo hold properties and items stored in the ship's hold
- **ShipEquipment –** equipment and weapons mounted on the vessel, and their firing mechanic
- **ShipMovementInput –** 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



Ship prefab configuration and attached components

2.2 Stations

Stations use a somewhat similar configuration to ships. The main component is, obviously, **Station,** and its primary function, apart from referencing station functionality, is handling ship docking and undocking.

Ships can request docking, and if allowed, either dock to the station using the dock entrance (for small ships) or moor externally to one of the station's mooring points. This component handles most of the related functionality such as approach control, animations, tracking of docked ships and similar.

It also has a reference to the **StationLoadout,** which similarly to the ModelInfo mentioned above, holds information on station sold wares and equipment. Stations also have a **StationDealer** which is populated with wares from the StationLoadout scriptable object. These are ships, equipment, weapons and cargo wares. Prices for cargo wares are generated randomly from a predefined interval each time the game starts (or scene loads).

2.3 Scene setup

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.

The main component which provides access to most common game features is **GameController**. It has references to the **ObjectFactory** which can spawn objects at runtime using their names. **UIElements** scriptableObject holds all UI prefabs used by the menu system, similarly. **AIShipController** ensures that when an AI ship has completed its order, it receives a new one. **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. The **MissionControl** script controls the player's current mission and displays data on the UI. More about the mission system later.

StartScenario scene loads when the player selects *Start New Game* from the main menu. It's an EmptyFlight scene with a player ship and some credits assigned to the player. This scene is only loaded when the game is started and could be replaced by a savefile if you wish to do so.

2.4 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 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 **SerializableStationData** for stations
- 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. It is recommendable to use this functionality in the **SectorEditor** scene because it only contains the sector data.

Sectors are connected into a **Universe** using a separate Universe file which contains sector connections (adjacent sectors), the owner faction of the sector and their influence. The universe file is updated automatically when a sector is exported via the SectorEditor.

Attention: Although the SectorEditor does a lot of work automatically, there are several things you need to do yourself before Exporting a sector to file:

1. For each JumpGate, Station and Asteroid field in the sector, enter an ID into their main component's ID field (Jumpgate.cs, Station.cs and AsteroidField.cs) in the format **sX_jg0N, sX_st0N and sX_f0N** respectively, where X is the sector unique number and N being the ordinal number of the entity of one type in the sector. For example, in a sector wih two stations, three jumpgates and an asteroid field with the selected number 7, IDs should be *s07_st01, s07_st02, s07_jg01, s07_jg02, s07_jg03, s07_f01*
2. Edit the sector coordinates on the **Universe Map**: both X and Y can be between -50 and 50. This dictates where the sector is shown on the ingame Universe Map menu.

2.5 Saving and loading

The system used for saving and loading of player's progress is quite similar to 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:

-
- general player data
- player's in-sector ships (ships located in the current sector where the game is saved)
- player's out-of-sector ships (in other sectors in the Universe)
- the current mission, if one exists

By default, during gameplay, the game is saved whenever a player jumps to the next sector. This is necessary because once the sector is loaded into the EmptyFlight scene, the player's autosave file is also loaded (as there is no good built-in way to transfer objects between scenes in Unity). Check out the **SaveGame and LoadGame** scripts for more details.

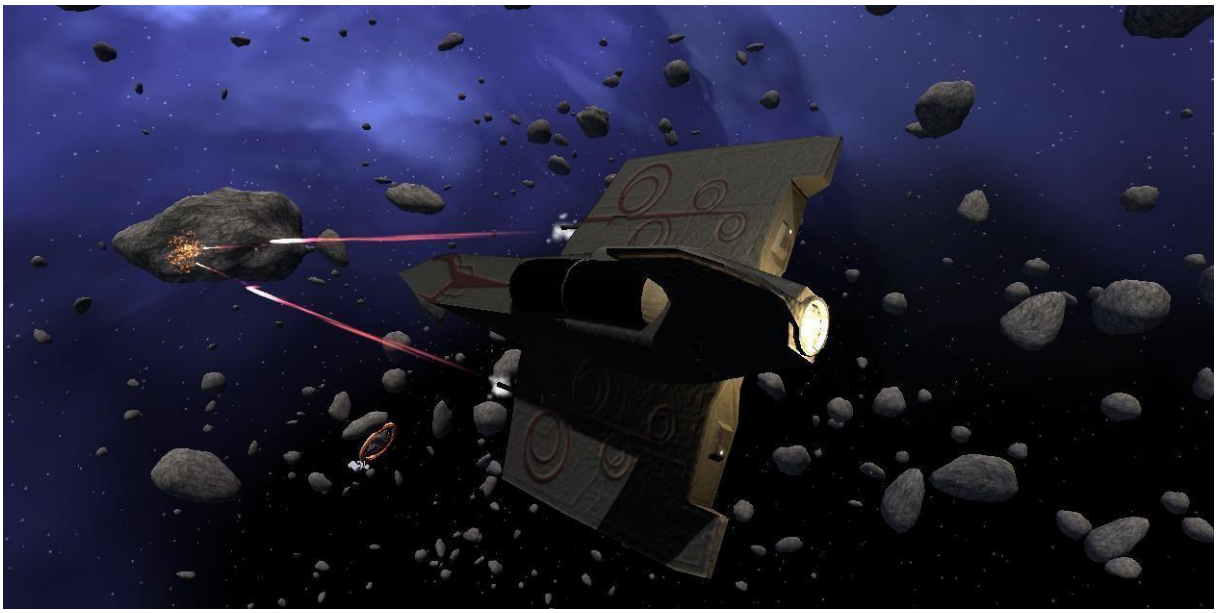
2.6 Weapon systems

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
- Missile 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.



2.7 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
- Trade in current sector
- Dock to station/jumpgate
- Follow target

2.8 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.

2.9 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.

2.10 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. A jumpgate will periodically spawn ships and depending on the sector owner faction their ships will be more common. Therefore while spawning a ship of a certain faction, the faction ship loadouts array is checked and a loadout is picked, if one exists. As for station loadouts, they come in while generating sector data in the SectorEditor. Neither are obligatory to have for each faction, as default loadouts will spawn when no faction specific loadouts exist.

2.11 Missions

Stations allow players to accept jobs of various types. These missions are limited to one sector as currently implemented, but are saved whenever the player leaves the sector in which the mission was taken. There are several types of jobs available: assassination, patrol, cargo delivery and courier missions. Each mission is tied to the faction which issues it (determined by the owner faction of the station). Jobs are generated whenever the job board is opened, however in certain cases a mission cannot be initialized – that's when the **GenerateMissionData** method will fail and notify that the mission won't be created. This can be the case, for example, when there is only one station in the sector and a cargodelivery mission can't be initialized.

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 expended 1.5 years developing 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 Sim Framework!

Kind regards,

Sincress