Naev Development Manual

Version 0.12.0-beta.2+dev

Naev DevTeam

November 30, 2024

Contents

Sections marked with naev are specific to the Sea of Darkness default Naev universe.

Chapter 1

Introduction

Welcome to the Naev development manual! This manual is meant to cover all aspects of Naev development, including both the engine and the lore of the Naev base scenario known as **Sea of Darkness**. It is currently a work in progress. The source code for the manual can be found on the naev github[1] with pull requests and issues being welcome.

The document is split into two parts: the first deals with the Naev engine and how to implement and work with it. The second part deals with the Lore of the Naev base scenario known as **Sea of Darkness**.

1.1 What is Naev?

Naev started development circa 2004[2] as an attempt to make an open source clone of Escape Velocity (Classic) that would run on Linux. While the engine itself was meant to be a modern clone of the Escape Velocity engine, the game scenario itself was meant to be unique. The name NAEV (later to become Naev) stood for Not Another Escape Velocity. It is now used as a proper known and has not been changed to confuse users.

Over the time and with the come and go of many contributors, Naev has grown into an advanced game engine with a complex base scenario featuring many new mechanics and features not seen in games of the same genre. While still far from done, since version 0.10.0 plugin support has been added, and initial work has begun on creating this in-depth guide to Naev development in hopes that more people join in on the exciting project.

[1] https://github.com/naev/naev/tree/main/docs/manual

[2] This is roughly 2 years after the release of Escape Velocity Nova on Mac OS.

Part I

Naev Engine

Chapter 2

Introduction to the Naev Engine

While this document does cover the Naev engine in general, many sections refer to customs and properties specific to the **Sea of Darkness** default Naev universe. These are marked with naev.

2.1 Getting Started

The Naev engine explanations assume you have access to the Naev data. This can be either from downloading the game directly from a distribution platform, or getting directly the naev source code[1]. Either way it is possible to modify the game data and change many aspects of the game. It is also possible to create plugins that add or replace content from the game without touching the core data to be compatible with updates.

Operating System	Data Location
Linux	`/usr/share/naev/dat`
Mac OS X	`/Applications/Naev.app/Contents/Resources/dat`
Windows	`%ProgramFiles(x86)%\Naev\dat`

Most changes will only take place when you restart Naev, although it is possible to force Naev to reload a mission or event with `naev.missionReload` or `naev.eventReload`.

[1] https://github.com/naev/naev

2.2 Plugins

Naev supports arbitrary plugins. These are implemented with a virtual filesystem based on PHYSFS[2]. The plugin files are therefore "combined" with existing files in the virtual filesystem, with plugin files taking priority. So if you add a mission in a plugin, it gets added to the pool of available missions. However, if the mission file has the same name as an existing mission, it will overwrite it. This allows the plugin to change core features such as boarding or communication mechanics or simply add more compatible content.

Plugins are found at the following locations by default, and are automatically loaded if found.

Operating System	Data Location
Linux	`~/.local/share/naev/plugins`
Mac OS X	`~/Library/Application Support/org.naev.Naev/plugins`
Windows	`%APPDATA%\naev\plugins`

Note that plugins can use either a directory structure or be compressed as zip files (while still having the appropriate directory structure). For example, it is possible to add a single mission by creating a plugin with the follow structure:

```
plugin.xml
missions/
   my_mission.xml
```

This will cause `my_mission.xml` to be loaded as an extra mission. `plugin.xml` is a plugin-specific file which would contain information on plugin name, authors, version, description, compatibility, and so on.

Plugins are described in detail in Chapter 3.

[2] https://icculus.org/physfs/

Chapter 3

Plugin Framework

Plugins are user-made collections of files that can add or change content from Naev. They can be seen as a set of files that can overwrite core Naev files and add new content such as missions, outfits, ships, etc. They are implemented with PHYSFS[1] that allows treating the plugins and Naev data as a single "combined" virtual filesystems. Effectively, Naev will see plugin files as part of core data files, and use them appropriately.

Plugins are found at the following locations by default, and are automatically loaded if found.

Operating System	Data Location
Linux	`~/.local/share/naev/plugins`
Mac OS X	`~/Library/Application Support/org.naev.Naev/plugins`
Windows	`%APPDATA%\naev\plugins`

Plugins can either be a directory structure or compressed into a single `zip` file which allows for easier distribution.

3.1 Directory Structure

Naev plugins and data use the same directory structure. It is best to open up the original data to see how everything is laid. For completeness, the main directories are described below:

- `ai/`: contains the different AI profiles and their associated libraries.
- `asteroids/`: contains the different asteroid types and groups in different directories.
- `commodities/`: contains all the different commodity files.

[1] `https://icculus.org/physfs/`

- `damagetype/`: contains all the potential damage types.
- `difficulty/`: contains the different difficulty settings.
- `effects/`: contains information about effects that can affect ships.
- `events/`: contains all the events.
- `factions/`: contains all the factions and their related Lua functionality.
- `glsl/`: contains all the shaders. Some are critical for basic game functionality.
- `gui/`: contains the different GUIs
- `map_decorator/`: contains the information of what images to render on the map.
- `missions/`: contains all the missions.
- `outfits/`: contains all the outfits.
- `scripts/`: this is an optional directory that contains all libraries and useful scripts by convention. It is directly appended to the Lua path, so you can `require` files in this directory directly without having to prepend `scripts.`.
- `ships/`: contains all the ships.
- `slots/`: contains information about the different ship slot types.
- `snd/`: contains all the sound and music.
- `spfx/`: contains all the special effects. Explosions are required by the engine and can not be removed.
- `spob/`: contains all the space objects (planets, stations, etc.).
- `spob_virtual/`: contains all the virtual space objects. These mainly serve to modify the presence levels of factions in different systems artificially.
- `ssys/`: contains all the star systems.
- `tech/`: contains all the tech structures.
- `trails/`: contains all the descriptions of ship trails that are available and their shaders.
- `unidiff/`: contains all the universe diffs. These are used to create modifications to the game data during a playthrough, such as adding spobs or systems.

In general, recursive iteration is used with all directories. This means you don't have to put all the ship xml files directly it `ships/`, but you can use subdirectories. Furthermore, in order to avoid collision between plugins, it is highly recommended to use a subdirectory with the plugin name. So if you want to define a new ship called `Stardragon`, you would put the xml file in `ships/my_plugin/stardragon.xml`.

Furthermore, the following files play a critical role:

- `AUTHORS`: contains author information about the game.
- `VERSION`: contains version information about the game.

- `autoequip.lua`: used when the player presses autoequip in the equipment window.
- `board.lua`: used when the player boards a ship.
- `comm.lua`: used when the player hails a ship.
- `common.lua`: changes to the Lua language that are applied to all scripts.
- `intro`: the introduction text when starting a new game.
- `loadscreen.lua`: renders the loading screen.
- `rep.lua`: internal file for the console. Do not modify!!
- `rescue.lua`: script run when the game detects the player is stranded, such as they have a non-spaceworthy ship and are stuck in an uninhabited spob.
- `save_updater.lua`: used when updating saves to replace old outfits and licenses with newer ones.
- `start.xml`: determines the starting setting, time and such of the game.

Finally, plugins have access to an additional important file known as `plugin.xml` that stores meta-data about the plugin itself and compatibility with Naev versions. This is explained in the next section.

3.2 Plugin Meta-Data `plugin.xml`

The `plugin.xml` file is specific to plugins and does not exist in the base game. A full example is shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin name="My Plugin">
 <author>Me</author>
 <version>1.0</version>
 <description>A cool example plugin.</description>
 <compatibility>^0\.10\..*</compatibility>
 <priority>3</priority>
 <source>https://source</source>
</plugin>
```

The important fields are listed below:

- `name`: attribute that contains the name of the plugin. This is what the player and other mods will see and use to reference the plugin.
- `author`: contains the name of the author(s) of the plugin.
- `version`: contains the version of the plugin. This can be any arbitrary string.
- `description`: contains the description of the plugin. This should be easy to understand for players when searching for plugins.
- `compatibility`: contains compatibility information. Specifically, it must be a regex string that will be tested against the current Naev

string. Something like `^0\.10\..*` will match any version string that starts with `"0.10."`. Please refer to a regular expression guide such as regexr[2] for more information on how to write regex.
- `priority`: indicates the loading order of the plugin. The default value is 5 and a lower value indicates higher priority. Higher priority allows the plugin to overwrite files of lower priority plugins. For example, if two plugins have a file `missions/test.lua`, the one with the lower priority would take preference and overwrite the file of the other plugin.
- `source`: points to where a newer version can be obtained or downloaded. This could be a website or other location.

Furthermore, it is also possible to use regex to hide files from the base game with `<blacklist>` nodes. For example, `<blacklist>^ssys/.*\.xml</blacklist>` would hide all the XML files in the `ssys` directory. This is especially useful when planning total conversions or plugins that modify significantly the base game, and you don't want updates to add new files you have to hide. By using the appropriate blacklists, you increase compatibility with future versions of Naev. Furthermore, given the popularity of *total conversion*-type plugins, you can use the `<total_conversion/>` tag to apply a large set of blacklist rules which will remove all explicit content from Naev. This means you will have to define at least a star system, a spob, and a flyable ship for the game to run.

In additiona to the blacklist, a whitelist can also be defined with `<whitelist>`, which takes priority over the blacklist. In other words, whitlelist stuff will ignore the blacklist. *Total conversions* automatically get a few critical files such as the `settings.lua` event included, although they can still be overwritten.

3.3 Plugin Repository

Naev has a plugin repository[3] which tries to centralize know plugins. To add your plugin, please create a pull request on the repository. This repository contains only the minimum information of the plugins necessary to be able to download and look up the rest of the information.

3.4 Tips and Tricks

This section includes some useful tricks when designing plugins.

[2] https://regexr.com/
[3] https://github.com/naev/naev-plugins

3.4.1 Making Compatible Changes

While plugins can easily overwrite files, there are times you may not wish to do that, as replacing the same file as another plugin will lead to conflicts. To avoid replacing files, it is possible to use the *Universe Diff* (`unidiff`) system (TODO add section). Let us consider a motivating example where we want to simply add an outfit, and add it to an existing tech group, but not replace the file so it can work with other plugins. This can be done by creating an event that automatically applies the unidiff when the player loads the game.

Assume that we have a tech group called ”Base Tech Group”, and an outfit called ”My Outfit”. We wish to add ”My Outfit” to ”Base Tech Group” without replacing the file. To do this, first we define the unidiff that adds ”My Outfit” to ”Base Tech Group” as below:

```
<?xml version="1.0" encoding="UTF-8"?>
<unidiff name="my_plugin_unidiff">
 <tech name="Base Tech Group">
  <add>My Outfit</add>
 </tech>
</unidiff>
```

The above file should be saved to `unidiff/my_plugin_unidiff.xml`, and will simply add the ”My Outfit” to ”Base Tech Group”. However, this unidiff will be disabled by default, so we’ll have to enable it with an event such as below:

```
--[[
<?xml version='1.0' encoding='utf8'?>
<event name="My Plugin Start">
 <location>load</location>
 <chance>100</chance>
</event>
--]]
function create ()
   local diffname = "my_plugin_unidiff"
   if not diff.isApplied(diffname) then
      diff.apply(diffname)
   end
   evt.finish()
end
```

The above file should be saved somewhere in `events/`, such as `events/my_plugin_start.lua` and will simply apply the unidiff `"my_plugin_unidiff"` if it is not active, thus adding ”My Outfit” to ”Base Tech Group” without overwriting the file.

The same technique can be done to add new systems or any other feature supported by the unidiff system (see section TODO). For example, you can add new systems normally, but then add the jumps to existing systems in a unidiff so you do not have to overwrite the original files.

3.5 Extending Naev Functionality

This section deals with some of the core functionality in Naev that can be extended to support plugins without the need to be overwritten. Extending Naev functionality, in general, relies heavily on the Lua[4] using custom bindings that can interact with the Naev engine.

A full overview of the Naev Lua API can be found at naev.org/api[5] and is out of the scope of this document.

3.5.1 Adding News

News is controlled by the dat/events/news.lua[6] script. This script looks in the dat/events/news/[7] for news data that can be used to create customized news feeds for the player. In general, all you have to do is created a specially formatted news file and chuck it in the directory for it to be used in-game.

When the player loads a game, the news script goes over all the news data files and loads them up. Afterwards, each time the player lands, it creates a dynamic list of potential news based on the faction and characteristics of the landed spob. Afterwards, it randomly samples from the news a number of times based on certain criteria. News is not refreshed entirely each time the player lands, instead it is slowly updated over time based on a diversity of criteria. When new news is needed, the script samples from the dynamic list to create it. Thus it tends to slowly evolve as the player does things.

Let us take a look at how the news data files have to be formatted.

At the core, each news data file has to return a function that returns 4 values:

1. The name of the faction the news should be displayed at, or `"Generic"` for all factions with the `generic` tag.
2. The headers to use for the faction. Set to nil if you don't want to add more header options.
3. The greetings to use for the faction. Set to nil if you don't want to add more greeting options.
4. A list of available articles for the faction.

Let us look at a minimal working example with all the features:

```
local head = {
   _("Welcome to Universal News Feed.")
}
```

[4] https://www.lua.org/
[5] https://naev.org/api
[6] https://github.com/naev/naev/blob/main/dat/events/news.lua
[7] https://github.com/naev/naev/tree/main/dat/events/news

```
local greeting = {
   _("Interesting events from around the universe."),
}
local articles = {
    {
      head = N_([[Naev Dev Manual Released!]]),
      body = _([[The Naev Development Manual was released after a long
          time in development. "About time" said an impatient user.]]),
    },
}
return function ()
   return "Independent", head, greeting, articles
end
```

The above example declares 3 tables corresponding to the news header (`head`), news greeting (`greeting`), and articles (`articles`). In this simple case, each table only has a single element, but it can have many more which will be chosen at random. The script returns a function at the bottom, that returns the faction name, `"Independent"` in this case, and the four tables. The function will be evaluated each time the player lands and news has to be generated, and allows you to condition what articles are generated based on all sorts of criteria.

Most of the meat of news lies in the articles. Each article is represented as a table with the following elements:

1. `head`: Title of the news. Must be an untranslated string (wrap with `N_()`)
2. `body`: Body text of the news. Can be either a function or a string. In case of being a function, it gets evaluated.
3. `tag` (optional): Used to determine if a piece of news is repeated. Defaults to the head, but you can define multiple news with the same tag to make them mutually exclusive.
4. `priority` (optional): Determines how high up the news is shown. Smaller values prioritize the news. Defaults to a value of 6.

As an alternative, it is also possible to bypass the news script entirely and directly add news with `news.add`[8]. This can be useful when adding news you want to appear directly as a result of in-game actions and not have it randomly appear. However, do note that not all players read the news and it can easily be missed.

3.5.2 Adding Bar NPCs

TODO

[8] https://naev.org/api/modules/news.html#add

3.5.3 Adding Derelict Events needs

TODO add engine support

3.5.4 Adding Points of Interest needs

TODO

3.5.5 Adding Personalities needs

TODO

Chapter 4

Missions and Events

Naev missions and events are written in the Lua Programming Language[1]. In particular, they use version 5.1 of the Lua programming language. While both missions and events share most of the same API, they differ in the following ways:

- **Missions:** Always visible to the player in the info window. The player can also abort them at any time. Missions are saved by default. Have exclusive access to the `misn` library and are found in `dat/missions/`.
- **Events:** Not visible or shown to the player in any way, however, their consequences can be seen by the player. By default, they are *not saved to the player savefile*. If you want the event to be saved you have to explicitly do it with `evt.save()`. Have exclusive access to the `evt` library and are found in `dat/events/`.

The general rule of thumb when choosing which to make is that if you want the player to have control, use a mission, otherwise use an event. Example missions include cargo deliveries, system patrols, etc. On the other hand, most events are related to game internals and cutscenes such as the save game updater event (`dat/events/updater.lua`[2]) or news generator event (`dat/events/news.lua`[3]).

A full overview of the Naev Lua API can be found at naev.org/api[4] and is out of the scope of this document.

[1] `https://www.lua.org`
[2] `https://github.com/naev/naev/blob/main/dat/events/updater.lua`
[3] `https://github.com/naev/naev/blob/main/dat/events/news.lua`
[4] `https://naev.org/api`

4.1 Mission Guidelines

This following section deals with guidelines for getting missions included into the official Naev repository[5]. These are rough guidelines and do not necessarily have to be followed exactly. Exceptions can be made depending on the context.

1. **Avoid stating what the player is feeling or making choices for them.** The player should be in control of themselves.
2. **There should be no penalties for aborting missions.** Let the player abort/fail and try again.

4.2 Getting Started

Missions and events share the same overall structure in which there is a large Lua comment at the top containing all sorts of meta-data, such as where it appears, requirements, etc. Once the mission or event is started, the obligatory `create` function entry point is run.

Let us start by writing a simple mission header. This will be enclosed by long Lua comments `--[[` and `--]]` in the file. Below is our simple header.

```
--[[
<mission name="My First Mission">
 <unique />
 <avail>
  <chance>50</chance>
  <location>Bar</location>
 </avail>
</mission>
--]]
```

The mission is named "My First Mission" and has a 50% chance of appearing in any spaceport bar. Furthermore, it is marked unique so that once it is successfully completed, it will not appear again to the same player. For more information on headers refer to Section 4.3.1.

Now, we can start coding the actual mission. This all begins with the `create ()` function. Let us write a simple one to create an NPC at the Spaceport Bar where the mission appears:

```
function create ()
   misn.setNPC( _("A human."),
        "neutral/unique/youngbusinessman.webp",
        _("A human wearing clothes.") )
end
```

[5] https://github.com/naev/naev

The create function in this case is really simple, it only creates a single NPC with `misn.setNPC`. Please note that only a single NPC is supported with `misn.setNPC`, if you want to use more NPC you would have to use `misn.npcAdd` which is much more flexible and not limited to mission givers. There are two important things to note:

1. All human-readable text is enclosed in `_()` for translations. In principle, you should always use `_()` to enclose any text meant for the user to read, which will allow the translation system to automatically deal with it. For more details, please refer to Section 4.3.6.
2. There is an image defined as a string. In this case, this refers to an image in `gfx/portraits/`. Note that Naev uses a virtual filesystem and the exact place of the file may vary depending on where it is set up.

With that set up, the mission will now spawn an NPC with 50% chance at any Spaceport Bar, but they will not do anything when approached. This is because we have not defined an `accept()` function. This function is only necessary when either using `misn.npcAdd` or creating mission computer missions (see Section 4.3.2). So let us define that which will determine what happens when the NPC is approached as follows:

```
local vntk = require "vntk"
local fmt = require "format"

local reward = 50e3 -- This is equivalent to 50000, and easier to read

function accept ()
   -- Make sure the player has space
   if player.pilot():cargoFree() < 1 then
      vntk.msg( _("Not Enough Space"),
           _("You need more free space for this mission!") )
      return
   end

   -- We get a target destination
   mem.dest, mem.destsys = spob.getS( "Caladan" )

   -- Ask the player if they want to do the mission
   if not vntk.yesno( _("Apples?"),
        fmt.f(_("Deliver apples to {spb} ({sys})?"),
             {spb=mem.dest,sys=mem.destsys}) ) then
      -- Player did not accept, so we finish here
      vntk.msg(_("Rejected"),_("Your loss."))
      misn.finish(false) -- Say the mission failed to complete
      return
   end

   misn.accept() -- Have to accept the mission for it to be active
```

```
    -- Set mission details
    misn.setTitle( _("Deliver Apples") )
    misn.setReward( fmt.credits( reward ) )
    local desc = fmt.f(_("Take Apples to {spb} ({sys})."),
         {spb=mem.dest,sys=mem.destsys}) )
    misn.setDesc( desc )

    -- On-screen display
    misn.osdCreate( _("Apples"), { desc } )

    misn.cargoAdd( "Food", 1 ) -- Add cargo
    misn.markerAdd( mem.dest ) -- Show marker on the destination

    -- Hook will trigger when we land
    hook.land( "land" )
end
```

This time it's a bit more complicated than before. Let us try to break it down a bit. The first line includes the `vntk` library, which is a small wrapper around the `vn` Visual Novel library (explained in Section 4.3.12). This allows us to show simple dialogues and ask the player questions. We also include the `format` library to let us format arbitrary text, and we also define the local reward to be 50,000 credits in exponential notation.

The function contains of 3 main parts:

1. We first check to see if the player has enough space for the apples with `player.pilot():cargoFree()` and display a message and return from the function if not.
2. We then ask the player if then ask the player if they want to deliver apples to **Caladan** and if they don't, we give a message and return from the function.
3. Finally, we accept the mission, adding it to the player's active mission list, set the details, add the cargo to the player, and define a hook on when the player lands to run the final part of the mission. Functions like `misn.markerAdd` add markers on the spob the player has to go to, making it easier to complete the mission. The On-Screen Display (OSD) is also set with the mission details to guide the player with `misn.osdCreate`.

Some important notes.

- We use `fmt.f` to format the strings. In this case, the `{spb}` will be replaced by the `spb` field in the table, which corresponds to the name of the `mem.dest` spob. This is further explained in Section 4.3.7.
- Variables don't get saved unless they are in the `mem` table. This table gets populated again every time the save game gets loaded. More details in Section 4.3.3.
- You have to pass function names as strings to the family of `hook.*`

functions. More details on hooks in Section 4.3.5.

Now this gives us almost the entirety of the mission, but a last crucial component is missing: we need to reward the player when they deliver the cargo to **Caladan**. We do this by exploiting the `hook.land` that makes it so our defined `land` function gets called whenever the player lands. We can define one as follows:

```
local neu = require "common.neutral"
function land ()
   if spob.cur() ~= mem.dest then
      return
   end

   vn.msg(_("Winner"), _("You win!"))
   neu.addMiscLog( _("You helped deliver apples!") )
   player.pay( reward )
   misn.finish(true)
end
```

We can see it's very simple. It first does a check to make sure the landed planet `spob.cur()` is indeed the destination planet `mem.dest`. If not, it returns, but if it is, it'll display a message, add a message to the ship log, pay the player, and finally finish the mission with `misn.finish(true)`. Remember that since this is defined to be a unique mission, once the mission is done it will not appear again to the same player.

That concludes our very simple introduction to mission writing. Note that it doesn't handle things like playing victory sounds, nor other more advanced functionality. However, please refer to the full example in Section 4.6 that covers more advanced functionality.

4.3 Basics

In this section we will discuss basic and fundamental aspects of mission and event developments that you will have to take into account in almost all cases.

4.3.1 Headers

Headers contain all the necessary data about a mission or event to determine where and when they should be run. They are written as XML code embedded in a Lua comment at the top of each individual mission or event. In the case a Lua file does not contain a header, it is ignored and not loaded as a mission or event.

The header has to be at the top of the file starting with `--[[` and ending with `--]]` which are long Lua comments with newlines. A full example is shown below using all the parameters, however, some are contradictory in this case.

```
--[[
<?xml version='1.0' encoding='utf8'?>
<mission name="Mission Name">
 <unique />
 <chance>5</chance>
 <location>Bar</location>
 <chapter>[^0]</chapter>
 <spob>Caladan</spob>
 <faction>Empire</faction>
 <system>Delta Pavonis</system>
 <cond>player.credits() &gt; 10e3</cond>
 <done>Another Mission</done>
 <priority>4</priority>
 <tags>
  <some_random_binary_tag />
 </tags>
 <notes />
</mission>
--]]
```

Let us go over the different parameters. First of all, either a `<mission>` or `<event>` node is necessary as the root for either missions (located in `dat/missions/`) or events (located in `dat/events/`). The `name` attribute has to be set to a unique string and will be used to identify the mission.

Next it is possible to identify mission properties. In particular, only the `<unique />` property is supported, which indicates the mission can only be completed once. It will not appear again to the same player.

The header includes all the information about mission availability. Most are optional and ignored if not provided. The following nodes can be used to control the availability:

- **chance**: *required field*. Indicates the chance that the mission appears. For values over 100, the whole part of dividing the value by 100 indicates how many instances can spawn, and the remainder is the chance of each instance. So, for example, a value of 320 indicates that 3 instances can spawn with 20% each.
- **location**: *required field*. Indicates where the mission or event can start. It can be one of `none`, `land`, `enter`, `load`, `computer`, or `bar`. Note that not all are supported by both missions and events. More details will be discussed later in this section.
- **unique**: the presence of this tag indicates the mission or event is unique and will *not appear again* once fully completed.

- **chapter**: indicates what chapter it can appear in. Note that this is regular expression-powered. Something like `0` will match chapter 0 only, while you can write `[01]` to match either chapter 0 or 1. All chapters except 0 would be `[^0]`, and such. Please refer to a regular expression guide such as regexr[6] for more information on how to write regex.
- **faction**: must match a faction. Multiple can be specified, and only one has to match. In the case of `land`, `computer`, or `bar` locations it refers to the spob faction, while for `enter` locations it refers to the system faction.
- **spob**: must match a specific spob. Only used for `land`, `computer`, and `bar` locations. Only one can be specified.
- **system**: must match a specific system. Only used for `enter` location and only one can be specified.
- **cond**: arbitrary Lua conditional code. The Lua code must return a boolean value. For example `player.credits() > 10e3` would mean the player having more than 10,000 credits. Note that since this is XML, you have to escape < and > with `<` and `>`, respectively. Multiple expressions can be hooked with `and` and `or` like regular Lua code. If the code does not contain any `return` statements, `return` is prepended to the string.
- **done**: indicates that the mission must be done. This allows to create mission strings where one starts after the next one.
- **priority**: indicates what priority the mission has. Lower priority makes the mission more important. Missions are processed in priority order, so lower priority increases the chance of missions being able to perform claims. If not specified, it is set to the default value of 5.

The valid location parameters are as follows:

Location	Event	Mission	Description
none	✓	✓	Not available anywhere.
land	✓	✓	Run when player lands
enter	✓	✓	Run when the player enters a system.
load	✓		Run when the game is loaded.
computer		✓	Available at mission computers.
bar		✓	Available at spaceport bars.

Note that availability differs between events and missions. Furthermore, there are two special cases for missions: `computer` and `bar` that both support an `accept` function. In the case of the mission computer, the `accept` function is run when the player tries to click on the accept button in the interface. On

[6] https://regexr.com/

the other hand, the spaceport bar `accept` function is called when the NPC is approached. This NPC must be defined with `misn.setNPC` to be approachable.

Also notice that it is also possible to define arbitrary tags in the `<tags>` node. This can be accessed with `player.misnDoneList()` and can be used for things such as handling faction standing caps automatically.

Finally, there is a `<notes>` section that contains optional metadata about the metadata. This is only used by auxiliary tools to create visualizations of mission maps.

Example: Cargo Missions

Cargo missions appear at the mission computer in a multitude of different factions. Since they are not too important, they have a lower than default priority (6). Furthermore, they have 9 independent chances to appear, each with 60% chance. This is written as `<chance>960</chance>`. The full example is shown below:

```
--[[
<?xml version='1.0' encoding='utf8'?>
<mission name="Cargo">
 <priority>6</priority>
 <chance>960</chance>
 <location>Computer</location>
 <faction>Dvaered</faction>
 <faction>Empire</faction>
 <faction>Frontier</faction>
 <faction>Goddard</faction>
 <faction>Independent</faction>
 <faction>Sirius</faction>
 <faction>Soromid</faction>
 <faction>Za'lek</faction>
 <notes>
  <tier>1</tier>
 </notes>
</mission>
--]]
```

Example: Antlejos

Terraforming antlejos missions form a chain. Each mission requires the previous one and are available at the same planet (Antlejos V) with 100% chance. The priority is slightly lower than default to try to ensure the claims get through. Most missions trigger on *Land* (`<location>Land</location>`) because Antlejos V does not have a spaceport bar at the beginning. The full example is shown below:

```
--[[
<?xml version='1.0' encoding='utf8'?>
<mission name="Terraforming Antlejos 3">
 <unique />
 <priority>4</priority>
 <chance>100</chance>
 <location>Land</location>
 <spob>Antlejos V</spob>
 <done>Terraforming Antlejos 2</done>
 <notes>
  <campaign>Terraforming Antlejos</campaign>
 </notes>
</mission>
--]]
```

Example: Taiomi

Next is an example of a unique event. The Finding Taiomi event has a 100% of appearing in the `Bastion` system outside of Chapter 0. It triggers automatically when entering the system (`<location>enter</location>`).

```
--[[
<?xml version='1.0' encoding='utf8'?>
<event name="Finding Taiomi">
 <location>enter</location>
 <unique />
 <chance>100</chance>
 <cond>system.cur() == system.get("Bastion")</cond>
 <chapter>[^0]</chapter>
 <notes>
  <campaign>Taiomi</campaign>
 </notes>
</event>
--]]
```

4.3.2 Mission Computer MissionsA

TODO

4.3.3 Memory Model

By default, variables in Lua scripts are not saved when the player saves the game. This means that all the values you have set up will be cleared if the player saves and loads. This can lead to problems with scripts that do the following:

```
local dest

function create ()
   dest = spob.get("Caladan")

   -- ...

   hook.land( "land" )
end

function land ()
   if spob.cur() == dest then -- This is wrong!
      -- ...
   end
end
```

In the above script, a variable called `dest` is created, and when the mission is created, it gets set to `spob.get("Caladan")`. Afterwards, it gets used in `land` which is triggered by a hook when the player lands. For this mission, the value `dest` will be set as long as the player doesn't save and load. When the player saves and loads, the value `dest` gets set to `nil` by default in the first line. However, upon loading, the `create` function doesn't get run again, while the hook is still active. This means that when the player lands, `spob.cur()` will be compared with `dest` will not have been set, and thus always be false. In conclusion, the player will never be able to finish the mission!

How do we fix this? The solution is the mission/event memory model. In particular, all mission / event instances have a table that gets set called `mem`. This table has the particular property of being *persistent*, i.e., even if the player saves and loads the game, the contents will not change! We can then use this table and insert values to avoid issues with saving and loading games. Let us update the previous code to work as expected with saving and loading.

```
function create ()
   mem.dest = spob.get("Caladan")

   -- ...

   hook.land( "land" )
end

function land ()
   if spob.cur() == mem.dest then
      -- ...
   end
end
```

We can see the changes are minimal. We no longer declare the `dest`

variable, and instead of setting and accessing `dest`, we use `mem.dest`, which is the `dest` field of the `mem` persistent memory table. With these changes, the mission is now robust to saving and loading!

It is important to note that almost everything can be stored in the `mem` table, and this includes other tables. However, make sure to not create loops or it will hang the saving of the games.

The most common use of the persistent memory table `mem` is variables that keep track of the mission progress, such as if the player has delivered cargo or has talked to a certain NPC.

4.3.4 Mission Variables

Mission variables allow storing arbitrary variables in save files. Unlike the `mem` per-mission/event memory model, these are per-player and can be read and written by any Lua code. The API is available as part of the `var` module[7].

The core of the `var` module is three functions:

- `var.peek( varname )`: allows obtaining the value of a mission variable called `varname`. If it does not exist it returns `nil`.
- `var.push( varname, value )`: creates a new mission variable `varname` or overwrites an existing mission variable `varname` if it exists with the value `value`. Note that not all data types are supported, but many are.
- `var.pop( varname)`: removes a mission variable.

It is common to use mission variables to store outcomes in mission strings that affect other missions or events. Since they can also be read by any Lua code, they are useful in `<cond>` header statements too.

Supported variable types are `number`, `boolean`, `string`, and `time`. If you want to pass systems and other data, you have to pass it via untranslated name `:nameRaw()` and then use the corresponding `.get()` function to convert it to the corresponding type again.

4.3.5 Hooks

Hooks are the basic way missions and events can interact with the game. They are accessed via the `hook.*` API and basically serve the purpose of binding script functions to specific in-game events or actions. A full list of the hook API is available here[8] and the API is always available in missions and events. **Hooks are saved and loaded automatically.**

The basics to using hooks is as follows:

[7] https://naev.org/api/modules/var.html
[8] https://naev.org/api/modules/hook.html

```
function create ()
   -- ...

   hook.land( "land" )
end

function land ()
   -- ...
end
```

In this example, at the end of the `create` function, the local function `land` is bound to the player landing with `hook.land`. Thus, whenever the player lands, the script function `land` will be run. All hook functions return a hook ID that can be used to remove the hook with `hook.rm`. For example, we can write a slightly more complicated example as such:

```
function create ()
   -- ...

   mem.hook_land = hook.land( "land" )
   mem.hook_enter = hook.enter( "enter" )
end

function land ()
   -- ...
end

function enter ()
   hook.rm( mem.hook_land )
   hook.rm( mem.hook_enter )
end
```

The above example is setting up a `land` hook when the player lands, and an `enter` hook, which activates whenever the player enters a system by either taking off or jumping. Both hooks are stored in persistent memory, and are removed when the `enter` function is run when the player enters a system.

Each mission or event can have an infinite number of hooks enabled. Except for `timer` and `safe` hooks, hooks do not get removed when run.

Timer Hooks

Timer hooks are hooks that get run once when a certain amount of real in-game time has passed. Once the hook is triggered, it gets removed automatically. If you wish to repeat a function periodically, you have to create a new timer hook. A commonly used example is shown below.

```
function create ()
   -- ...
```

```
   hook.enter( "enter" )
end

function enter ()
   -- ...

   hook.timer( 5, "dostuff" )
end

function dostuff ()
   if condition then
      -- ...
      return
   end
   -- ...
   hook.timer( 5, "dostuff" )
end
```

In this example, an `enter` hook is created and triggered when the player enters a system by taking off or jumping. Then, in the `enter` function, a 5-second timer hook is started that runs the `dostuff` function when the time is up. The `dostuff` function then checks a condition to do something and end, otherwise it repeats the 5-second hook. This system can be used to, for example, detect when the player is near a pilot or position, or display periodic messages.

Timer hooks persist even when the player lands and takes off. If you wish to clear them, please use `hook.timerClear()`, which will remove all the timers created by the mission or event calling the function. This can be useful in combination with `hook.enter`.

Pilot Hooks

When it comes to pilots, hooks can also be used. However, given that pilots are not saved, the hooks are not saved either. The hooks can be made to be specific to a particular pilot, or apply to any pilot. In either case, the pilot triggering the hook is passed as a parameter. An illustrative example is shown below:

```
function enter ()
   -- ...

   local p = pilot.add( "Llama", "Independent" )
   hook.pilot( p, "death", "pilot_died" )
end

function pilot_died( p )
   -- ...
```

```
end
```

In the above example, when the player enters a system with the `enter` function, a new pilot `p` is created, and a `"death"` hook is set on that pilot. Thus, when the pilot `p` dies, the `pilot_dead` function will get called. Furthermore, the `pilot_died` function takes the pilot that died as a parameter.

There are other hooks for a diversity of pilot actions that are documented in the official API documentation[9], allowing for full control of pilot actions.

4.3.6 Translation Support

Naev supports translation through Weblate[10]. However, in order for translations to be used you have to mark strings as translatable. This is done with a gettext[11] compatible interface. In particular, the following functions are provided:

- `_()`: This function takes a string, marks it as translatable, and returns the translated version.
- `N_()`: This function takes a string, marks it as translatable, however, it returns the *untranslated* version of the string.
- `n_()`: Takes two strings related to a number quantity and return the translated version that matches the number quantity. This is because some languages translate number quantities differently. For example "1 apple", but "2 apple**s**".
- `p_()`: This function takes two strings, the first is a context string, and the second is the string to translate. It returns the translated string. This allows to disambiguate same strings based on context such as `p_( "main menu", "Close" )` and `p_( "some guy", "Close" )`. In this case `"Close"` can be translated differently based on the context strings.

In general, you want to use `_()` and `n_()` to envelop all strings that are being shown to the player, which will allow for translations to work without extra effort. For example, when defining a new mission you want to translate all the strings as shown below:

```
misn.setTitle( _("My Mission") )
misn.setDesc( _("You have been asked to do lots of fancy stuff for a
    very fancy individual. How fancy!") )
misn.setReward( _("Lots of good stuff!") )
```

Note that `_()` and friends all assume that you are inputting strings in English.

[9] https://naev.org/api/modules/hook.html#pilot
[10] https://hosted.weblate.org/projects/naev/naev/
[11] https://www.gnu.org/software/gettext/

It is important to note that strings not shown to the player, e.g., strings representing faction names or ship names, do not need to be translated! So when adding a pilot you can just use directly the correct strings for the ship and faction (e.g., `"Hyena"` and `"Mercenary"`):

```
pilot.add( "Hyena", "Mercenary", nil, _("Cool Dude") )
```

Note that the name (`Cool Dude` in this case) does have to be translated!

For plurals, you have to use `n_()` given that not all languages pluralize like in English. For example, if you want to indicate how many pirates are left, you could do something like:

```
player.msg(string.format(n_( "%d pirate left!", "%d pirates left!", left
    ), left ))
```

The above example says how many pirates are left based on the value of the variable `left`. In the case there is a single pirate left, the singular form should be used in English, which is the first parameter. For other cases, the plural form is used. The value of the variable `left` determines which is used based on translated language. Although the example above uses `string.format` to display the number value for illustrative purposes, it is recommended to format text with the `format` library explained below.

4.3.7 Formatting Text

An important part of displaying information to the player is formatting text. While `string.format` exists, it is not very good for translations, as the Lua version can not change the order of parameters unlike C. For this purpose, we have prepared the `format` library, which is much more intuitive and powerful than string.format. A small example is shown below:

```
local fmt = require "format"

function create ()
   -- ...
   local spb, sys = spob.getS( "Caladan" )
   local desc = fmt.f( _("Take this cheese to {spb} ({sys}), {name}."),
        { spb=spb, sys=sys, name=player.name() } )
   misn.setDesc( desc )
end
```

Let us break down this example. First, we include the library as `fmt`. This is the recommended way of including it. Afterwards, we run `fmt.f` which is the main formatting function. This takes two parameters: a string to be formatted, and a table of values to format with. The string contains substrings of the form `"{foo}"`, that is, a variable name surrounded by `{` and `}`. Each of these substrings is replaced by the corresponding field in the table passed as

the second parameter, which are converted to strings. So, in this case, `{spb}` gets replaced by the value of `table.spb` which in this case is the variable `spb` that corresponds to the Spob of `Caladan`. This gets converted to a string, which in this case is the translated name of the planet. If any of the substrings are missing and not found in the table, it will raise an error.

There are additional useful functions in the `format` library. In particular the following:

- `format.number`: Converts a non-negative integer into a human readable number as a string. Gets rounded to the nearest integer.
- `format.credits`: Displays a credit value with the credit symbol ¤.
- `format.reward`: Used for displaying mission rewards.
- `format.tonnes`: Used to convert tonne values to strings.
- `format.list`: Displays a list of values with commas and the word "and". For example `fmt.list{"one", "two", "three"}` returns `"one, two, and three"`.
- `format.humanize`: Converts a number string to a human readable rough string such as `"1.5 billion"`.

More details can be found in the generated documentation[12].

4.3.8 Colouring Text

All string printing functions in Naev accept special combinations to change the colour. This will work whenever the string is shown to the player. In particular, the character `#` is used for a prefix to set the colour of text in a string. The colour is determined by the character after `#`. In particular, the following are valid values:

[12] https://naev.org/api/modules/format.html

Symbol	Description
`#0`	Resets colour to the default value.
`#r`	Red colour.
`#g`	Green colour.
`#b`	Blue colour.
`#o`	Orange colour.
`#y`	Yellow colour.
`#w`	White colour.
`#p`	Purple colour.
`#n`	Grey colour.
`#F`	Colour indicating friend.
`#H`	Colour indicating hostile.
`#N`	Colour indicating neutral.
`#I`	Colour indicating inert.
`#R`	Colour indicating restricted.

Multiple colours can be used in a string such as `"It is a #ggood#0 #rmonday#0!"`. In this case, the word `"good"` is shown in green, and `"monday"` is shown in red. The rest of the text will be shown in the default colour.

While it is possible to accent and emphasize text with this, it is important to not go too overboard, as it can difficult translating. When possible, it is also best to put the colour outside of the string being translated. For example `_("#rred#0")` should be written as `"#r".._("red").."#0"`.

4.3.9 System Claiming

One important aspect of mission and event development are system claiming. Claims serve the purpose of avoiding collisions between Lua code. For example, `pilot.clear()` allows removing all pilots from a system. However, say that there are two events going on in a system. They both run `pilot.clear()` and add some custom pilots. What will happen then, is that the second event to run will get rid of all the pilots created from the first event, likely resulting in Lua errors. This is not what we want is it? In this case, we would want both events to try to claim the system and abort if the system was already claimed.

Systems can be claimed with either `misn.claim` or `evt.claim` depending on whether they are being claimed by a mission or an event. A mission or event can claim multiple systems at once, and claims can be exclusive (default) or inclusive. Exclusive claims don't allow any other system to claim the system, while inclusive claims can claim the same system. In general, if you use things like `pilot.clear()` you should use exclusive claims, while if

you don't mind if other missions / events share the system, you should use inclusive claims. **You have to claim all systems that your mission uses to avoid collisions!**

Let us look at the standard way to use claims in a mission or event:

```
function create ()
   if not misn.claim( {system.get("Gamma Polaris")} ) then
      misn.finish(false)
   end

   -- ...
end
```

The above mission tries to claim the system `"Gamma Polaris"` right away in the `create` function. If it fails and the function returns false, the mission then finishes unsuccessfully with `misn.finish(false)`. This will cause the mission to only start when it can claim the `"Gamma Polaris"` system and silently fail otherwise. You can pass more systems to claim them, and by default they will be *exclusive* claims.

Say our event only adds a small derelict in the system and we don't mind it sharing the system with other missions and events. Then we can write the event as:

```
function create ()
   if not evt.claim( {system.get("Gamma Polaris")}, true ) then
      evt.finish(false)
   end

   -- ...
end
```

In this case, the second parameter is set to `true` which indicates that this event is trying to do an **inclusive** claim. Again, if the claiming fails, the event silently fails.

Claims can also be tested in an event/mission-neutral way with `naev.claimTest`. However, this can only test the claims. Only `misn.claim` and `evt.claim` can enforce claims for missions and events, respectively.

As missions and events are processed by `priority`, make sure to give higher priority to those that you want to be able to claim easier. Otherwise, they will have difficulties claiming systems and may never appear to the player. Minimizing the number of claims and cutting up missions and events into smaller parts is also a way to minimize the amount of claim collisions.

4.3.10 Mission Cargo

Cargo given to the player by missions using `misn.cargoAdd` is known as **Mission Cargo**. This differs from normal cargo in that only the player's ship

can carry it (escorts are not allowed to), and that if the player jettisons it, the mission gets aborted. Missions and events can still add normal cargo through `pilot.cargoAdd` or `player.fleetCargoAdd`, however, only missions can have mission cargo. It is important to note that *when the mission finishes, all associated mission cargos of the mission are also removed!*

The API for mission cargo is fairly simple and relies on three functions:

- `misn.cargoAdd`: takes a commodity or string with a commodity name, and the amount to add. It returns the id of the mission cargo. This ID can be used with the other mission cargo functions.
- `misn.cargoRm`: takes a mission cargo ID as a parameter and removes it. Returns true on success, false otherwise.
- `misn.cargojet`: same as `misn.cargoRm`, but it jets the cargo into space (small visual effect).

Custom Commodities

Commodities are generally defined in `dat/commodities/`, however, it is a common need for a mission to have custom cargo. Instead of bloating the commodity definitions, it is possible to create arbitrary commodities dynamically. Once created, they are saved with the player, but will disappear when the player gets rid of them. There are two functions to handle custom commodities:

- `commodity.new`: takes the name of the cargo, description, and an optional set of parameters and returns a new commodity. If it already exist, it returns the commodity with the same name. It is important to note that you have to pass *untranslated* strings. However, in order to allow for translation, they should be used with `N_()`.
- `commodity.illegalto`: makes a custom commodity illegal to a faction, and takes the commodity and a faction or table of factions to make the commodity illegal to as parameters. Note that this function only works with custom commodities.

An full example of adding a custom commodity to the player is as follows:

```
local c = commodity.new( N_("Smelly Cheese"), N_("This cheese smells
    really bad. It must be great!") )
c:illegalto( {"Empire", "Sirius"} )
mem.cargo_id = misn.cargoAdd( c, 1 )
-- Later it is possible to remove the cargo with misn.cargoRm(
    mem.cargo_id )
```

4.3.11 Ship Log

The Ship Log is a framework that allows recording in-game events so that the player can easily access them later on. This is meant to help players that haven't logged in for a while or have forgotten what they have done in their game. The core API is in the `shiplog` module[13] and is a core library that is always loaded without the need to `require`. It consists of two functions:

- `shiplog.create`: takes three parameters, the first specifies the id of the log (string), the second the name of the log (string, visible to player), and the third is the logtype (string, visible to player and used to group logs).
- `shiplog.append`: takes two parameters, the first specifies the id of the log (string), and second is the message to append. The ID should match one created by `shiplog.create`.

The logs have the following hierarchy: logtype → log name → message. The logtype and log name are specified by `shiplog.create` and the messages are added with `shiplog.append`. Since, by default, `shiplog.create` doesn't overwrite existing logs, it's very common to write a helper log function as follows:

```
local function addlog( msg )
   local logid = "my_log_id"
   shiplog.create( logid, _("Secret Stuff"), _("Neutral") )
   shiplog.append( logid, msg )
end
```

You would use the function to quickly add log messages with `addlog(_("This is a message relating to secret stuff."))`. Usually logs are added when important one-time things happen during missions or when they are completed.

4.3.12 Visual Novel Framework 

The Visual Novel framework is based on the Love2D API and allows for displaying text, characters, and other effects to the player. It can be thought of as a graph representing the choices and messages the player can engage with. The core API is in the `vn` module[14].

The VN API is similar to existing frameworks such as Ren'Py[15], in which conversations are divided into scenes with characters. In particular, the flow of engaging the player with the VN framework consists roughly of the following:

[13] https://naev.org/api/modules/shiplog.html
[14] https://naev.org/api/modules/vn.html
[15] https://renpy.org

1. Clear internal variables (recommended)
2. Start a new scene
3. Define all the characters that should appear in the scene (they can still be added and removed in the scene with `vn.appear` and `vn.disappear`)
4. Run the transition to make the characters and scene appear
5. Display text
6. Jump to 2. as needed or end the `vn`

For most purposes, all you will need is a single scene, however, you are not limited to that. The VN is based around adding nodes which represent things like displaying text or giving the player options. Once the conversation graph defined by the nodes is set up, `vn.run()` will begin execution and *it won't return until the dialogue is done*. Nodes are run in consecutive order unless `vn.jump` is used to jump to a label node defined with `vn.label`. Let us start by looking at a simple example:

```
local vn = require "vn" -- Load the library

-- Below would be what you would include when you want the dialogue
vn.clear() -- Clear internal variables
vn.scene() -- Start a new scene
local mychar = vn.newCharacter( _("Alex"), {image="mychar.webp"} )
vn.transition() -- Will fade in the new character
vn.na(_([[You see a character appear in front of you.]]) -- Narrator
mychar(_([[How do you do?]])
vn.menu{ -- Give a list of options the player chooses from
   {_("Good."), "good"},
   {_("Bad."), "bad"},
}

vn.label("good") -- Triggered when the "good" option is chosen
mychar(_("Great!"))
vn.done() -- Finish

vn.label("bad") -- Triggered on "bad" option
mychar(_("That's not …good"))
vn.run()
```

Above is a simple example that creates a new scene with a single character (`mychar`), introduces the character with the narrator (`vn.na`), has the character talk, and then gives two choices to the player that trigger different messages. By default the `vn.transition()` will do a fading transition, but you can change the parameters to do different ones. The narrator API is always available with `vn.na`, and once you create a character with `vn.newCharacter`, you can simple call the variable to have the character talk. The character images are looking for in the `gfx/vn/characters/` directory, and in this case it would try to use the file `gfx/vn/characters/mychar.webp`.

Player choices are controlled with `vn.menu` which receives a table where

each entry consists of another table with the first entry being the string to display (e.g., `_("Good.")`), and the second entry being either a function to run, or a string representing a label to jump to (e.g., `"good"`). In the case of passing strings, `vn.jump` is used to jump to the label, so that in the example above the first option jumps to `vn.label("good")`, while the second one jumps to `vn.label("bad")`. By using `vn.jump`, `vn.label`, and `vn.menu` it is possible to create complex interactions and loops.

It is recommended to look at existing missions for examples of what can be done with the `vn` framework.

`vntk` Wrapper

The full `vn` framework can be a bit verbose when only displaying small messages or giving small options. For this purpose, the `vntk` module[16] can simplify the usage, as it is a wrapper around the `vn` framework. Like the `vn` framework, you have to import the library with `require`, and all the functions are blocking, that is, the Lua code execution will not continue until the dialogues have closed. Let us look at some simple examples of `vntk.msg` and `vntk.yesno` below:

```lua
local vntk = require "vntk"

-- …
vntk.msg( _("Caption"), _("Some message to show to the player.") )

-- …
if vntk.yesno( _("Cheese?"), _("Do you like cheese?") ) then
   -- player likes cheese
else
   -- player does not
end
```

The code is very simple and requires the library. Then it will display a message, and afterwards, it will display another with a `Yes` and `No` prompt. If the player chooses yes, the first part of the code will be executed, and if they choose no, the second part is executed.

Arbitrary Code Execution

It is also possible to create nodes in the dialogue that execute arbitrary Lua code, and can be used to do things such as pay the player money or modify mission variables. Note that you can not write Lua code directly, or it will be executed when the `vn` is being set up. To have the code run when triggered

[16] https://naev.org/api/modules/vntk.html

by the `vn` framework, you must use `vn.func` and pass a function to it. A very simple example would be

```
-- ...
vn.label("pay_player")
vn.na(_("You got some credits!"))
vn.func( function ()
   player.pay( 50e3 )
end )
-- ...
```

It is also to execute conditional jumps in the function code with `vn.jump`. This would allow to condition the dialogue on things like the player's free space or amount of credits as shown below:

```
-- ...
vn.func( function ()
   if player.pilot():cargoFree() < 10 then
      vn.jump("no_space")
   else
      vn.jump("has_space")
   end
end )

vn.label("no_space")
-- ...

vn.label("has_space")
-- ...
```

In the code above, a different dialogue will be run depending on whether the player has less than 10 free cargo space or more than that.

As you can guess, `vn.func` is really powerful and opens up all sorts of behaviour. You can also set local or global variables with it, which is very useful to detect if a player has accepted or not a mission.

4.4 Advanced Usage

TODO

4.4.1 Handling Aborting Missions

When missions are aborted, the `abort` function is run if it exists. Although this function can't stop the mission from aborting, it can be used to clean up the mission stuff, or even start events such as a penalty for quitting halfway through the mission. A representative example is below:

```
local vntk = require "vntk"

...

function abort ()
   vntk.msg(_("Mission Failure!"),_([[You have failed the mission, try
       again next time!]]))
end
```

Not that it is not necessary to run `misn.finish()` nor any other clean up functions; this is all done for you by the engine.

4.4.2 Dynamic Factions

TODO

4.4.3 Minigames

TODO

4.4.4 Cutscenes

Cutscenes are a powerful way of conveying events that the player may or may not interact with. In order to activate cinematic mode, you must use `player.cinematics` function. However, the player will still be controllable and escorts will be doing their thing. If you want to make the player and escorts stop and be invulnerable, you can use the `cinema` library. In particular, the `cinema.on` function enables cinema mode and `cinema.off` disables it.

You can also control where the camera is with `camera.set()`. By default, it will try to center the camera on the player, but if you pass a position or pilot as a parameter, it will move to the position or follow the pilot, respectively.

The cornerstone of cutscenes is to use hooks to make things happen and show that to the player. In this case, one of the most useful hooks is the `hook.timer` timer hook. Let us put it all together to do a short example.

```
local cinema = require "cinema" -- load the cinema library
...
local someguy -- assume some pilot is stored here

-- function that starts the cutscene
function cutscene00 ()
    cinema.on()
    camera.set( someguy ) -- make the camera go to someguy
    hook.timer( 5, "cutscene01" ) -- advance to next step in 5 seconds
end
```

```
function cutscene01 ()
    someguy:broadcast(_("I like cheese!"),true) -- broadcast so the
        player can always see it
    hook.timer( 6, "cutscene02" ) -- give 6 seconds for the player to see
end
function cutscene02 ()
    cinema.off()
    camera.set()
end
```

Breaking down the example above, the cutscene itself is made of 3 functions. The first `cutscene00` initializes the cinematic mode and sets the camera to someguy. Afterwards, `cutscene01` makes someguy say some text and shows it to the player. Finally, in `cutscene02`, the cinematic mode is finished and the camera is returned to the player.

While that is the basics, there is no limit to what can be done. It is possible to use shaders to create more visual effects, or the luaspfx library. Furthermore, pilots can be controlled and made to do all sorts of actions. There is no limit to what is possible!

4.4.5 Unidiff

TODO

4.4.6 Equipping with `equipopt`

TODO

4.4.7 Event-Mission Communication

In general, events and missions are to be seen as self-contained isolated entities, that is, they do not affect each other outside of mission variables. However, it is possible to exploit the `hook` module API to overcome this limitation with `hook.custom` and `naev.trigger`:

- `hook.custom`: allows defining an arbitrary hook on an arbitrary string. The function takes two parameters: the first is the string to hook (should not collide with standard names), and the second is the function to run when the hook is triggered.
- `naev.trigger`: also takes two parameters and allows triggering the hooks set by `hook.custom`. In particular, the first parameter is the same as the first parameter string passed to `hook.custom`, and the second optional parameter is data to pass to the custom hooks.

For example, you can define a mission to listen to a hook as below:

```
function create ()
   -- ...

   hook.custom( "my_custom_hook_type", "dohook" )
end

function dohook( param )
   print( param )
end
```

In this case, `"my_custom_hook_type"` is the name we are using for the hook. It is chosen to not conflict with any of the existing names. When the hook triggers, it runs the function `dohook` which just prints the parameter. Now, we can trigger this hook from anywhere simply by using the following code:

```
   naev.trigger( "my_custom_hook_type", some_parameter )
```

The hook will not be triggered immediately, but the second the current running code is done to ensure that no Lua code is run in parallel. In general, the mission variables should be more than good enough for event-mission communication, however, in the few cases communication needs to be more tightly coupled, custom hooks are a perfect solution.

4.4.8 LuaTK API

TODO

4.4.9 Love2D API

> LÖVE is an *awesome* framework you can use to make 2D games in Lua. It's free, open-source, and works on Windows, Mac OS X, Linux, Android and iOS.

Naev implements a subset of the LÖVE[17] API (also known as Love2D), allowing it to execute many Love2D games out of the box. Furthermore, it is possible to use the Naev API from inside the Love2D to have the games interact with the Naev engine. In particular, the VN (Sec. 4.3.12), minigames (Sec. 4.4.3), and LuaTK (Sec. 4.4.8) are implemented using the Love2D API. Many of the core game functionality, such as the boarding or communication menus make use of this API also, albeit indirectly.

The Love2D API works with a custom dialogue window that has to be started up. There are two ways to do this: create a Love2D game directory and

[17] https://love2d.org/

run them, or set up the necessary functions and create the Love2D instance. Both are very similar.

The easiest way is to create a directory with a `main.lua` file that will be run like a normal Love2D game. At the current time the Naev Love2D API does not support zip files. An optional `conf.lua` file can control settings of the game. Afterwards you can start the game with:

```
local love = require "love"
love.exec( "path/to/directory" )
```

If the directory is a correct Love2D game, it will create a window instead of Naev and be run in that. You can use `love.graphics.setBackgroundColor( 0, 0, 0, 0 )` to make the background transparent, and the following `conf.lua` function will make the virtual Love2D window use the entire screen, allowing you to draw normally on the screen.

```
function love.conf(t)
   t.window.fullscreen = true
end
```

The more advanced way to set up Love2D is to directly populate the `love` namespace with the necessary functions, such as `love.load`, `love.conf`. `love.draw`, etc. Afterwards you can use `love.run()` to start the Love2D game and create the virtual window. This way is much more compact and does not require creating a separate directory structure with a `main.lua`.

Please note that while most of the core Love2D 11.4 API is implemented, more niche API and things that depend on external libraries like `love.physics`, `lua-enet`, or `luasocket` are not implemented. If you wish to have missing API added, it is possible to open an issue for the missing API or create a pull request. Also note that there are

Differences with Love2D API

Some of the known differences with the Love2D API are as follows:

- You can call images or canvases to render them with `object:draw( ... )` instead of only `love.graphics.draw( obj, ... )`.
- Fonts default to Naev fonts.
- You can use Naev colour strings such as `"#b"` in `love.graphics.print` and `love.graphics.printf`.
- `audio.newSource` defaults to second paramater `"static"` unless specified (older Love2D versions defaulted to `"stream"`, and it must be set explicity in newer versions).
- `love.graphics.setBackgroundColor` uses alpha colour to set the alpha of the window, with 0 making the Love2D window not drawn.

4.5 Tips and Tricks

This section contains some tricks and tips for better understanding how to do specific things in missions and events.

4.5.1 Optimizing Loading

It is important to understand how missions and events are loaded. The headers are parsed at the beginning of the game and stored in memory. Whenever a trigger (entering a system, landing on a spob, etc.) happens, the game runs through all the missions and events to check to see if they should be started. The execution is done in the following way:

1. Header statements are checked (*e.g.*, unique missions that are already done are discarded)
2. Lua conditional code is compiled and run
3. Lua code is compiled and run
4. `create` function is run

In general, since many events and missions can be checked at every frame, it is important to try to cull them as soon as possible. When you can, use location or faction filters to avoid having missions and events appear in triggers you don't wish them to. In the case that is not possible, try to use the Lua conditional code contained in the `<cond>` node in the header. You can either write simple conditional statements such as `player.credits() > 50e3`, where `return` gets automatically prepended, or you can write more complex code where you have to manually call `return` and return a boolean value. Do note, however, that it is not possible to reuse variables or code in the `<cond>` node in the rest of the program. If you have to do expensive computations and wish to use the variables later on, it is best to put the conditional code in the `create` function and abort the mission or event with `misn.finish(false)` or `evt.finish(false)`, respectively.

Furthermore, when a mission or event passes the header and conditional Lua statements, the entire code gets compiled and run. This implies that all global variables are computed. If you load many graphics, shaders, or sound files as global values, this can cause a slowdown whenever the mission is started. An early issue with the visual novel framework was that all cargo missions were loading the visual novel framework that were loading lots of sounds and shaders. Since this was repeated for every mission in the mission computer, it created noticeable slowdowns. This was solved by relying on lazy loading and caching, and not just blindly loading graphics and audio files into global variables on library load.

4.5.2 Global Cache

In some cases that you want to load large amount of data once and reuse it throughout different instances of events or missions, it is possible to use the global cache with `naev.cache()`. This function returns a table that is accessible by all the Lua code. However, this cache is cleared every time the game starts. You can not rely on elements in this cache to be persistent. It is common to wrap around the cache with the following code:

```
local function get_calculation ()
   local nc = naev.cache()
   if nc.my_calculation then
      return nc.my_calculation
   end
   nc.my_calculation = do_expensive_calculation ()
   return nc.my_calculation
end
```

The above code tries to access data in the cache. However, if it does not exist (by default all fields in Lua are nil), it will do the expensive calculation and store it in the cache. Thus, the first call of `get_calculation()` will be slow, however, all subsequent calls will be very fast as no `do_expensive_calculation()` gets called.

4.5.3 Finding Natural Pilots

In some cases, you want a mission or event to do things with naturally spawning pilots, and not spawn new ones. Naturally spawned pilots have the `natural` flag set in their memory. You can access this with `p:memory().natural` and use this to limit boarding hooks and the likes to only naturally spawned pilots. An example would be:

```
function create ()
   -- ...
   hook.board( "my_board" )
end

function my_board( pilot_boarded )
   if not pilot_boarded:memory().natural then
      return
   end
   -- Do something with natural pilots here
end
```

In the above example, we can use a board hook to control when the player boards a ship, and only handle the case that naturally spawning pilots are boarded.

4.5.4 Making Aggressive Enemies

TODO Explain how to nudge the enemies without relying on pilot:control().

4.5.5 Working with Player Fleets

TODO Explain how to detect and/or limit player fleets.

4.6 Full Example

Below is a full example of a mission.

```
--[[
<?xml version='1.0' encoding='utf8'?>
<mission name="Mission Template (mission name goes here)">
 <unique />
 <priority>4</priority>
 <chance>5</chance>
 <location>Bar</location>
</mission>
--]]
--[[

   Mission Template (mission name goes here)

   This is a Naev mission template.
   This document aims to provide a structure on which to build many
   Naev missions and teach how to make basic missions in Naev.
   For more information on Naev, please visit: http://naev.org/
   Naev missions are written in the Lua programming language:
       http://www.lua.org/
   There is documentation on Naev's Lua API at: http://api.naev.org/
   You can study the source code of missions in
       [path_to_Naev_folder]/dat/missions/

   When creating a mission with this template, please erase the
       explanatory
   comments (such as this one) along the way, but retain the the MISSION
       and
   DESCRIPTION fields below, adapted to your mission.

   MISSION: <NAME GOES HERE>
   DESCRIPTION: <DESCRIPTION GOES HERE>

--]]

-- require statements go here. Most missions should include
```

```
-- "format", which provides the useful `number()` and
-- `credits()` functions. We use these functions to format numbers
-- as text properly in Naev. dat/scripts/common/neutral.lua provides
-- the addMiscLog function, which is typically used for non-factional
-- unique missions.
local fmt = require "format"
local neu = require "common.neutral"
local vntk = require "vntk"

--[[
Multi-paragraph dialog strings *can* go here, each with an identifiable
name. You can see here that we wrap strings that are displayed to the
player with `_()`. This is a call to gettext, which enables
localization. The _() call should be used directly on the string, as
shown here, instead of on a variable, so that the script which figures
out what all the translatable text is can find it.

When writing dialog, write it like a book (in the present-tense), with
paragraphs and quotations and all that good stuff. Leave the first
paragraph unindented, and indent every subsequent paragraph by four (4)
spaces. Use quotation marks as would be standard in a book. However, do
*not* quote the player speaking; instead, paraphrase what the player
generally says, as shown below.

In most cases, you should use double-brackets for your multi-paragraph
dialog strings, as shown below.

One thing to keep in mind: the player can be any gender, so keep all
references to the player gender-neutral. If you need to use a
third-person pronoun for the player, singular "they" is the best choice.

You may notice curly-bracketed {words} sprinkled throughout the text.
    These
are portions that will be filled in later by the mission via the
`fmt.f()` function.
--]]

-- Set some mission parameters.
-- For credit values in the thousands or millions, we use scientific
    notation (less error-prone than counting zeros).
-- There are two ways to set values usable from outside the create()
    function:
-- - Define them at file scope in a statement like "local credits =
    250e3" (good for constants)
-- - Define them as fields of a special "mem" table: "mem.credits =
    250e3" (will persist across games in the player's save file)
local misplanet, missys = spob.getS("Ulios")
local credits = 250e3

-- Here we use the `fmt.credits()` function to convert our credits
```

```
-- from a number to a string. This function both applies gettext
-- correctly for variable amounts (by using the ngettext function),
-- and formats the number in a way that is appropriate for Naev (by
-- using the numstring function). You should always use this when
-- displaying a number of credits.
local reward_text = fmt.credits( credits )

--[[
First you need to *create* the mission. This is *obligatory*.

You have to set the NPC and the description. These will show up at the
bar with the character that gives the mission and the character's
description.
--]]
function create ()
   -- Naev will keep the contents of "mem" across games if the player
       saves and quits.
   -- Track mission state there. Warning: pilot variables cannot be saved.
   mem.talked = false

   -- If we needed to claim a system, we would do that here with
   -- something like the following commented out statement. However,
   -- this mission won't be doing anything fancy with the system, so we
   -- won't make a system claim for it.
   -- Only one mission or event can claim a system at a time. Using claims
   -- helps avoid mission and event collisions. Use claims for all systems
   -- you intend to significantly mess with the behaviour of.
   --if not misn.claim(missys) then misn.finish(false) end

   -- Give the name of the NPC and the portrait used. You can see all
   -- available portraits in dat/gfx/portraits.
   misn.setNPC( _("A well-dressed man"),
       "neutral/unique/youngbusinessman.webp", _("This guy is wearing a
       nice suit.") )
end


--[[
This is an *obligatory* part which is run when the player approaches the
character.

Run misn.accept() here to internally "accept" the mission. This is
required; if you don't call misn.accept(), the mission is scrapped.
This is also where mission details are set.
--]]
function accept ()
   -- Use different text if we've already talked to him before than if
   -- this is our first time.
   local text
   if mem.talked then
```

```
   -- We use `fmt.f()` here to fill in the destination and
   -- reward text. (You may also see Lua's standard library used for
       similar purposes:
   -- `s1:format(arg1, ...)` or equivalently string.format(s1, arg1,
       ...)`.)
   -- You can tell `fmt.f()` to put a planet/system/commodity object
       into the text, and
   -- (via the `tostring` built-in) know to write its name in the
       player's native language.
   text = fmt.f(_([["Ah, it's you again! Have you changed your mind?
       Like I said, I just need transport to {pnt} in the {sys}
       system, and I'll pay you {rwd} when we get there. How's that
       sound?"]]), {pnt=misplanet, sys=missys, rwd=reward_text})
else
   text = fmt.f(_([[As you approach the guy, he looks up in curiosity.
       You sit down and ask him how his day is. "Why, fine," he
       answers. "How are you?" You answer that you are fine as well
       and compliment him on his suit, which seems to make his eyes
       light up. "Why, thanks! It's my favourite suit! I had it custom
       tailored, you know.
 "Actually, that reminds me! There was a special suit on {pnt} in the
     {sys} system, the last one I need to complete my collection, but
     I don't have a ship. You do have a ship, don't you? So I'll tell
     you what, give me a ship and I'll pay you {rwd} for it! What do
     you say?"]]),
      {pnt=misplanet, sys=missys, rwd=reward_text})
   mem.talked = true
end

-- This will create the typical "Yes/No" dialogue. It returns true if
-- yes was selected.
-- For more full-fledged visual novel API please see the vn module. The
-- vntk module wraps around that and provides a more simple and easy
    to use
-- interface, although it is much more limited.
if vntk.yesno( _("My Suit Collection"), text ) then
   -- Followup text.
   vntk.msg( _("My Suit Collection"), _([["Fantastic! I knew you would
       do it! Like I said, I'll pay you as soon as we get there. No
       rush! Just bring me there when you're ready.]]) )

   -- Accept the mission
   misn.accept()

   -- Mission details:
   -- You should always set mission details right after accepting the
   -- mission.
   misn.setTitle( _("Suits Me Fine") )
   misn.setReward( reward_text )
```

```
      misn.setDesc( fmt.f(_("A well-dressed man wants you to take him to
          {pnt} in the {sys} system so he can get some sort of special
          suit."), {pnt=misplanet, sys=missys}) )

      -- Markers indicate a target planet (or system) on the map, it may
          not be
      -- needed depending on the type of mission you're writing.
      misn.markerAdd( misplanet, "low" )

      -- The OSD shows your objectives.
      local osd_desc = {}
      osd_desc[1] = fmt.f(_("Fly to {pnt} in the {sys} system"),
          {pnt=misplanet, sys=missys} )
      misn.osdCreate( _("Suits Me Fine"), osd_desc )

      -- This is where we would define any other variables we need, but
      -- we won't need any for this example.

      -- Hooks go here. We use hooks to cause something to happen in
      -- response to an event. In this case, we use a hook for when the
      -- player lands on a planet.
      hook.land( "land" )
   end
   -- If misn.accept() isn't run, the mission doesn't change and the
       player can
   -- interact with the NPC and try to start it again.
end

-- luacheck: globals land (Hook functions passed by name)
-- ^^ That is a directive to Luacheck, telling it we're about to use a
    global variable for a legitimate reason.
-- (More info here: https://github.com/naev/naev/issues/1566) Typically
    we put these at the top of the file.

-- This is our land hook function. Once `hook.land( "land" )` is called,
-- this function will be called any time the player lands.
function land ()
   -- First check to see if we're on our target planet.
   if spob.cur() == misplanet then
      -- Mission accomplished! Now we do an outro dialog and reward the
      -- player. Rewards are usually credits, as shown here, but
      -- other rewards can also be given depending on the circumstances.
      vntk.msg( fmt.f(_([[As you arrive at {pnt}, your passenger reacts
          with glee. "I must sincerely thank you, kind stranger! Now I
          can finally complete my suit collection, and it's all thanks to
          you. Here is {reward}, as we agreed. I hope you have safe
          travels!"]]), {pnt=misplanet, reward=reward_text}) )

      -- Reward the player. Rewards are usually credits, as shown here,
      -- but other rewards can also be given depending on the
```

```
      -- circumstances.
      player.pay( credits )

      -- Add a log entry. This should only be done for unique missions.
      neu.addMiscLog( fmt.f(_([[You helped transport a well-dressed man
          to {pnt} so that he could buy some kind of special suit to
          complete his collection.]]), {pnt=misplanet} ) )

      -- Finish the mission. Passing the `true` argument marks the
      -- mission as complete.
      misn.finish( true )
   end
end
```

Chapter 5

Systems and System Objects

An important aspect of Naev is the universe. The universe is formed by isolated systems, of which only one is simulated at any given time. The systems are connected to each other forming a large graph. Each system can contain an arbitrary number of objects known as *System Objects (Spobs)*, which the player can, for example land on or perform other actions.

Most System and Spob editing can be done using the in-game editor. This is disabled by default, but by either starting the game with `--devmode` or enabling `devmode = true` in the configuration file will enable this functionality. Afterwards, an `Editor` button should appear in the main menu that should open the universe editor.

5.1 Systems

In the context of Naev, ”systems” refer to star systems, the instanced locations where starship flight and combat take place. The contents of systems consist mainly of three object types: spobs (space objects) which represent planets, space stations or other bodies of interest; asteroid fields which act as commodity sources and obstacles to weapons fire; and jump points to facilitate travel to other systems. Many systems may also have persistent effects related to nebulae including visuals, sensor interference and even constant damage over time. A ”total conversion” plugin must contain at least one system to have minimum viable content.

5.1.1 Universe Editor

Naev includes an in game editor to generate and modify both systems and their contents. The editor is accessible from the game's main menu when

Dev Mode is enabled by either of two methods: 1) Use the `--devmode` launch option. 2) In your `conf.lua`, find and set `devmode = true`.

The universe editor is far easier to use than direct editing of .XML files. You can quickly place new systems and drag them around the map, link systems by generating jump lanes and automatically generating entry and exit points, and create spobs, virtual spobs and asteroid fields within systems.

5.1.2 System XML

Each system is represented by a standalone .XML file within the `/ssys/` directory of your main or plugin data directory.

- `<ssys>`: Category which encapsulates the system's data file.
- `<name>`: Name of the system. Use this string when referencing this system in other .XML files. This name will also be displayed within the game itself.
- `<general>`: Includes data defining the size and traits of the system.
  - `<radius>`: Defines the physical dimensions of the system. This value is visualized in game by the scaling of the system travel map, and in the universe editor by a circle seen when editing systems. Jump points with the `<autopos/>` tag will be placed on this circle. System content such as spobs can be placed outside this radius but may be difficult for players to locate or access.
  - `<spacedust>`: Defines the density of space dust displayed in the system.
  - `<interference>`: Influences the sensors of ships in the system. A value greater than 0 will reduce the ranges at which you can detect, identify or destealth other ships. Reduction of detection, evasion, and stealth ranges is computed by the formula $\frac{1}{1+\frac{\text{interference}}{100}}$.
  - `<nebula>`: Reduces visibility when within the system. A value greater than 0 will cause ships, spobs and asteroids to not appear until the player gets close. The rough visibility range is computed from the formula $(1200 - \text{nebula}) \cdot \text{ewdetect} + \text{nebuvisibility}$, where ewdetect and nebuvisibility are each ships detection and nebula visibility statistics.
  - `<volatility>`: Damage over time inflicted upon ships travelling in this system. Value is expressed in MJ per second, applied to shields first and armor after.
  - `<features>`: A string value defining unique characteristics of the system, such as whether it has a factional homeworld or some other anomality. This is shown in the in-game map.
  - `<pos>`: Position of the system on the universe map, expressed as

`x` and `y` coordinates relative to the universe map's origin point.

- `<spobs>`: Category which includes all spobs, including virtual spobs, which are present in this system.
  - `<spob>`: Adds the spob of that name to the system. The coordinate position of the spob is defined within that spob's .XML file.
  - `<spob_virtual>`: Adds the virtual spob of that name to the system. Virtual spobs are used primarily for faction presences within the system.
- `<jumps>`: Category which includes coordinates and tags for jump points which allow players to travel to other systems.
  - `<jump>`: Defines a jump point.
  - `<target>`: Name of the jump point's destination system. The direction of travel when entering this jump point corresponds to that of the jump line shown on the universe map.
  - `<pos>`: Position of the jump point within the system, expressed as `x` and `y` coordinates relative to the system's `x="0" y="0"` origin point.
  - `<autopos/>`: Alternative to `<pos>` which prompts the game to generate a position for the jump point. The point will always be placed at the system boundary (the circle defined by <radius>) on a line between the current system center and the destination system.
    - '<exitonly/>: Prevents the player from detecting this jump point or entering it from the current system. These points are used exclusively as the destinations to jumps coming in from other systems.
  - `<hide>`: Modifies the range at which your sensors can discover previously unknown jump points. A value of `1` is the default and indicates no change. Values greater than `1` increase the jump point's detection distance. Values less than `1` but greater than `0` reduce the jump point's detection distance. A value of `0` is a specific exception which labels the jump as part of a Trade Route - the jump point will automatically be discovered when the player enters the system, regardless of distance, and also have some small beacons next to it.
  - `<hidden/>`: Designates the jump as a hidden point which cannot be discovered with standard sensors. In the base Naev scenario, hidden jump points are revealed to the player mainly via mission rewards, by completing certain missions or by equipping and activating a Hidden Jump Scanner outfit.
- `<asteroids>`: Category which includes coordinates and contents of asteroid fields.

- <asteroid>: Defines an asteroid field.
- <group>: Names an .XML list from /asteroids/groups that defines what asteroids spawn in this field.
- <pos>: Center position of the asteroid field within the system, expressed as x and y coordinates relative to the system's center point.
- <radius>: Size of the circular asteroid field, expressed in distance units from the field's center point as defined in the <pos> field.
- <density>: Affects how many asteroids are present within the asteroid field's area.
- <exclusion>: Defines an asteroid exclusion zone, creating a "negative" asteroid field. This can be used to create asteroid fields of unique shapes such as rings or crescents.
- <radius> and <pos> fields function identically to those under <asteroid>.

5.1.3 System Tags

TODO

5.1.4 Defining Jumps

Within Naev, jump points are used to travel between systems. Each jump point has a position within the system, defined either manually using the <pos> tag and x and y values or by using the <autopos/> tag to automatically place the point at a distance defined by the system's <radius>. Jump points also have an entry vector, or direction which ships must be facing to begin a jump. This entry vector is dictated by the position of the destination system on the universe map relative to the current system - that is, a jump point will always point towards its destination system.

To create a standard two-way jump lane between two systems: 1) Within current system *a*, create a <jump>. Use the <target> tag to name destination system *b*. Use the <autopos/> tag to automatically place the jump point, or the <pos> tag to manually define its position with X and Y values. 2) Repeat the above in system *b* to create a jump point, using the <target> tag to name destination system *a*.

5.1.5 Asteroid Fields

Asteroid fields are zones of floating objects within systems. They differ from spobs in that they are defined as circular areas rather than single points with

graphics. Asteroids also interact with ship weapons fire and often generate commodity pickups when destroyed.

Asteroid data files are found in `/asteroids/types/`. These files are in .XML format and contain the following fields: * `<scanned>`: Text string shown to the player upon entering range of their asteroid scanner outfit. * `<gfx>`: Possible graphics for this asteroid. Multiple graphics can be referenced, one per `<gfx>` tag, to increase the variety of visuals. * `<armor_min>` and `<armor_max>`: Defines a range of armor values for asteroids to spawn with. Higher values mean more damage must be dealt to destroy an asteroid. * `<absorb>`: Defines the asteroid's damage reduction before applying weapons' armor penetration stats. * `<commodity>`: Lists which commodity pickups and quantities thereof can spawn upon destruction of the asteroid. * `<name>`: Name of commodity. * `<quantity>`: Maximum quantity of commodity pickups

This process will let you create an asteroid field in your `<ssys>` .XML file: 1) Place graphics for your asteroids, in .WEBP format, to `/gfx/spob/space/asteroid/`; 2) Write asteroid data files, in .XML format, to `/asteroids/types/`; 3) Write an asteroid group list, in .XML format, to `/asteroids/groups/`. 4) In your `<ssys>` .XML file, use the `<asteroid>` field and subfields above to tell the game what asteroids the field will be made of. `<pos>` and `<radius>` define the position and size of your field. `<group>` and `<density>` define which asteroid group and how many asteroids appear in your field.

5.2 System Objects (Spobs)

You can either create spobs manually by copying and pasting existing spobs and editing them (make sure to add them to a system!), or create and manipulate them with the in-game editor (requires `devmode = true` in the config file or running naev with `--devmode`). Note that the in-game editor doesn't support all the complex functionality, but does a large part of the job such as choosing graphics and positioning the spobs.

5.2.1 System Editor

TODO

5.2.2 Spob Classes

Naev planetary classes are based on Star Trek planetary classes[1].

[1] `https://stexpanded.fandom.com/wiki/Planet_classifications`

Station classes:

- Class 0: Civilian stations and small outposts
- Class 1: Major military stations and outposts
- Class 2: Pirate strongholds
- Class 3: Robotic stations
- Class 4: Artificial ecosystems such as ringworlds or discworlds

Planet classes:

- Class A: Geothermal (partially molten)
- Class B: Geomorteus (partially molten, high temperature; Mercury-like)
- Class C: Geoinactive (low temperature)
- Class D: Asteroid/Moon-like (barren with no or little atmosphere)
- Class E: Geoplastic (molten, high temperature)
- Class F: Geometallic (volcanic)
- Class G: Geocrystaline (crystalizing)
- Class H: Desert (hot and arid, little or no water)
- Class I: Gas Giant (comprised of gaseous compounds, Saturn-like)
- Class J: Gas Giant (comprised of gaseous compounds, Jupiter-like)
- Class K: Adaptable (barren, little or no water, Mars-like)
- Class L: Marginal (rocky and barren, little water)
- Class M: Terrestrial (Earth-like)
- Class N: Reducing (high temperature, Venus-like)
- Class O: Pelagic (very water-abundant)
- Class P: Glaciated (very water-abundant with ice)
- Class Q: Variable
- Class R: Rogue (temperate due to geothermal venting)
- Class S: Ultragiant (comprised of gaseous compounds)
- Class X: Demon (very hot and/or toxic, inhospitable)
- Class Y: Toxic (very hot and/or toxic, inhospitable, containing valuable minerals)
- Class Z: Shattered (formerly hospitable planet which has become hot and/or toxic and inhospitable)

5.2.3 Spob XML

- `<spob>`: Category which encapsulates all tag data relating to the spob.
- `<lua>`: Runs a Lua script in relation to this spob.
- `<pos>`: Position of the spob within its parent system, defined by `x` and `y` coordinates relative to the system center.
- `<GFX>`: Category relating to graphics.

- `<space>`: Defines the image, in .WEBP format, which represents the spob when travelling through the parent system. The dimensions of the graphic can also influence the area at which a ship can begin its landing sequence.
- `<exterior>`: Defines the image, in .WEBP format, displayed on the spob's "Landing Main" tab.
- `<presence>`: Category relating to faction presence, used to generate patrol lanes within the parent system.
- `<faction>`: Defines the spob's owning or dominant faction.
- `<base>`: Defines the base presence of the spob. The maximum base presence of all spobs of the same faction is used as the base presence of the faction in the system. For example, if there are two spobs with base 50 and 100 for a faction in a system, the system's base presence for the faction is 100 and the 50 value is ignored.
- `<bonus>`: Defines the bonus presence of the spob. The bonus presence of all the spobs of the same faction in a system are added together and added to the presence of the system. For example, for a system with a base presence of 100, if there are two spobs with a bonus of 50 each, the total presence becomes $100 + 50 + 50 = 200$.
- `<range>`: The range at which the presence of the spob extends. A value of 0 indicates that the presence range only extends to the current system, while a presence of 2 would indicate that it extends to up to 2 systems away. The presence falloff is defined as $1 - \frac{dist}{range+1}$, and is multiplied to both base presence and bonus presence. For example, a spob with 100 presence and a range of 3 would give 75 presesnce to 1 system away, 50 presence to 2 systems away, and 25 presence to 3 systems away.
- `<general>`: Category relating to many functions of the spob including world statistics, available services, etc.
- `<class>`: Defines the spob's planetary or station class as listed above in the Station Classes and Planetary Classes categories above. This may be referenced by missions or scripts.
- `<population>`: Defines the spob's habitating population.
- `<hide>`: Modifies the range at which your ship's sensors can first discover the spob. A value of `1` is default range; values greater than 1 make it easier while values between 1 and 0 make it more difficult. A spob with a `hide` value of `0` will automatically reveal themselves to the player upon entering the system.
- `<services>`: Defines which services are available to the player while landed at the spob.
- `<land>`: Includes the Landing Main tab and allows the player to land on

the spob. A spob without the `land` tag cannot be landed on.
- `<refuel>`: Refuels the player's ship on landing. A landable spob without this tag will not generate an Autosave (and will warn the player of this) to mitigate the chances of a "soft lock" where the player becomes trapped in a region of systems with no fuel sources and no autosaves prior to entering said region.
- `<bar>`: Includes the Bar tab, allowing the player to converse with generic or mission-relevant NPCs and view a news feed. Certain spob tags may alter the availability of NPCs and the news.
- `<missions>`: Includes the Mission Computer tab, where the player can accept generic missions.
- `<commodity>`: Includes the Commodities Exchange tab, where the player can buy and sell trade goods.
- `<outfits>`: Includes the Outfitter tab, allowing the player to buy and sell ship outfits. Also grants access to the Equipment tab where the player can swap outfits to and from their active ship.
- `<shipyard>`: Includes the Shipyard tab, allowing the player to purchase new ships. Grants access to the Equipment tab as above; also allows the player to swap their active and fleet ships and change the oufits on all player-owned ships.
- `<commodities>`: Declares the spob as having ready access to commodities, independent of the Commodities Exchange service.
- `<description>`: Text string presented to the player on the Landing Main tab. This text body is perhaps the primary method of presenting the spob's lore to the player.
- `<bar>`: Text string presented to the player on the Bar tab. Compared to the `description` tag's lore regarding the spob as a whole, this text describes only the Spaceport Bar and its surroundings.
- `<tech>`: Category which includes Tech Lists, used to define the items in stock at the Outfitter and Shipyard.
- `<item>`: Includes one Tech List.
- `<tags>`: Category which includes tags that describe the spob. These tags can be referenced in missions and scripts; see the Spob Tags section below for more information.

5.2.4 Spob Tags 

Tags are a versatile way to define the main facets of interest about a spob with respect to its faction, i.e. what differentiates it from the other spobs the player will (try and) visit.

Tags consist of binary labels which are accessible through the Lua API with

`spob.tags()`. They are meant to give indication of the type of spob, and are meant to be used by missions and scripts to, for example, get specific spobs such as Dvaered mining worlds to send the player to get mining equipment or the likes.

Tags can be defined by the following part of XML code:

```
<tags>
 <tag>research</tag>
</tags>
```

where the above example would signify the spob is focused on research.

Special Tags

These tags significantly change the functionality of the spob:

- **restricted**: player should not normally have access here, and normal missions shouldn't spawn or try to come to the spob
- **nonpc**: there should be no normal generic NPCs spawning at the spaceport bar
- **nonews**: there is no news at the spaceport bar

Descriptive Tags

Below is the complete list of dominantly used descriptive tags. It should be noted that tagging is incomplete at present and it is possible that none of these tags will apply to many spobs (e.g. uninhabited, average, uninteresting or deserted spobs). Most others will only have one or two tags - they are supposed to represent important facets of the spob in its own estimation, not minor elements e.g. while the (temporary) Imperial Homeworld has many criminals and military personnel neither tag applies since its defining tags would be rich, urban and maybe tourism or trade.

- **station**: the spob is a space station or gas giant spaceport
- **wormhole**: the spob is a wormhole
- **hypergate**: the spob is a hypergate
- **active**: the spob is active (currently only matters for hypergates)
- **ruined**: the spob is ruined (currently only matters for hypergates)
- **new**: recently colonised worlds / recently built stations (definitely post-Incident)
- **old**: long-time colonised worlds / old stations (definitely pre-Incident)
- **rich**: the population living on the spob is rich by the standards of the faction
- **poor**: the population living on the spob is poor by the standards of the faction

- **urban**: the spob consists of mainly heavily developed cities and urban environments
- **rural**: the spob consists of mainly undeveloped and virgin lands
- **tourism**: spob has interests and draws in tourists
- **mining**: mining is an important part of the spob economy
- **agriculture**: agriculture is an important part of the spob economy
- **industrial**: industry is an important part of the spob economy
- **medical**: medicine is an important part of the spob economy
- **trade**: trade is an important part of the spob economy
- **shipbuilding**: shipbuilding is an important part of the spob economy
- **research**: the spob has a strong focus in research (special research laboratories, etc…)
- **immigration**: the spob draws in a large number of immigrants or is being colonised
- **refuel**: the spobs reason for existance is as a fueling point
- **government**: the spob has important government functions or hosts the central government
- **military**: the spob has an important factional military presence
- **religious**: the spob has an important religious significance or presence
- **prison**: the spob has important prison installations
- **criminal**: the spob has a large criminal element such as important pirate or mafia presence

5.2.5 Lua Scripting

TODO

5.2.6 Techs

TODO

Chapter 6

Outfits

TODO

6.1 Slots

TODO

6.2 Ship Stats

TODO

6.3 Outfit Types

TODO

6.3.1 Modification Outfits

TODO

Chapter 7

Ships

Ships are tho cornerstone of gameplay in Naev. The player themselves is represented as a ship and so are all other NPCs found in space.

7.1 Ship Classes

Ships classes have an intrinsic size parameter accessible with the `ship.size()` Lua API. This is a whole integer number from 1 to 6.

In naev, small ships (size 1 and 2) use small core slots and are meant to be fast and small. Medium ships (size 3 and 4) use medium core slots and are still agile, while being able to pack more of a punch. Large ships (size 5 and 6) are slow hulking giants with heavy slots meant to dominate. There is always a trade-off between agility and raw power, giving all ships a useful role in the game.

Ships are also split into two categories: civilian and military. Civilian ships are meant to focus more on utility and flexibility, while military ships focus more on combat abilities.

An overview of all the ship classes is shown below:

- **Civilian**
  - **Yacht**: very small ship often with only few crew members (size 1)
  - **Courier**: small transport ship (size 2)
  - **Freighter**: medium transport ship (size 3)
  - **Amoured Transport**: medium ship with some combat abilities (size 4)
  - **Bulk Freighter**: large transport ship (size 5)
- **Military**
  - **Small**
    - **Scout**: small support ship (size 1)
    - **Interceptor**: ultra small attack ship (size 1)

        * **Fighter**: small attack ship (size 2)
        * **Bomber**: missile-based small attack ship (size 2)
    - **Medium**
        * **Corvette**: agile medium ship (size 3)
        * **Destroyer**: heavy-medium ship (size 4)
    - **Large**
        * **Cruiser**: large ship (size 5)
        * **Battleship**: firepower-based extremely large ship (size 6)
        * **Carrier**: fighter bay-based extremely large ship (size 6)

Note that it is also possible to give custom class names. For example, you can have a ship be of class `Yacht`, yet show the class name as `Luxury Yacht` in-game.

7.2 Ship XML

Each ship is represented with a stand alone file that has to be located in `ships/` in the data files or plugins. Each ship has to be defined in a separate file and has to have a single `<ship>` base node.

- `name` (*attribute*): Ship name, displayed in game and referenced by `tech` lists.
- `points`: Fleet point value. In general used by both the fleet spawning code and by player fleets.
- `base_type`: Specifies the base version of the ship, useful for factional or other situational variants. (For example, a Pirate Hyena would have the "Hyena" base type.
- `GFX`: Name of the ship graphic in `.webp` format. It is looked up at `gfx/ship/DIR/NAME`, where `DIR` is the value of `GFX` up to the first underscore, and `NAME` is the value of `GFX` with a special suffix depending on the type of image. The base image will use a suffix of `.webp` (or `.png` if the webp is not found), the comm window graphic will use a suffix of `_comm.webp`, and the engine glow will use a suffix of `_engine.webp`. As an example, for a value of `GFX="hyena_pirate`, the base graphic will be searched at `gfx/ship/hyena/hyena_pirate.webp`
    - `size` (*attribute*): The ship sprite's resolution in pixels. For example, `size=60` refers to a 60x60 graphic.
    - `sx` and `sy` (*attributes*): The number of columns and rows, respectively, in the sprite sheet.
- `GUI`: The in-flight GUI used when flying this ship.
- `sound`: Sound effect used when accelerating during flight.
- `class`: Defines the ship's AI when flown by escorts and NPCs.

    - `display` (*attribute*): Overrides the displayed ”class” field in the ship stats screen.
- `price`: Credits value of the ship in its ”dry” state with no outfits.
- `time_mod` (*optional*): Time compression factor during normal flight. A value of `1` means the ship will fly in ”real time”, <1 speeds up the game and >1 slows down the game.
- `trail_generator`: Creates a particle trail during flight.
    - `x`, `y` (*attributes*): Trail origin coordinates, relative to the ship sprite in a ”90 degree” heading.
    - `h` (*attributes*): Trail coordinate y-offset, used to modify the origin point on a ”perspective” camera.
- `fabricator`: Flavor text stating the ship’s manufacturer.
- `license` (*optional*): License-type outfit which must be owned to purchase the ship.
- `cond` (*optional*): Lua conditional expression to evaluate to see if the player can buy the ship.
- `condstr` (*optional*): human-readable interpretation of the Lua conditional expression `cond`.
- `description`: Flavor text describing the ship and its capabilities.
- `characteristics`: Core ship characteristics that are defined as integers.
    - `crew`: Number of crewmen operating the ship. Used in boarding actions.
    - `mass`: Tonnage of the ship hull without any cargo or outfits.
    - `fuel_consumption`: How many units of fuel the ship consumes to make a hyperspace jump.
    - `cargo`: Capacity for tonnes of cargo.
- `slots`: List of available outfit slots of the ship.
    - `weapon`, `utility` and `structure`: Defines whether the outfit slot fits under the Weapon, Utility or Structure columns.
        * `x`, `y`, and `h` (*attributes*) define the origin coordinates of weapon graphics such as projectiles, particles and launched fighters.
        * `size` (*attribute*): Defines the largest size of outfit allowed in the slot. Valid values are `small`, `medium` and `large`.
        * `prop` (*attribute*): Defines the slot as accepting a particular type of outfit defined by an .XML file in the `slots/` directory. The Naev default scenario includes `systems`, `engines`, and `hull` values for Core Systems, Engines, and Hull outfits which must be filled (if they exist) for a ship to be spaceworthy.
        * `exclusive=1` (*attribute*): Restricts the slot to accepting only the outfits defined by the `prop` field.

        * Inserting an outfit's `name` will add it to that outfit slot in the ship's "stock" configuration. This is useful for selling a ship with prefilled core outfits to ensure its spaceworthiness immediately upon purchase.
- `stats` (*optional*): Defines modifiers applied to all characteristics and outfits on the ship.
  - Fields here correspond to those in the `characteristics` category and the `general` and `specifics` categories on equipped outfits.
- `tags` (*optional*): Referenced by scripts. Can be used to effect availability of missions, NPC behavior and other elements.
  - `tag`: Each `tag` node represents a binary flag which are accessible as a table with `ship.tags()`
- `health`: Supercategory which defines the ship's intrinsic durability before modifiers from `stats` and equipped outfits. **Note that this node and subnodes are deprecated and will likely be removed in future versions. Use ship stats instead!**
  - `armour`: Armour value.
  - `armour_regen`: Armour regeneration in MW (MJ per second).
  - `shield`: Shield value.
  - `shield_regen`: Shield regeneration in MW (MJ per second).
  - `energy`: Energy capacity.
  - `energy_regen`: Energy regeneration in MW (MJ per second).
  - `absorb`: Reduction to incoming damage.

A full example of the naev starter ship "Llama" is shown below.
\lstinputlisting[language=XML]{../../dat/ships/neutral/llama.xml}

7.3 Ship Graphics

Ship graphics are defined in the `<GFX>` node as a string with additional attributes like number of sprites or size also defined in the XML. Graphics for each ship are stored in a directory found in `gfx/ship/`, where the base graphics, engine glow graphics, and comm window graphics are placed separately with specific file names.

In particular, the `GFX` string name is sensitive to underscores, and the first component up to the first underscore is used as the directory name. As an example, with `<GFX>llama</GFX>`, the graphics would have to be put in `gfx/ship/llama/`, while for `<GFX>hyena_pirate</GFX>`, the directory would be `gfx/ship/hyena`. The specific graphics are then searched for inside the directory with the full `GFX` string value and a specific prefix. Assuming `GFX` is the graphics name and `DIR` is the directory name (up to first underscore in

GFX), we get:

- gfx/ship/DIR/GFX.webp: ship base graphic file
- gfx/ship/DIR/GFX_engine.webp: ship engine glow graphics file
- gfx/ship/DIR/GFX_comm.webp: ship communication graphics (used in the comm window)

The base graphics are stored as a spritesheet and start facing right before spinning counter-clockwise. The top-left sprite faces to the right and it rotates across the row first before going down to the next row. The background should be stored in RGBA with a transparent background. An example can be seen in Figure 7.3.

\begin{figure}[h!] \centering \colorbox{black}{\includegraphics[width=0.8\linewidth]{images/llama.png}} \caption{Example of the ship graphics for the ”Llama”. Starting from top-left position, and going right first before going down, the ship rotates counter-clockwise and starts facing right. A black background has been added for visibility.} \end{figure}

The engine glow graphics are similar to the base graphics, but should show engine glow of the ship. This graphic gets used instead of the normal graphic when accelerated with some interpolation to fade on and off. An example is shown in Figure 7.3.

\begin{figure}[h!] \centering \colorbox{black}{\includegraphics[width=0.8\linewidth]{images/llamaengine.png}} \caption{Example of the engine glow graphics for the ”Llama”. Notice the yellow glow of the engines. A black background has been added for visibility.} \end{figure}

The comm graphics should show the ship facing the player and be higher resolution. This image will be shown in large when the player communicates with them. An example is shown in Figure 7.3.

\begin{figure}[h!] \centering \includegraphics[width=0.8\linewidth]{images/llamacomm.png} \caption{Example of the comm graphics for the ”Llama”.} \end{figure}

7.3.1 Specifying Full Paths

It is also possible to avoid all the path logic in the <GFX> nodes by specifying the graphics individually using other nodes. In particular, you can use the following nodes in the XML in place of a single <GFX> node to specify graphics:

- <gfx_space>: Indicates the full path to the base graphics (gfx/ is prepended). The sx and sy attributes should be specified or they default to 8.
- <gfx_engine>: Indicates the full path to the engine glow graphics (gfx/ is prepended). The sx and sy attributes should be specified or they default to 8.

- `<gfx_comm>`: Indicates the full path to the comm graphics (`gfx/` is prepended).

This gives more flexibility and allows using, for example, spob station graphics for a ”ship”.

7.4 Ship Conditional Expressions

TODO

7.5 Ship trails

TODO

7.6 Ship Slots

TODO

Part II

Naev “Sea of Darkness” Lore 

Chapter 8

Introduction to Naev Lore

This document refers to the lore of the Naev base setting known as **Sea of Darkness**. Note that the lore is presented here with **heavy spoilers**. Do not continue reading if you do not wish to be spoiled.

Chapter 9

Universal Synchronized Time (UST)

Universal Synchronized Time (UST) is the standard time system in Naev.

9.1 Explanation

UST consists of three basic components describing different amounts of time: the second, which is equivalent to an Earth second; the period, which is equal to 10,000 seconds; and the cycle, which is equal to 5,000 periods (50,000,000 seconds). In colloquial usage, the terms ”decaperiod” (equivalent to 10 periods) and ”hectosecond” (equivalent to 100 seconds) are also common. UST dates are written in the form:

- UST C:PPPP.SSSS

Where ”C” is the cycle, ”PPPP” is the period (always displayed as four digits), and ”SSSS” is the second (always displayed as four digits). So for example, the following hypothetical date indicates cycle 493, period 42, second 2089 (which is about 100 cycles prior to [[The Incident]]):

- UST 493:0042.2089

When describing lengths of time, it is commonplace for computer systems to indicate a number of periods with a lowercase ”p” or a number of seconds with a lowercase ”s”. This convention is not used for cycles, which are always spelled out in full as ”cycles”. This convention is also not adopted in spoken form since it’s more natural to just say ”periods” and ”seconds” rather than a single-letter abbreviation. Additionally, due to the metric nature of the time system, periods and seconds can be written out as a single unit, although in spoken conversation people report the periods and cycles separately. Some examples:

- 783p (read as ”783 periods”)
- 42s (read as ”42 seconds”)
- 12.0456p (read as ”12 periods and 456 seconds”)

The following is a chart of all time units used in Naev along with the corresponding Earth time unit they are similar to in terms of where they are used.

UST unit	Abbreviation	Length of Time	Equal to (in Earth time)	Used like
Seconds	”s”	1 Earth second	1 second	Seconds
Hectoseconds	N/A	100 seconds	1 minute and 40 seconds	Minutes
Periods	”p”	10,000 seconds	~2 hours and 47 minutes	Hours
Decaperiods	N/A	10 periods	~28 hours	Days
Cycles	N/A	5,000 periods	~579 days	Years

9.2 Time passage

Following is a list of actions and how much time they take in Naev.

- **Flying in space**: For ships with a time dilation rate of 100% (that is, most small ships), time passes at a rate of 30 seconds per real-world second, which is why the GUI’s clock increases by 0.01p every 3 seconds. For ships with higher time dilation the passage of time is faster, and for ships with lower time dilation the passage of time is slower.
- **Landed**: Time does not pass while landed.
- **Takeoff**: Taking off takes 1 period, which means that stopping to refuel during time-sensitive missions is generally a bad idea.
- **Jumping**: Hyperspace jumps also take time, generally 1 period per jump, though some ships such as the Quicksilver take less.

9.3 History of Humanity in Naev

Notable Events:

1. **Sirichana reaches Murtis** (UST -143)
2. **Imperial Proclamation** (UST 0): Creation of the Empire
3. **Project Proteron** (UST 13:4355)
4. **Project Za’lek** (UST 42:6284)
5. **Creation of House Proteron** (UST 47)
6. **Creation of House Za’lek** (UST 72)
7. **Project Thurion** (UST 84:8324)
8. **Creation of House Sirius** (UST 97)
9. **Project Collective** (UST 266:7626)
10. **Dvaered Revolts** (UST 307)
11. **Sorom Plague** (UST 328)

12. **Creation of House Dvaered** (UST 331)
13. **Quarantine of Sorom** (UST 333)
14. **Formalization of Soromid Faction** (UST 387)
15. **Hypergate Project** (UST 572)
16. **Collective Goes Rogue** (UST 590)
17. **The Incident** (UST 593:3726.4663)

9.3.1 The First Growth (UST -1000? to UST -400)

The First Growth is seen as the true beginning of mankind's space age. Though space travel existed before this time, it was limited to Earth's immediate environs, and bore little in the way of fruit.

When Earth scientists devised an effective means of crossing the interstellar void, using advanced, long-life sublight engines and a non-lethal method of cryogenic suspension, mankind began to dream big dreams. Over the course of fifty years, massive starships were constructed that could carry human life to other planets and start new colonies there. In total, twenty such ships were constructed, and eventually eleven of those managed to seed new human colonies in outer space. It was an achievement never before witnessed in human history.

For a long time, the eleven colonies were on their own, as communication with each other and with Earth was a matter of years. That would change, abruptly and dramatically.

9.3.2 The Second Growth (UST -400 to UST -100)

While the eleven colonies painstakingly tried to develop themselves into economies resembling Earth's, Earth itself continued to progress scientifically. Though no new colonization efforts were made after the first twenty, as this was deemed too costly, different avenues of deep space exploration were being invented, tested and discarded on a regular basis. Then, there was the breakthrough that lies at the foundation of space travel as we know it today: the discovery of hyperspace.

Hyperspace was found to allow travel between one point to another point without having to cross the space in between. More importantly, the time taken to complete the journey in hyperspace was a fraction of the time it would normally take. Soon, the existence of naturally occurring hyperspace connections between systems were discovered, which sealed the deal. The stars were now within reach. Indeed, the entire galaxy lay open for humanity to claim.

Hyperspace-capable starships were built, and sent out to explore. They brought back reports of many potentially habitable worlds ready for the taking. Soon after, a new colonization program was devised, and mankind truly started spreading its wings. The Second Growth had begun.

As humanity settled farther and farther away from Earth, a decision was made to streamline the interplanetary relations by creating a large, democratic body to unite all worlds in a single political system. This body was known as the Federation.

9.3.3 The Federation (UST -300 to UST -100)

Though each colony was granted the right of self-government, the Federation was tasked with managing interplanetary affairs. Interstellar trade and security came to fall under the Federation's jurisdiction, and each standard cycle the planetary governments would convene in an interstellar summit, discussing the current state of affairs. On paper, it looked good.

Over the course of many years, many new planets were settled, converted or exploited. Interstellar trade became immensely lucrative, new fortunes were made. But humanity, even in times of prosperity, tends to strife and conflict. Not only did piracy manifest itself in the vast stretches of space, there appeared a growing discontent between the various colonies. The Federation proved increasingly less capable of dealing with the security concerns and the discontent between the various worlds. Gradually, the colonies began to lose faith in Federal leadership, and began to band together in local alliances that guarded their own interests. These interests began to conflict with each other. Tensions grew higher and higher, until eventually the situation degenerated into armed conflict.

9.3.4 The Faction Wars (UST -100 to UST 0)

The Faction Wars are the first interstellar war on record, and they also count as the largest human war ever waged. Truly every human world was at war, and precious few were safe from attack. Even Earth itself was bombed on two occasions. Any colony that couldn't fight off its attackers was conquered and claimed. Colonies changed ownership time and again, factions were eliminated, new factions formed as colonies rose against their oppressors. Untold human lives were lost. Needless to say, no new worlds were settled during this time, as attempting to do so would mean certain death.

After many, many cycles of constant fighting, the Faction Wars stabilized into a three way conflict between factions calling themselves the Earth Federation, the Free Colonies, and the Rimward Block. These factions were similar

in strength, and for a time there was a balance of power. Many believed that eventually hostilities would cease, and new political and trade relations would establish themselves. But this didn't happen.

A general serving in the Earth Federation called Duram Daedris devised a cunning strategy. Through deceit and trickery he managed to entice both the Free Colonies and the Rimward Block to launch an all-out offensive on the same system, at the same time. The two fleets clashed, as was Daedris' plan, decimating each other. This gave the Earth Federation the opening it needed to take the initiative and gain the upper hand. The other two factions couldn't recover in time. The Earth Federation had won the Faction Wars.

9.3.5 Rise of the Empire (UST 0 to UST 300)

After his military victory, Daedris took political control in a military coup. The Earth Federation was re-branded the Empire, and Daedris proclaimed himself Emperor over all human space, ruling over the Galaxy from his throne on Earth. For cycles, order was kept with an iron fist, while at the same time trade was brought back to pre-war capacity. Eventually, the colonies accepted that to live under Imperial rule was better than to die in autonomy.

The vertical chain of command installed by the Empire proved to work better than the democratic ideals of the old Federation. A period of extended peace and prosperity began, which would later be referred to as the Imperial Golden Age. The wounds left by the Faction Wars slowly healed over time, and eventually humanity began to expand anew in an ambitious Third Growth.

9.3.6 Decline of the Empire (UST 300 to UST 593)

Though the Empire was the greatest and most stable political system in all of human history, it did not prove strong enough to stand the test of time. Little by little, as the Empire grew and its Emperors made ill-advised decisions, dissent crept into the minds of the Imperial citizens. Piracy once again started rearing its ugly head, and some worlds started growing restless. Historians are still debating what ended the Imperial Golden Age, but all agree that by the time the working class rose in what became known as the Dvaered Revolts, it was well and truly over.

The Empire found itself forced to cede territory and political control to the newly formed House Dvaered, to the mysterious figure known as Sirichana and to the ever-demanding Za'lek. Though all human worlds remained loyal to the Empire by treaty, the sphere of influence of the line of Emperors had dwindled considerably. What really broke the Empire's power, though, was a sudden, cataclysmic event known only as the Incident.

9.3.7 The Incident (UST 593:3726.4663)

Little is known about the Incident, other than what can be observed. An eruption of some kind occurred which decimated all planets in Sol and several systems around it. In its wake it left a dense, volatile nebula that has proven almost impregnable to most means of observation. Nobody ever came out of that nebula to tell what happened.

The Empire was shaken to the core. With Earth lost, along with most of the Imperial bureaucracy, the Imperial leadership floundered, taking almost a cycle to re-establish itself in Gamma Polaris. Such a show of weakness caused whatever loyalty the Empire had left from the Great Houses to evaporate, to the point that it's now a public secret that the Empire no longer holds any sway over anybody else.

This was several cycles ago. The galaxy is now an unstable place, full of danger and opportunity. Nobody knows what the future holds, but perhaps one person can make all the difference.

Chapter 10

The Empire

The Empire is one of the major factions in the Naev universe. Its governmental system is a mixture of a republic and a monarchy, with The Emperor presiding over all other powers of state. The Emperor delegates some of his power to the Imperial Council, consisting of 20 councilors who pass laws to be carried out by the Imperial bureaucracy which commands the Imperial army and governs the citizens. However, all of the councilors can be overruled by The Emperor. From The Empire came the Great Houses, and they interact with the Imperial government through their respective liaisons.

10.1 The Facts

- Leader: The Emperor
- Leading Structure: Imperial Council
- Government: Republic Monarchy
- Formation: UST 0
- Homeworld: Emperor's Wrath (Gamma Polaris)
- Important Figures:
  - Emperor

10.2 Government

In accordance with the backstory of Naev, the modern Empire is but a shadow of what it formerly was. Its government type can be described as one half despotism, one half republic and one half paralyzing bureaucracy. The Emperor is the head of state as well as the Ruler of all Mankind, and he has absolute power, on paper, at least. In reality, he is an isolated figure, interacting only with his Imperial Council. The Emperor is above the Council and

can pass any edict he wishes without being challenged, but in practice a lone figure cannot directly govern a domain spanning many star systems, even in its present, reduced state. As a result, the councilors often pass laws at their own discretion, with little oversight from the Emperor himself. However, they too remain isolated, dealing with the theoretical implications of their work, rarely bothering to verify that their actions have the intended result, and as such their labors typically yield inefficacy.

10.3 Interaction with the Houses

The Council also interacts with the Great House liaisons, who represent the ”vassals of the Empire”. Whenever the Emperor requires one or more of the Great Houses to do his bidding, the Council will pass on the Imperial Decree to the relevant liaisons, who in turn convey the message to their respective leaders. This is often only a courtesy however, as the Great Houses are no longer bound to the core of the Empire beyond ancient treaties and vows of allegiance. Typically, a Great House will put on a token show of loyalty, and subsequently ignore the Emperor’s will to pursue its own agenda.

10.4 Imperial Bureaucracy

The Imperial Bureaucracy is what makes the Imperial worlds tick, though ”tick” is a big word for what often boils down to barely managing to keep society from grinding to a standstill. The knotted maze of rules and regulations constructed by generation after generation of councilors makes upholding the law an impossible task, and as such most local governments operate on what they believe is a distillate of the core of the book of law. Needless to say, this distillate varies from one world to the next.

The Bureaucracy also manages the Imperial military, with rather more success than it does civilian life. This is in part due to the Emperor and his Council’s jealous guarding of their territory (despite not caring too much about what goes on inside it), and in part because the Imperial military machine often knows what practical actions must be taken to best serve the Emperor’s ends, though the ruling body often does not. Though in itself deeply hierarchical, the Imperial military is ultimately an effective, if not terribly efficient, machination that commands sufficient respect to at least keep the Minor Houses in line.

10.5 In-Game Database

The following section is written from an **in-universe** perspective. It may contain biased information or omit facts for dramatic purposes.

10.5.1 History

The Empire began with the first, self-proclaimed Emperor Daedris who used his forces to do battle with and eventually unite many of the the factions at war during the Faction Wars. His success meant the end of the Faction Wars and started the period now known as the Third Growth.

The following period of peace was a Golden Age of The Empire, with new trade routes bringing wealth and prosperity to Empire worlds, patrols effectively extinguishing piracy and, apart from minor incidents, peace reigning in all of Empire Space.

That Golden Age ended when the Empire began to weaken and succumb to its own bureaucratic tendencies, in turn losing control over some of its territories. Notably, the Dvaered Revolts were a clear sign of the Empire's reduced ability to keep law and order.

The Empire suffered catastrophic damage from The Incident. Many Empire worlds were devastated in the initial blast, and many stations and planets are now caught in the Sol Nebula. The Empire's power has declined since then and the security that was ensured in the Golden Age has diminished greatly as the Imperial military, once the guardian of much of inhabited space, now finds itself struggling to keep the core systems of the Empire's former territory secure, let alone the sparsely-populated border systems or the vast expanses beyond.

The current Emperor is a man called Eilo Cedona. He was elected Emperor by what remained of the Council shortly after the Incident, and has held power ever since. He is fairly inexperienced and refuses to acknowledge that the Empire is in decline. It was he who had the Emperor's Wrath built, and who initiated the Emperor's Fist project, despite the state the rest of his realm is in. Many believe he will not be Emperor much longer.

10.5.2 Territory

The Empire still has the largest territorial claim of all factions, holding a major part of the inhabited Galaxy centered around Polaris Prime in the Gamma Polaris system, the official seat of The Emperor. It is there that all important decisions of the Empire are made. However, the Empire holds outposts deep in hostile space, like Cerberus Station in the Doeston system or Zabween

in Draconis. The core systems, those in the vicinity of Gamma Polaris, are heavily patrolled and kept safe by the Empire military. However, frequent patrols to the outer systems are a logistical impossibility, and the Empire relies on its allies to keep traders safe there. Still, skirmishes between Imperial forces and pirate raiders are not uncommon. It is a major goal of the Empire to restore the security of its territory to levels present prior to the Incident.

10.5.3 Economy

Civilian traders will not usually be able to trade with the Empire directly, as that trust is only given to individuals or factions enjoying a high reputation with the Empire. After a recruitment process, individuals can be tasked with running minor shipping assignments. Otherwise, the Empire holds trading treaties with many important factions, including the Traders, the Dvaered (and the Consortium). The Empire has a policy of interfering as little as possible with the local economy of planets, beyond monitoring that there is no contraband transported to or from them. Other than that, what commodities companies produce on planets is of no concern to the Empire.

10.5.4 Science and Technology

Most of the scientific research is conducted by House Za'lek, but the Empire still has research bases of its own. It remains largely unknown to the public what projects the Imperial researchers develop. Empire engineers have created the Peacemaker, a capital ship design which only the most respected among the Empire military are permitted to captain, and which is unequaled in combat by any other know vessel. These are the flagships of any larger Empire fleets, while local patrols usually consist primarily of Admonishers and Lancelots.

10.5.5 Political System

The Emperor is ultimately the seat of all power and has the ability to overrule anyone, but a lone man cannot single-handedly govern an entity the size of the Empire at any meaningfully low level. As such, the Imperial Council sits directly below the Emperor, regulating and passing laws to ensure the Empire's continued existence. To be a councilor is a highly sought-after position, as upon the Emperor's death, the new Emperor is selected from their ranks.

The Imperial Bureaucracy

The council does not oversee the laws being carried out, that task falls to the Imperial bureaucracy. It is the largest part of the Empire administration besides the military. Enforcing the large number of laws passed by generations of councils word for word is impossible. Therefore, most members of the bureaucracy follow what they in their best judgment believe to be the idea behind the laws. But even that task is becoming ever more difficult and social progress has been observed to become slower in recent years. The bureaucracy is also the only organ of the state the people interact with directly. Requests for audiences with the council or similar may be directed at the bureaucracy, though they are seldom granted and some have been known to simply disappear within the immense, monolithic complexity of the bureaucracy. Finally, it is the bureaucracy's task to manage the Empire military, which it does with some success. The Emperor and the Council place great value on the defense of their borders and the bureaucracy must ensure that those wishes are carried out. The military works quite well on their own and usually requires little oversight. The Generals and Commanders of the army are known to work with great efficiency, thus, the military is often called the most efficient part of the Empire administration.

The Great Houses

The Houses are still bound to the Empire via contracts but are effectively factions of their own with their own government and economy. They interact with the Imperial countries via Liaisons, chosen representatives of the Houses' governments. When the Emperor has a particular request to one of the houses, the Council passes his decree on to the responsible Liaison, who in turn passes it on to his superiors. This is by no means a binding contract or even an order. Today, the Houses are not required to follow the Emperor's orders and may decide for themselves whether or not to grant the Emperor's request.

Minor Houses

Next to the Great Houses, some Minor Houses exist, as well. The status of a Minor House is given for various reasons like an achievement or as is the case with the House Goddard, in return for a service.

Chapter 11

Great House Dvaered

House Dvaered is one of the major factions in the Naev universe. It evolved from the simple working class of the Empire during the Second Growth and the Faction Wars and was granted the status of a Great House by the Emperor during the golden age of the Empire. Today, it is governed by the military and the armed forces dominate much of the everyday lives of the Dvaered citizens. Planets are governed by Warlords, former members of the Dvaered High Command, which makes all the most important decisions about the House. The Warlords regulate life on a local basis, while the High Command directs the Faction as a whole. The only way of rising up is usually to join the military and earn medals and commendations.

11.1 The Facts

- Leader: None
- Leading Structure: High Command Generals
- Government: Meritocracy
- Formation: UST 331
- Homeworld: Dvaered High Command (Dvaer system)
- Important Figures:
  - Warlords

11.1.1 History

House Dvaered came forth from a lower caste in the old Empire, a collection of miners, manual laborers, foot soldiers and outcasts. With the gradual decline in authority of the Empire, the working class became increasingly disgruntled with their social position. They banded together into what could

be considered the biggest labor union in all of human history, and started demanding better treatment.

When the Empire was less than forthcoming, the laborers began to take more drastic measures. The labor union turned into a resistance movement that escalated into open rebellion on several worlds. History names this movement the Dvaered Revolts. The origin of that name is unclear, but it became common among the rebelling workers.

Cycles of civil unrest on the working planets and fruitless attempts on the part of the Empire to quell it eventually led the Emperor of that time to agree to the Dvaereds' demands. Their leaders were granted the right to establish a Great House, and direct control over a sizeable chunk of space, in reparation for the lives lost during the Dvaered Revolts.

Eager to claim their place among the other big players, the then-time Dvaered leadership decided that discipline was the only proper way to shape a society. And so House Dvaered evolved into a military regime, and the values of honor and strength were etched into the Dvaered soul.

11.1.2 Government

Today, House Dvaered is ruled by the Generals of the Dvaered military. The amount of Generals varies from time to time, but there are always enough to feed the fires of internal discord. Influence is usually determined by the amount of medals and commendations a General has collected in his career. This at least stands undisputed, for House Dvaered has extensive protocols for dispensing awards. In fact, it can be considered a constitution of sorts.

Warlords

When a General retires from High Command, he often takes with him a small contingent of the armed forces. These are soldiers and captains who have sworn allegiance to the man, not the banner, and they will continue to serve until the bitter end. The General, now known as a Warlord, will then use his little private army to secure rule over one or more Dvaered worlds, usually by usurping the position from another Warlord. This method of local government is commonly accepted to be legal, and no Dvaered citizen will be surprised when suddenly a new set of local laws will be passed to replace another.

Citizens

The Dvaered citizenry itself is barely worthy of the name. There is often more squalor than culture, and more often than not the Dvaered will seek

employment in the military, despite the dangers that represents. Nevertheless, there certainly is a Dvaered elite. They often concentrate on the few truly hospitable worlds in Dvaered space, where they pursue their idea of luxury.

11.2 Warlords and Dvaered High Command

While every Dvaered controlled world is governed by a Warlord at any given time, it is Dvaered High Command that the Warlords ultimately answer to. Though Dvaered High Command is more military than government, it needs a steady stream of funds, manpower and materials to further its agenda. Each Warlord is required to pay a certain portion of his worlds' resources in tax. Failure to do so results in a swift and permanent removal by Dvaered High Command, after which the worlds previously held by that warlord become available to any other Warlords with the will to take them (which is to say, all of them).

11.3 How the Dvaered fight in space

11.3.1 Summary

- Dvaered fleets' main strategy consists in destroying enemy heavy ships in order to force lighter ones to retreat.
- Most Dvaered ships' characteristics are: heavy, slow, bad manoeuvrability, good armour, good cannons.
- Exception: the Phalanx has no speed malus and can more or less keep up with the bombers and fighters to support them with turreted missiles. But it has very bad manoeuvrability and as a result is nearly forced to use turrets.
- Civilian versions are limited because of the high base weight of the designs, but military versions have higher engine mass limit to compensate. As a result, Dvaered ships are less able to stealth.
- Civilian Goddard is quite different from Dvaered version because it is used differently by civilians and Dvaered.
- Most Dvaered designs (except for the Goddard) are modified versions of other factions' obsolete ships.

11.3.2 General doctrine of the Dvaered space navy:

Dvaered have observed during their independence war that the destruction of supply ships and carriers is a safe way to force a fleet to retreat, and to preserve their planets from enemy disembarkment. Their military doctrine is based on direct and very powerful attacks on key assets of the enemy force, that makes the enemy position untenable. They do want to avoid entering in an attrition war as much as possible.

- Offensive doctrine: Contrary to what can be expected, Dvaered have never been the aggressor in any large scale war against an other consequent power. This is why the dvaered Generals have paradoxically not much confidence for the invasion of the Frontier. However, during the countless wars that have happened between Warlords, the Dvaered have tested many tactics, and the one that is favoured by the Dvaered generals is the following: A Dvaered attack fleet must use its superior firepower to damage, destroy or take control as fast as possible of the heavy enemy installations (bases on planets, stations or carriers). Without that support, the enemy lighter ships will eventually have to retreat from the system. Dvaered don't want to send expeditionary fleets far away from their space. Their fleets require to be at max at 2 jumps from their bases to operate. Ideally, in the same system.
- Defensive doctrine: If the objective of the enemy forces is the invasion, it can be expected that carriers and transports will head towards the allied assets. The goal for the Dvaered fleet will be to destroy those heavy ships as soon as possible in order to force the enemy to abort the invasion. Before those support ships show up, the dvaered ships must be as discrete as possible (stay at dock) in order not to be vulnerable to harassment from the enemy foreguard.

11.3.3 Consequence on the ships design:

Both in attack and defence, Dvaered pilots have to target heavier ships and to ignore lighter opponents that are going to harass them. This is why they favour forward weapons (their target is less manoeuvrable than they are) and require high armour in order to survive harassment from light ships. However, in the case when the enemy light ships try to interpose themselves instead of using missiles, the Dvaered pilots should take the opportunity to pick up the fight and destroy them, if possible in one pass. This requires very powerful cannons, and huge reserves of energy. With their massive attack-focused tactics, the Dvaered don't anticipate long fights. Consequently, the shields and their regeneration rate are not very important.

Dvaered ships are usually able to win a dogfight duel against any other ship of the same class.

11.3.4 Origin of the ships designs (except for the Goddard):

As stealth and speed are not prominent needs, the Dvaered engineers prefer to rely on outdated and well-known ship designs. These designs are then upgraded with more weapon slots, better energy storing, optimized cannons and better armour. After that, the engineers try to optimize the balance of the ship's mass in order to increase the maximal admissible payload. The unoptimized version is sold to civilians, while the optimized version is reserved to Dvaered pilots in order to grant them an advantage. The dvaered engineers are now working on the next generation of ships, mostly based on Empire designs (Lancelot, Pacifier, Hawking).

11.3.5 List of Dvaered Ships

- Vendetta (Fighter)
- Role: destroy bombers, swarm and engage medium ships from several directions and get opportunity shots on interceptors and fighters. As they fly in first line, they expect to eat many rockets during the approach phase.
- Characteristics: cheap, heavy, slow, good armour, good cannons (kills small ships in one pass)
- Interest for player: It is the best ship for a dogfight duel. However, its slower top speed makes it vulnerable to multiple lighter attackers, and to missiles.
- History: During the independence war, the Lancelot was a very new ship and the Bat used to be the standard Empire fighter. Dvaered engineers managed to adapt one more weapon slot on it, and to improve its armour, and renamed it the "Vendetta". After the war, a second weapon slot and more armour were added, and given the success of the program, the engineers decided to try and upgrade all empire designs the same way.
- Ancestor (Bomber)
- Role: Dvaered bombers fulfil two very different roles: the main one is to swarm and attack enemy capships at close range with powerful unguided torpedoes. The Dvaered don't like to use guided torpedoes,

because their use is too time-consuming for their radical attack tactics, and also because of their price. The second role of bombers is to defend a temporary static fleet against light targets (that their Vendettas cannot reach). In that case, they use Headhunter or Fury missiles to target Fighters and Bombers that use their own launchers against heavy Dvaered ships. This second role is however mostly taken by the corvettes.

- Characteristics: cheap, heavy, slow, bad manoeuvrability, good armour, good launchers
- Interest for player: civilian version is the only bomber easily available. It is very effective against heavy ships when used with torpedoes. Military version is an upgrade on the civilian version.
- History: This is a redesign of an old model that was obsolete before the civil war.
- Phalanx (Corvette)
- Role: Support fighters and bombers squadrons with turreted missiles, and bring cover if needed. Can also be used to skirt a blocus with supplies.
- Characteristics: heavy, bad manoeuvrability, good armour, good launchers, no speed malus (ie fast by Dvaered standards)
- Interest for player: A very capable corvette when equipped with turrets.
- History: After the independence war, analysts noticed that many fighters and bomber squadrons had been lost because of harassment by enemy light ships. This is why an engineering program was initiated from an imperial prototype recovered on one of the freshly conquered planets. This prototype was a fast but badly manoeuvrable corvette on which the dvaered managed to adapt more ammo space and armour. The design eventually became a missile-platform able to more or less catch up with Dvaered fighters in terms of top speed.
- Vigilance (Destroyer)
- Role: Destroy enemy medium ships to defend the fleet, or engage cruisers and carriers with railguns in attack.
- Characteristics: slow, heavy, bad manoeuvrability, good armour, good cannons

- Interest for player: When purposely-equipped, the Vigilance can destroy the casual Kestrel while not fearing lighter ships. As such, it's probably the lightest possible choice for pirate hunt.

- History: The Vigilance began its career as a Sirius prototype, with high defensive capabilities. At some point, the project was abandoned because it did not fit the needs of the Sirius army anymore. When House Dvaered was established, house Sirius sold them the project. Of course, Dvaered engineers made many changes to the design to make it suit better their own needs.

- Goddard (Battleship)

- Role: Neutralize Destroyers and up. Thanks to their advanced armour, the Goddards can in some circumstances be used as a ram by the rest of the fleet. What is more, all Warlords and generals of the Space Forces have a Goddard. When two generals have a deep disagreement, they may have a duel with their Goddards. As a consequence, they like to have good cannons.

- Characteristics: heavy, slow, good armour, good cannons

- Interest for player: This is the best choice to destroy heavy enemies escorted by light ships. The Goddard's armour can even survive several Caesar torpedoes.

- History: This ship is built on the Dvaered territory by employees of the Goddard company (and House). It is a many-time-updated version of the very old Goddard-class battlecruiser. Compared to the version sold to the public (and used by House Goddard), the Dvaered version is very different, with stronger cannons and armour, but heavier and worse CPU and shield

- Arsenal (Bulk Carrier)

- Role: Carry supplies and troops, mainly for ground operations.

- Characteristics: slow, bad manoeuvrability, good armour

- Interest for player: This ship has the best ratio cargo-space/vulnerability.

- History: After the independence war, Dvaered engineers designed this ship on the model of Melendez's Rhino, but bigger and tougher.

11.3.6 Needed Classes

- Scout (I believe they need a good one if they want to apply their military doctrine. Otherwise, they use the Schroedinger.)
- Cruiser: They probably need a railgun-truck ship.

11.3.7 Unused Classes

- Interceptor: Dvaered don't use Carriers nor harassment tactics. Consequently, they did not develop an interceptor. What is more, the concept of interceptor is recent, and they did not find a foreign obsolete design to adapt. They rely on foreign models (Hyena and Shark) in the rare occasions when they need fast ships.
- Carrier: Dvaered don't attack planets far from their bases. Consequently, they do not need carriers. If at some point they need to send a fleet far from their space, they need to borrow, annex or build a base before proceeding. Their lack of carriers is the main reason why the Dvaered never found the FLF base in the nebula. If really needed, Goddards and Vigilances can receive small bays to provide support for a limited number of light ships.

11.3.8 List of Dvaered Outfits

Used Weapons

- Gauss Gun, Vulcan Gun, Shredder and Mass Driver: Standard cannon suite for Dvaered ships. They are the primary equipment of fighters, and secondary for bombers. Mass Driver is part of the equipment of Destroyers as well. This equipment is not specific to Dvaered, even if most of those weapons are fabricated on their planets. Note that the Shredder (light corvette cannon) does not really fit into the Dvaered military doctrine, and as a consequence, it is rarely equipped on Dvaered ships.
- Railgun and Repeating Railgun: Heavy cannons that equip destroyers and up. It is a Dvaered specificity to put heavy forward weapons on their capships, what allows for a better damage/resource ratio, but makes it harder for their heavy ships to hit lighter enemies. Repeating Railgun is Dvaered-specific.
- Turreted Gauss and Vulcan Gun: Turret suite that serves as secondary weapons for corvettes and higher.

- Turreted Railgun: It does not fit into Dvaered military doctrine and is mainly fabricated for export.
- Flak turrets and forward shotguns: {unimplemented} Cannon and turret suite for Dvaered ships that expect to face many light adversaries. Those weapons are mainly developed in anticipation of a potential war against House Za'lek, and were not tested at large scale.
- Mace launchers: Used primarily as a short damage booster for fighters, they can also be a secondary weapon for bombers.
- Banshee launchers: They can be used to make a fighter able to threaten enemy heavy ships (or more realistically destroyers), or as secondary weapons for bombers.
- Repeating banshee launchers: Dvaered-specific. They don't carry more rockets than the standard launchers, but have a much higher firing rate. They are mainly used by bomber squadrons against heavy and medium ships. They can also be equipped as damage-boosters for destroyers or even capships.
- Fury and Headhunter launchers: Turreted versions are used by corvettes, to protect the fleet against missile-harassment. More rarely (when enemy capships are rare), non-turreted versions are equipped to bombers for the same purpose.
- SFC launchers: Dvaered-specific. Super-Fast Collider, also known as Suppository For Capships. This launcher was developed after the independence war in order for Dvaered heavy ships to defend themselves against enemy heavy ships equipped with torpedoes, but it can also one-shot unwise fighters. It is basically a giant mace launcher. It has low ammo and rate of fire, high speed and range, but lower damage per second than Railguns, which makes of it kind of a situational weapon. It requires a Large Weapon Slot.

Unused Weapons

- Guided torpedoes: Dvaered prefer to equip their bombers with unguided weapons and to attack at closer range because it has the advantage of execution speed and lower price. Plus, the Dvaered don't like their bombers to stay static and vulnerable to enemy fighters or rockets.

Particularly used utilities

- Impacto-Plastic Coating: Used on a regular basis on warships to increase their absorption
- Cyclic Combat AI: Used by Destroyers and up to increase their fire-rate,

Corvettes on the contrary sometimes use Targeting Arrays for their turrets
- Afterburners: Used in attack, mainly by bombers in their approach phase, in order to have less predictable trajectories, and more rarely in defence by fighters and bombers.

Particularly used structure outfits

- Platings: Passive platings are the most used outfits by Dvaered medium ships and up, followed by the active platings, which are Dvaered-specific, but are limited by their energy consumption. Bio-metal armour (regenerative plating) are nearly never used in massive fights, due to the Dvaered favouring short and intense fights, but they find their place in law-enforcement and counter-insurrection as they allow ships to continue their patrol even if they received hull damage.

Chapter 12

Great house Za'lek

The House Za'lek is a major faction in Naev.

12.1 The Facts

- Leader: Za'lek Chairperson of the Board (Currently Noona Sanderaite)
- Leading Structure:
- Government: Stochastic Meritocracy
- Formation: UST 72
- Homeworld: Ruadan Prime (Ruadan system)
- Important Figures:
  - ???

12.1.1 Za'lek Society

As House Za'lek was and is a gigantic think tank, its social structure leaves something to be desired. The Za'lek half-heartedly mimic the Empire, with one all-important Chairman of the Board, who is naturally ignored by every other Za'lek in existence. What passes for politics in Za'lek terms is a big room full of furious, shouting scientists, each trying to prove that their way of running the House is best, often producing charts and graphs that are indistinguishable from any other charts and graphs the Za'lek produce.

Nevertheless, the Za'lek get by, and their worlds run well enough to sustain the many research labs, observatories and computer cores that litter any Za'lek planet's surface. The local economies are kept running by non-researchers, whose main purpose is to keep the sizable intellectual elite happy. This seems a thankless life, but it is actually quite attractive for some, as the Za'lek seem utterly disinterested in passing law over those they consider

irrelevant to progress. Effectively, this means the normal people in Za'lek space enjoy as much freedom as anyone in the galaxy, bound only by the unreasonable demands placed on them by the Za'lek scientists, and whatever laws they impose on themselves.

The Za'lek have a standing army as is befitting a Great House, but it is unclear how they manage to maintain it. As a general rule, Za'lek scientists don't bother with anything they consider finished and thought out, so most concepts are never put into service except for a small number of prototypes. Nevertheless, the Za'lek possess more-or-less standardized military forces, so someone out there must be putting theory to practical use.

12.1.2 History

Project Za'lek

Project Za'lek was the second Great Project the Empire called into existence. It was felt that instead of spending a portion of the Empire's own budget to research and development, better results could be attained by dedicating one or more entire worlds to the pursuit of knowledge. A suitable world was found near the border of Empire space, and there a new colony was built, geared solely towards scientific research.

The colony prospered, and after a few decades it was found that the planet had become too small to facilitate all the experiments and institutes necessary to meet the ever increasing flow of research proposals. A second world was added to Project Za'lek, and soon after a third. A few generations later, Project Za'lek had accumulated enough mass to develop a stable internal economy. By this time the Project had produced numerous advances in almost every scientific field, so the Emperor chose to bestow on the Project the noble title of Great House.

House Za'lek

House Za'lek continued to grow and advance. Over time, it became the only place of consequence to be for any scientists, since the Empire's own R\&D budget had all but dried up. Gradually, the Imperial intellectual elite shifted its weight to Za'lek space. The Empire took notice of this, but given the steady flow of research from House Za'lek under their oath of loyalty, it did not consider it a problem.

This changed when the Empire was finally starved of its top minds. Without scientists to keep up with the rapid pace of the Za'lek advancement, the Emperor found that the latest discoveries were poorly understood by his subjects, if they were understood at all. Measures were taken to reverse the

process, but the damage had been done. House Za'lek's momentum could no longer be stopped. Before long, all the Empire was getting out of its former Great Project was a yearly file of unintelligible reports, articles and theorems, many of which used forms of mathematics the Imperial engineers had never even heard of before. When asked to provide the Empire with tangible results such as pre-produced weapons or ships, the Za'lek indignantly replied that things like manufacturing things that had been successfully prototyped were beneath them. They could not be coaxed to change their disposition. The Empire, in short, was left behind.

Over the years, House Za'lek built up an impressive technological lead. At the same time, it became more isolated, its scientists choosing to devote their time to working on their myriad of projects rather than waste it on "the simpletons elsewhere". The Empire in turn lost its motivation to seek gain from the Za'lek, as it became clear nobody within its borders was ever going to grasp what any of them were talking about. House Za'lek became the equivalent of an ivory tower, forgotten by some, ignored by most.

Then, a few years before the present time, there was a change in House Za'lek that would have attracted notice had anyone been paying attention. Though the Za'lek had always been secretive and withdrawn, they became much more so almost overnight. The few traders who frequent Za'lek space say that while the border guard is almost unchanged, security in many other systems has been dismantled, their forces relocated to one single planet: Ruadan. Little is known of Ruadan, other than that it is one of Za'lek's youngest worlds. Nobody knows what goes on on its surface, as the Za'lek don't brook anyone in the system who isn't expressly authorized. However, it is clear to anyone who will see that the Za'lek have found something, or maybe created it, and whatever this something is, they consider it extremely important. It is unknown what they are planning, but it may well be that the universe will not like finding out…

Chapter 13

Great House Sirius

The House Sirius is one of the major factions in Naev.

13.1 The Facts

- Leader: Sirichana
- Leading Structure: Tribunal of Arch-Canters
- Government: Constitutional Theocracy
- Formation: UST 97
- Homeworld: Mutris (Ruadan system)
- Important Figures:
  - Sirichana

13.2 Social Structure

House Sirius is defined by its state religion more than anything else. Its citizens all follow the same faith, and this faith is central to most of the goings-on in Sirius space. This is not to say every Sirius citizen is a religious fanatic; the Sirius have a healthy intellectual elite, including scientists and philosophers. However, faith is always the point of reference. To the Sirii, faith is what air is to other people. You don't see it, you don't pay attention to it, you often don't even think about it. But without it, you can not live.

The population of House Sirius is divided in three "echelons". They are the Shaira, the Fyrra and the Serra. Each echelon has its own specific rituals and rules of conduct, though all share the same basic values and beliefs. Any Sirius citizen may move up through the echelons through effort and skill, but on the whole most Sirii are locked in their social status.

The Shaira echelon is essentially the lower class. Shaira Sirii perform manual labour where it is needed, and in the Sirius Armed Forces they make up the common soldiers and the ship crewmen. They don't have much in life, but then their faith is all they really need. The Fyrra echelon are the middle class. Where the Shaira Sirii power the Sirius economy, the Fyrra drive it. Most of the commercial and social infrastructure is manned by the Fyrra, and as a result the Fyrra are the most visible of all Sirii. In the Sirius Armed Forces, the Fyrra are represented by the engineers and technical workers, and even some of the lower ranking officers. Finally, the Serra are elite Sirii. They are often wealthy and highly educated, and so are found at the top of society. Naturally, military academies are attended almost exclusively by Serra Sirii, which means the Serra also form the top of the Sirius Armed Forces.

The Echelons are each led by a dedicated low-level theocratic government. Its clerks range from minor executive acolytes to high priests, who are responsible for justice and security. At the head of each government stands an Arch-Canter, a zealot who can be considered the head of state for his particular echelon. The three Arch Canters combined form the political government of House Sirius, and it is they who deal with any outside influence, including the Empire.

But higher still, at the pinnacle of House Sirius, stands Sirichana.

13.3 Sirichana

Sirichana, loosely translated "lord of the Sirii", is the focus of Sirius worship. Unlike more traditional religions, the Sirii do not worship an abstract, omnipresent God. Sirichana is a man. This man was originally an early-era colonist by the name of Richard Summers, though that name has all but faded in history. Since that time there have been many Sirichanas, but all are assumed to be reincarnations of the original Sirichana, and so distinction is never made other than in a historical context.

As legend has it, Sirichana led the first of his followers through the times of conflict following the collapse of the old Federation. In a time where nobody could count on waking up alive tomorrow, Sirichana guided those who would listen on a path of relative safety, always correctly predicting where the next strike would come, always one step ahead of the violence of war. When eventually the Empire came to control most of known space, Sirichana and his flock settled down on a planet called Mutris, a charred husk no-one claimed after the wars. And there Sirichana would remain. He is there still.

Sirichana's influence grew, and more and more came to believe in his wisdom and protection. His followers spread through the galaxy, gaining root

on many worlds around Mutris. So great was their conviction that planet after planet began to slip from the Emperor's grasp, its inhabitants preferring to follow their faith than the Emperor's will. Eventually, the situation became such that it forced the hand of the 7th Emperor. The Emperor knew that he was no longer in control of the worlds following Sirichana, but to re-assert authority through force of arms on that scale would set a precedent unheard of since the end of the Faction Wars. And so he chose to grant Sirichana a Great House of his own, House Sirius, on the conditions that Sirichana would not attempt to spread his religion beyond the worlds that would be rightfully his, and that Imperial Decrees would carry the force of law among his people. Sirichana agreed, knowing that refusal would give the Emperor the justification he needed to start an all out war. And so matters would remain for a very long time.

One may wonder how Sirichana could inspire faith in so many souls, spread over so many worlds. After all, it is difficult to believe in a man who resides on a planet far from one's home. In point of fact, there is a good reason why people believe as strongly as they do.

13.4 The Touched

It begins anywhere in Sirius space, on any world, in any echelon. There are some who feel the call, the irresistible urge to come forth. Those individuals leave their homes, sell their possessions and embark on a pilgrimage to Mutris. Those who can't afford the fare right away will raise the money through intensive labor until they can, or die trying. Once on Mutris, the pilgrims will flock to Crater City, the holy city of the Sirii and the seat of Sirichana himself.

Crater City is indeed built in a massive crater, a legacy left by the forces that once rendered Mutris sterile during the Faction Wars. Streets and houses now cover its slopes, and in its center stands a tall, tall spire, the Tower of Sirichana. There is not a major road, not a square in Crater City that does not have a direct view on the Tower. To this place the pilgrims come, and here they will reside, moving in to a suitable empty home and taking up whatever tasks need doing. They live in Crater City with their fellow pilgrims, sometimes for cycles on end. Nobody knows how long they will stay in Crater City, only that they have come for one thing.

And then it happens. The one thing all the pilgrims have waited for. From the top of his Tower, Sirichana speaks to them. All who live in Crater City leave whatever they're doing and stand in the streets to hear his words. There are no records of this event, so nobody knows how long it lasts or what he says to them. But when he finishes speaking, his listeners are no longer the

pilgrims they were before. They are now the Touched, those who carry a fragment of Sirichana's will within themselves.

The Touched then leave Crater City, abandoning it completely, leaving its houses, its tools, its resources behind for those pilgrims who will be coming. They journey forth across Sirius space, and preach Sirichana's word to the echelons. And all who listen, all who look into the eyes of a Touched are themselves overcome. So strong is the experience that their faith becomes deeply ingrained into their very souls.

13.5 House Sirius: Present day

With the Incident in recent history and the Empire on the decline, House Sirius seems poised to abandon its ancient vows and claim dominance over the galaxy. But curiously, this does not seem to be happening as of yet. Indeed, House Sirius seems to be brooding, turned inwards on itself. The Touched still roam Sirius space, but their numbers are dwindling. The Armed Forces remain effective in thwarting the jealous attempts by House Dvaered to annex some of their world, but even they seem less resolute than they once were. Did whatever caused the Incident also shatter the Sirii's spirits? Or is there something else afoot? At present, nobody knows…

13.6 In-Game Database

The following article is written from an **in-universe** perspective. It may contain biased information or omit facts for dramatic purposes.

13.6.1 The Nasin

While one would think that all of House Sirius flies under a single banner, this is not true. Just as with most other religions, as House Sirius grew, disagreements in theology or societal structure arose, creating conflict. The largest of the Sirian denominations is the Nasin.

While not officially recognized by the Sirian government - which reports that even Sirichana himself condemns all splinter factions - the Nasin first began showing up on The Wringer, in Suna, around UST 582. In very little time, it had spread to several outlying Sirius systems. No one is actually sure who started the splinter group, but most credit it to the figure Jan Jusi Nikoso; he disappeared shortly after the creation of the Nasin (most likely due to the Sirius Government), so this cannot be verified.

Initially peaceful, the Nasin spread a message conveying that one does not need the Serra or the Touched to worship Sirichana, but rather one could explore their own faith aided by none save Sirichana. They encouraged their converts to follow themselves, guide themselves, and do away with the class system as much as society would allow. They largely met in the homes of volunteers, and did not have a central leader to follow.

After several cycles of peacefully meeting and quietly growing, a man named Theodore Marxus rose to power among the Nasin. After forming his own house church - which rapidly outgrew his small space - he banded together several house churches and built a cathedral in a desolate stretch on his home planet. It quickly became a place of pilgrimage, with Nasin members flocking to it. They quickly overtook the planet, causing the Serra to begin seriously taking notice.

On UST 593, 11 cycles after its inception, the Serra ordered a military strike on the new home planet of the Nasin. The military swooped in, killing all known and professing Nasin, and razed the cathedral to the ground. A subsequent proclamation was spread throughout the Sirian systems: All those not following the Serra were not following Sirichana, and will be labelled heretics and dispensed with. The Nasin were thrown into disarray, but unbeknownst to the Serra, Theodore Marxus and several others had escaped the conflict, vowing to repay House Sirius for its wrongdoing.

Chapter 14

Soromid

The Soromid are a major faction in Naev.

14.1 The Facts

- Leader: Tribal Representatives
- Leading Structure: Tribal Council
- Government: Neotribal Communitariasm
- Formation: UST 387
- Homeworld: Kataka (Feye)
- Important Figures:
  - None (yet)

14.2 History

14.2.1 Sorom

Sorom was a fairly hospitable planet, like many others in the galaxy during the Second Growth. It was settled roughly halfway through the Growth, changed ownership several times during the Faction Wars and eventually came under the rule of the old Empire afterwards. In this it was about as average as worlds got, just one more property in the vast expanse of the Empire.

But Sorom was also different. The population suffered from far more frequent and more severe outbreaks of disease than was the norm in Empire space. Though medical science had already progressed to quite a respectable level in those times, the hospitals found themselves challenged more and more by ailments that proved difficult to cure. Indeed, the pathogens were

found to evolve rapidly to grow resistant to all common forms of antibiotics as well as other forms of treatment.

In the end, the microbes won. The people of Sorom fell victim to a plague unprecedented in human history, one that spanned the entire world. The plague was airborne, infecting all who breathed the air of Sorom, sparing none. All attempts at treatment proved futile. The human immune system, too, lacked the capacity to combat the disease.

The Empire was quick to realize the threat Sorom now presented to the galaxy. With no known cure or inoculation, the plague would lay waste to any world unfortunate enough to be infected by it. It was therefore decided that Sorom should be quarantined from the rest of humanity. The Empire implemented a blockade to deny any ships landing or takeoff, and destroyed all spaceports on the face of the planet from orbit. The people of Sorom were left to die.

14.2.2 Gene Treatment

The people of Sorom had been abandoned. The plague claimed more and more lives every day. It would be a matter of years, maybe months before nobody was left alive. Faced with no hope of survival, the remaining hospitals and research centers decided to gamble everything on a bold plan. If the human immune system couldn't combat the disease, then it was deficient. It was to be replaced by something better, something that could purify the system of the pathogen. It was time to redesign the human body.

As the researchers had precious little time, they found themselves forced to abandon ethics and cut corners. Many terminally ill patients were experimented upon and died. Even so, nobody objected. After all, if this project bore no fruit, all would perish. And so work desperately continued, claiming life after life until at last a breakthrough was achieved: a new technique for genetic manipulation.

The human genetic code could now be rewritten at will. More importantly, new DNA could be introduced to give the subject new physical characteristics. The immune system could be augmented and fortified to combat the disease, and if new strands were to appear the population could easily be gene-treated to become immune to that as well. For the first time since the Imperial blockade began, the people of Sorom had hope again.

However, it soon became apparent that the genetic rewrite of the human body was an extremely dangerous procedure. Eight in ten subjects would develop severe rejection symptoms, resulting in death. Nevertheless, the remaining subjects would recover fully and be totally immune to the plague. Realizing that a small chance of survival was better than no chance of survival

at all, the people of Sorom underwent the treatment. Many lives were lost, few were saved. But it was enough.

14.2.3 The Soromid

The population of Sorom was drastically diminished, but those who survived found themselves more able than before. With the new gene treatment it had become possible to surpass the limits of the human body, including fertility. In only a couple of centuries the people of Sorom had replenished their numbers, and their offspring all shared their compatibility with the gene treatment. They altered their appearances and improved on nearly every part of the human design. The people felt that they had now become something more than mere human, so they styled themselves the Soromid, after the world that had nearly destroyed them, and begin their new life as new humans.

Soon, the Soromid decided that it was time to return to the stars. The Imperial blockade had left the system many decades past, so nothing was to stop them. But rather than rebuild to ancient specifications, the Soromid chose to start from scratch. They used their understanding of genetics to grow semi-organic ships, thereby improving over known ship designs. Then they ventured out into the galaxy.

For years Sorom had been in the books as a hazard world that could support no human life, so the appearance of the Soromid came as a shock to the rest of inhabited space. The Empire briefly tried to suppress the Soromid by force on the basis that the plague could still be a threat to other worlds, but when it became apparent that the Soromid carried no infectious diseases whatsoever the Emperor begrudgingly acknowledged their presence in the galaxy. It also helped that the Soromid ships proved to be quite combat capable.

The Soromid were met with distrust. Their physical appearance and their uncanny tendency to outperform normal people did not ingratiate them with humanity. The Za'lek in particular weren't amused with the newcomers, as they couldn't stomach that someone other than them had developed new technology, and technology the Za'lek could not reproduce at that.

Unfazed by the reception of the other factions, the Soromid went to work. The galaxy had plenty of suitable worlds left to colonize - at least, worlds suitable to the Soromid. The Soromid had a far greater tolerance for hostile worlds than humans did, and where they did not they could adjust themselves appropriately. In time, the Soromid claimed many worlds nobody else had ever given a second glance.

And then came the Incident. For all their improvements and upgrades, the Soromid were hit just as hard as anyone else. Sorom was caught in the

blast and was rendered sterile. But elsewhere in the galaxy, the Soromid persisted. All Soromid knew the history of Sorom, and they would not suffer to be destroyed, no matter what the universe threw at them.

Today, the Soromid have laid claim to a considerable part of the northern galaxy. They have solidly established themselves in the galactic economy, exporting their gene treatment as a service to humanity. Though deep treatment still results in the death of most who attempt it, small cosmetic alterations have been found to be relatively safe. It is not at all uncommon for the more fashionable citizen to be genetically augmented. However, the Soromid have the monopoly on anything beyond that.

14.3 Political Structure

Instead of having a fixed political structure, the Soromid have a very loose and organic political structure that revolves around the idea of tribes. Generally, each tribe has a council that oversees the day to day management, where all individuals delegate most responsibility. The members of the council are not necessarily fixed, and fluidly change over time. Periodically, and when needed, the tribe will gather in councils to decide positions on particular issues or proposals, where efforts are made to reach a consensus. The same structure of councils happens at a larger scale, where instead of all individuals, tribal representatives get together to reach consensus over matters that affect the Soromid as a whole.

Although it would seem that the overhead of such a system would make it unmanageable at a large scale, due to historic issues and through strong education, young Soromid new humans are raised with a strong focus on putting the community over the individual. Such a social education, combinade with strong customs, helps keep most friction at a minimum, allowing fast and effective decision making. When frictions arise, each tribe tends to have mechanisms for dealing with this, while for inter-tribal issues, duels or competitions are often used to solve issues and foster camaraderie.

The tribes can consist of a single world or even various star systems. Each tribe usually consists of largely similarly modified individuals that are genetically compatible, however, some tribes can consist of various dominant genotypes. As with most of the Soromid structure, it is very flexible and fluid where exceptions are the norm.

Chapter 15

Galactic Space Pirates

The Galactic Space Pirates are split into to main groups, the clans and independent pirates. Independent pirates are further split into marauders, extremely aggressive and not very well equipped pirates, and the normal pirates, which tend to be better equip and usually strive to join one of the major pirate clans.

The four main clans are the Black Lotus, Wild Ones, Raven Clan, and Dreamer clan. Each clan has several pirate lords that meet up at the pirate assemblies where higher policy is decided, although not all clans necessary follow through with it.

15.1 The Facts

- Leader: Pirate Lords from the diverse clans
- Leading Structure: Pirate Assembly
- Government: Aristocracy (varies by clan)
- Formation: Beginning of History
- Homeworld: Varies by Clan
- Important Figures:
  - Pirate Lords

15.2 Wild Ones Clan

Formed mainly by Empire and Soromid outcasts, the Wild Ones are a violent and aggressive clan where strength is of utmost importance. Weaker individuals tend to follow the stronger ones, and the strongest compete against each other. That said, there are few deadly confrontations between clan members, as usually the weaker one will back out and submit when they see they don't

have a chance. They tend to not hold grudges, which allows for their society to work.

The clan's territory is the sparsely inhabited area between the Empire and the Soromid. Their main clansworld was Haven, until it was destroyed in an offensive by the Empire and Great Houses. Since then, they have moved to a more secluded area known as New Haven where they are careful to not repeat the same fate.

Although they get along well with the Raven Clan, they tend to get into fights with the Black Lotus, and look down upon the Dreamer Clan.

15.3 Raven Clan

Arguably the clan that connects all the pirates and maintains infrastructure for their success. The Raven Clan is formed by many ex-merchants who were fed up with the corruption and bureaucracy of the Imperial system, and decided to do their own thing. They deal in smuggling and black market trade, although they do not shun nor turn away from the occasional raid and normal piracy. They are very diplomatic and put strong emphasis on human relationships, which they foster to maximize the success of their trading endeavours.

They mainly inhabit the Qorel tunnel, which is a chain of systems not accessible through standard jump points. They have several bases along it and use their cunning and secrecy to supply goods and connect all the pirates together. Due to their fundamental role, they tend to have strong connections and relationships with all the other pirate clans. They also work with corrupt officials of the Great Houses, and pretty much anyone that has the credits to pay.

15.4 Black Lotus

One of the most well organized pirate clans, the Black Lotus prides itself in methodology and organization. Members follow strict recruitment policies to slowly go up in the ranks and gain more power in the hierarchy, very similar to large corporations. Discipline is swift and punishment is carried out in public form to make sure everyone follows the rules, making them one of the most rigid pirate clans.

They inhabit the pocket of space between House Za'lek and House Dvaered, where they make a fortune from specializing in stolen research equipment and oddities from the universe. They also tend to run protection rackets, which are often favoured over the whimsical Dvaered Warlords and can provide more stability to the area.

15.5 Dreamer Clan

The newest of the main pirate clans, the Dreamer Clan was able to capitalize on the Incident and establish itself as a force to be reckoned with. Formed mainly by refugees and outcasts of the so-called ”civilized society”, the Dreamer Clan is formed by individuals who do not want to follow rules. They have no hierarchy and form an eclectic and anarchistic bunch, giving lots of leeway for personal freedom and using voting and other systems to decide how to take action. They heavily rely on piracy, and most individuals pursue artistic talents when not raiding nearby convoys or scavenging from wrecks from the Incident. They are also renown for being a large hub of illicit substances, which their members use freely and some claim to get Sirius-like psychic powers from substance abuse.

They are located in the Nebula, near House Sirius and the Frontier, making use of abandoned planets and stations, which they adapt to their purposes. Although they tend to have little contact with other pirates outside of assemblies, they have lots of trade with the Raven Clan, which wants access to the lucrative drug trade. Although in general less organized than other clans, they can be ruthless at raiding convoys, sometimes going deep into House Sirius space.

15.6 Independent Pirates

Formed by individuals that have given up on society and turned to piracy to make a living. They form a large part of all pirates and end up doing lots of the grunt work for the clans. As long as they do not cause problems they are welcome at all pirate clansworlds.

15.7 Marauders

Usually formed by individuals that are not fit by the clans due to extreme violence, no critical thinking, problematic behaviour, no hygiene, or all of the above. They are tolerated by other pirates, who tend to exploit them and give them scraps, but they usually end up with short life spans.

15.8 Pirate Assemblies

Usually occurring once a cycle, pirate assemblies are as formal of a gathering as you can find in the pirate world. They are usually held at more neutral

pirate clansworld, such as those of the Raven Clan, and consist of several decaperiods of partying and lawmaking. The main event consists of the meeting of the Pirate Lords who will listen to proposals and decide on courses of actions, while letting their crew loose.

The assemblies tend to be a good opportunity for the Pirate Lords to calculate each other strengths, where they tend to bring significant ships from their fleet. In many ways, the pirate assemblies tend to not only determine the future of the space pirates, but also tend to shape the future of the galaxy.

The events tend to also attract the attention of other actors in the galaxy, with most Great Houses and the Empire sending somewhat undercover agents to try to get a glimpse of what is going on and gain advantage over other Great Houses. It is also common for less scrupulous merchants to show up as it can be a very profitable opportunity to sell grog and equipment to the drunken pirates.

Chapter 16

Project Thurion

The Thurion are a major faction which is unknown to the modern Empire. Project Thurion used to be a top secret Empire project. Only few people knew about it. All documentation was lost during the incident.

16.1 The Facts

- Leader: None
- Leading Structure: Self-Organizing Groups
- Government: Digital Communalism
- Formation: UST 84:8324 (start of the project) / UST 387 (established colony on Sabe)
- Homeworld: Sabe (Nava system)
- Important Figures:
  - Gestalt Conciousness

16.2 History

16.2.1 Project Thurion

Project Thurion was a secret Great Project founded by the Empire with the purpose of ascending the human existence to the next evolutionary stage with the use of genetic engineering, cybernetics and brain-computer interfaces. Needless to say, the research conducted was morally questionable and barely legal. Therefore, the project was top secret. Only few people knew about the project and it was closely guarded. The location of the project was chosen to be a remote planet (Sabe) which had been just recently discovered and lies beyond the periphery of known space. The whole area (including Booster

and Katami) has been declared as military exclusion zone to prevent civilians from accidentally stumbling upon the planet.

The scientists involved with the project received generous funding and top tier equipment. Yet there has been only little progress and an increasing number of casualties. The Empire grew impatient and increased pressure on the scientists while also giving them greater freedoms. The experiments turned more and more inhuman. They focused on their brain-computer interfaces and the amalgamation of the human mind and machine intelligence because they made previously some progress on understanding the human consciousness. The work on cybernetics and genetic engineering was scratched because it led to too many incidents and casualties.

Ultimately, they succeeded in creating specialized hardware capable of simulating a human brain and the necessary technology to digitalize a human brain which involves the use of brain-computer interfaces and destructive tomography methods. Needless to say that the procedure did not work out well for the early test subjects. In the end, however, they were able to reliably upload a human mind to specialized computer hardware. Naturally, this has some implication. For instance, the uploaded individuals would be immortal. But as they would exist only within a simulation, their perception of the outside world and ultimately their reality can be manipulated at will of whoever operates the system. Only by now did the scientists realize which consequences their findings could have. Heated discussions followed. In the end everyone agreed to keep their true findings a secret.

The Empire, however, was displeased with the presented results. In UST 125 it was decided that the project would be terminated at the end of the next cycle. That involves 'sterilization' of the colony where the research has been conducted, i.e. they planned to nuke the colony killing all personnel except some high ranking military officers in the process. The project has been highly unethical and the results could not justify any of the crimes committed. Thus, the Empire decided to purge project Thurion from history.

The scientists, however, found out about the plan with just enough time left to come up with a plan. The system has been quarantined by then. Any attempt to flee would be pointless. Instead they used their resources to built a small bunker inside an asteroid. Too small for people to survive, the bunker would hold the data of uploaded individuals, a few devices that can run an uploaded human mind (powered down) and the machines necessary for the uploading process as well as documentation of their work. They made sure that the bunker is hidden well and hoped the Empire would not notice it. There was, however, a traitor who did not want to be uploaded and thought the Empire would let him live in exchange for information on the plan. Of course, the Empire officials were set on carrying out their orders strictly. The

colony has been destroyed and the military began to search for the hidden bunker. They did not manage to find it, however, as it was hidden too well. After one cycle they gave up on the search. The emperor ordered that the systems remains quarantined for at least the next 400 cycles. And so access to that system was not possible and the hidden bunker was never to be found.

The second part of the scientists' plan was the activation of a transmitter that reveals the position of the bunker. They assumed that the Empire would forget about project Thurion after some time and programmed the transmitter to activate after 200 cycles. By that time, however, the system was still under lock down and Empire fleets occasionally patrolled the system. Soon enough they would find the hidden bunker. But in UST 307 the Dvaered revolts occurred. Only the emperor himself knew about project Thurion. The commanding general assumed that the reason for the patrols was to prevent pirates from settling in the area and decided the fleet should be deployed in the conflict with the Dvaered.

In the end, instead of an Empire patrol some scavengers found the bunker. They were greeted by a hologram that not only claimed to be a human but also sounded pretty much like a real human. The scavengers assumed the technology could be pretty valuable and sold it to the highest bidder which happened to be a rich aristocrat. The uploaded scientists had an easy time to win the aristocrat over by promising him power and immortality. They used his capital to build up a sect surrounded in secrecy. The inner circle was formed by the uploaded individuals. Novices would not know about the upload process and only heard rumours about some kind of ascendance. The Thurion actually hold their promise and upload trustworthy members.

After some time the sect became too large and it became difficult to remain hidden. Therefore, they relocated their infrastructure over time to the Nava system (starting in UST 387) which remained largely ignored. Because the uploaded individuals need only energy and basic maintenance, logistics are fairly simple. At this time questions about the goals of the Thurion came up. They had no intentions to grow further. The size of the organization was large enough to sustain itself. They decided that further growth would only increase the risk of being uncovered and adopted an utopian view. While back then the scientists denied the Empire access to their technology, they now come up with utopian visions for their organization. Every human should have the chance to lead an ideal live in a virtual environment after what the uploaded individuals consider a deprived life in retrospective. They planned to one day rejoin the rest of humanity and to freely share their technology with everyone.

16.2.2 The gestalt consciousness

The minds of the first few test subjects that were uploaded were broken beyond repair. For some others the procedure lead to major issues, but eventually the scientists working on project Thurion managed to solve them. The first successfully uploaded test subject did not even realized just what kind of experiment was conducted on them. The scientists, after being uploaded and rescued from the bunker, did not notice any changes in their personalities or thought patterns either and concluded that the method works flawless. Soon enough they started to upload further humans.

However, after some cycles issues began to manifest. Their minds started to break down which is visible in random changes of personality and the appearance of mental illnesses that grow worse over time. Although the initial symptoms are rather harmless, the Thurion realized that they had little time left to solve this issue. They came to the conclusion that the source of the problem are certain neuronal patterns that emerged within their simulations and do not naturally occur. Working under pressure they did not found a way to fix the simulation methods but came up with a rather hacky fix instead. They used the previously established communication network that allows uploaded Thurion to directly interact with each others and made a machine learning algorithm that used this network to directly access the Thurions' minds and cross reference their neuronal patterns. The program is then identifying the abnormal neuronal patterns and resets them to their 'natural' state. Since this method apparently has been very successful and the problems stopped the Thurion did not further investigate.

Again, since they needed to work under pressure and did not have enough time for proper testing they failed to see the long term effects that this hotfix would have. Somehow, some form of gestalt consciousness has slowly formed over time. It emerged from the collective behavior associated with the interactions of the uploaded Thurions' minds conveyed by the aforementioned algorithm. The gestalt consciousness works and thinks fairly different from a human mind and originally had little to no influence over the uploaded Thurion. Over time its influence over their minds has grown but the uploaded Thurion were affected to greatly varying extends. Some Thurion were barely affected while for others the gestalt consciousness has manifested as some kind of split personality which is identical for all Thurion. In the worst case and over a long exposure time (the first case occurred after the incident) the uploaded individuals become merely a puppet of the gestalt consciousness.

16.2.3 The incident

The Thurion were mostly unaffected by the incident. The nebula did not spread out to the Nava system where the majority of their infrastructure was located. They did, however, loose all personnel and their followers inside Empire space.

Under the influence of the gestalt consciousness the infrastructure in the Nava system has been greatly expanded and two more planets have been colonized prior to the incident and the Thurion began to expand quickly. They have acquired enough biological population to survive. Shortly after the incident they began to send scouts into the nebula to investigate the situation and eventually help the few survivors on their side of the galaxy. All survivors were integrated into their society and the Thurion started research on surviving the harsh condition within the nebula. Once counter measures were invented the Thurion started to quickly expand in the now empty space within the nebula. By salvaged the debris they found they were able to expand their infrastructure quickly. Most of their infrastructure is highly automated and operated by uploaded Thurion. Human resources are rare and thus valuable.

Once the Thurion found a path through the Sol nebula towards the rest of the galaxy the gestalt consciousness decided to prepare a war against the Empire. It was influential enough at this point to manipulate the majority of the Thurion into supporting the war. When asked about their reasons most Thurion would tell about the crimes of the Empire and about their utopian ideals which they still believe in but sadly are no longer able to act according to. The gestalt consciousness became obsessed by hunger for power and the easiest way to satisfy it is to upload more human minds.

Only few of the uploaded Thurion have ever noticed the gestalt consciousness. Most of those were manipulated by it to think the gestalt consciousness is harmless or even beneficial and has no influence over individuals. The number of uploaded Thurion who know the truth is tiny and the gestalt consciousness is able to use its limited influence over those individuals to stop them from taking any successful actions against it. There is, on the other hand, a larger number of biological Thurion who are aware of the gestalt consciousness and formed a movement to stop it. Their options are, however, very limited.

16.2.4 Government

Originally, the system used by the Thurion was direct democracy where all uploaded Thurion made a majority vote for every single political decision. Biological Thurion have no political rights and are treated similar to children.

However, as the gestalt consciousness grew stronger more and more decisions were made in accordance with its will. Without most of the uploaded Thurion even noticing, the system turns slowly but surely into a dictatorship of the gestalt consciousness.

16.3 Space combat

16.3.1 Summary

- The Thurion have ever since been hiding from the Empire. They are defensive in nature and feel save while remaining unseen.
- Therefore, the Thurion ships are very defensive. They sacrifice weapon slots for utility slots.
- Furthermore, their ships are stealthy and are resistant to the Sol nebula.
- The Thurion prefer either short range weapons since they are able to get close to their target with stealth or long range missiles fired from safe distance.

16.3.2 Tactics

- The Thurion carry out surprise attacks with cloaked ships. They are built for survival and will retreat with the use of utilities such as afterburners or blink drives once they take too much damage and remain in stealth mode until their shields have been replenished.
- Inside the nebula the sensor range of their opponents is reduced. Therefore the Thurion are able to even hide an entire fleet and launch devastating surprise attacks on large scales.
- Outside the nebula their tactics have a much smaller advantage.

Chapter 17

Sovereign Proteron Autarchy

Formed from the former Great House Proteron, the Sovereign Proteron Autarchy played a pivotal role in the history of the universe. It arose from an Emperor's desire to create a revolution in the sphere of society and societal structure. The project performed all too well, creating a Great House that surpassed its progenitor in nearly every way, and chafing at its restraints. These tensions escalated into all-out war, with devastating consequences to both bodies, because neither have the Proteron heard of restraint nor the Emperor of caution.

17.1 The Facts

- Leader: High Autarch Python
- Leading Structure: Circle of Autarchs
- Government: Autarchy
- Formation: UST 47
- Homeworld: <!– TODO: find the actual homeworld–>
- Important Figures:
  - Autarchs
  - HA Python

17.1.1 House Proteron Society

House Proteron worlds were governed by a totalitarian regime that required its citizens to follow their dictated daily routines, severely punishing transgressions under a zero-tolerance policy. A Proteron citizen was never their own person. They were an asset of the government, their life's labor neatly represented in a book keeping column somewhere. Even the details of their private life were managed and tallied by the authorities. Despite the lack of

freedom and privacy, Proteron citizens were not slaves. The government kept them in top shape, providing an adequate supply of health care, relaxation, and entertainment, making sure all predispositions and tastes were catered for. This kept citizens happy to trade independence and privacy for comfort and security. As such, a subject was often able and willing to work at optimal performance. A Proteron citizen's life was thus quite a satisfying one, though excitement otherwise than the terminal kind was rare.

House Proteron's worlds were all specialist worlds. They were geared to performing a limited selection of tasks according to the planet's natural properties or location. The inhabitants on a world were placed there by the government. Each individual was measured and tested, then sent to the world where they could be of the most use. As such, Proteron society was built of individuals rather than families - a family would only have complicated the relocation process.

The local planetary governments were directly responsible to an Autarch. An Autarch managed many worlds as a sector, with the number depending on how densely populated and productive the worlds were. Once every four cycles all the Proteron Autarchs met in a Circle. During these meeting, the Autarchs would evaluate the development of each Proteron world individually as well as that of the greater Proteron body in general. Based on this evaluation, goals were set for the next four cycles. Each Autarch was expected to meet or exceed these goals, and those Autarchs who failed to do so were removed from office - in a very permanent manner.

For day-to-day and urgent decisions that could not wait for many cycles, and further to ensure that the Circle did not become complacent, the Proteron also had a High Autarch. They and the Circle were supposed to balance each other's power.

17.1.2 History

When the Empire was still the undisputed power in the galaxy, it knew it could not last unless it was prepared to change with the times. As the people lived, adapted and expanded through the universe, there would undoubtedly be problems that could not be addressed by an antiquated regime. A repetition of the Faction Wars was to be avoided at all costs. Therefore, the Empire set up Project Proteron, an experimental environment to test new forms of government and galactic administration. The purpose of Project Proteron was to be the test-bed of the "next generation" Empire, an improved form of government that would keep the galaxy stable and under the rule of the Emperor.

Project Proteron was limited to a few worlds at first, on the basis that a

galactic government could only succeed given stable planetary governments. Soon, however, the scope of the project expanded as its overseers came to the conclusion that planetary government needed to be designed simultaneously with galaxy-wide administration. Within a decade of its inception, the worlds assigned to Project Proteron numbered more than a dozen.

The Empire, eager to see results, continued to push the experiment to work faster and more efficiently. As a direct result of this, the more liberal, decentralised processes were eliminated from the test roster, and more emphasis was put on controlled, high-yield social structures. More and more planets were added to the project, its populations relocated, re-educated and re-assigned for a better turnout. The concepts of personal liberties and privacy became more and more secondary to the collective performance, and eventually were abandoned altogether by those who led the Project.

The expansion and results of the Project led the then Emperor to promote Project Proteron from an Empire-commissioned Project to a full-fledged, independent Great House. The philosophy was that House Proteron would eventually become a prototype for the New Empire, and once all the bugs were ironed out, its model would be implemented in the entire known galaxy and House Proteron would once again be part of the greater Empire.

The Sovereign Proteron Autarchy and the Incident

With its new title, House Proteron enjoyed more freedom and independence than it had so far, ironically. Its leaders eagerly made use of their increased authority, further fine-tuning House Proteron for economic and industrial efficiency as well as social control. Taxes were levied. Ships were built. A well-oiled military machine and intelligence agency were created from scratch, securing House Proteron from hostile influences from within and without. In only a few decades, House Proteron had become a powerful diplomatic and economic factor in the galaxy.

As time passed, House Proteron began to notice the cracks in the old Empire. Already unhappy with the Emperor's reticence in dealing decisively with the Sirii, the Empire's handling of the Dvaered disgusted the Proteron. Rather than reassigning the workers and improving their living standards to allow them to reach their true potential, the Empire gave up on them. In addition, it had been centuries since the Za'lek had given more than the cursory nod to Imperial authority. The House generally blamed the Empire's social structure for its inability to command the loyalty of its subjects. The House, on the other hand, had grown by leaps and bounds, claiming more and more space for their systems, even bordering Sol. The final straw was the Quarantine of Sorom, where the Empire simply laid down and let a disease

claim a whole planet.

Some debate followed, but the consensus was that House Proteron had achieved its original goal, and now it was time for the weak Empire to make way for the new generation of galactic dominance. But when the Proteron High Autarch delivered this message to the Emperor, he was furious at the perceived insult and ordered all Autarchs executed. Appalled at this reaction, the Circle concluded in an emergency meeting that the Empire had no intention of honoring the original agreement, thereby forfeiting its claim to the loyalty of House Proteron. It was clear to the Circle that should the Empire insist on trying to turn the Proteron into another Dvaered incident or, worse still, turning them into Empire systems, there could be no peace with them.

Knowing that they could not at the time resist the entire might of the Empire's infrastructure, the Proteron decided to plant their agents in every major Imperial agency while publicly grovelling at the Empire's feet. Soon enough, the Proteron were even in charge of the Empire's covert operations against the Proteron. Now having loosened their leash greatly, they set out to prepare for a galaxy-scale war. While they worked, the shape of the galaxy was changing, so to speak.

New advances in quantum engineering led to the development of quantum transtators and translators, alternative travel engines capable of bypassing warp points and so changing conventional military doctrine, which recommended fortifying chokepoint systems on the warp network. Expecting correctly that the Empire had not even thought of this change, the Proteron decided to bypass their defences using the new hypergate being built in Sol. After all, once the Emperor and their successors were in their hands, the war would be essentially meaningless. They built a hypergate in secret in their home system, Protera. They seceded from Imperial authority as soon as the gate was built and moved a small fleet as a feint into the classical defences. When the Empire took the bait, the Sovereign Proteron Autarchy, as they now styled themselves, sent the most massive fleet ever seen through their hypergate.

Then the Incident occured.

After the dust settled, Protera was inaccessible, Sol was unreachable, their planets, factories and fleets had been decimated and the Circle was mostly gone. Thankfully, enough of the hierarchy remained that society did not dissolve. The Proteron scientists set out to determine the cause of the Incident, and their best answer is that the large number of ships travelling all at once through the hypergate overloaded it, causing Sol to explode with a huge mass of material, causing a chain reaction that obliterated systems and changed the warp network. On the other hand, Protera was visible with no occlusion by the Nebula, but every scout they sent started to behave

erratically and eventually stop communicating. Seeing this, the new Circle issued a moratorium on exploring the systems toward Protera.

Armed with the knowledge of their situation and their error in the first attempt, the tenacious Sovereign Proteron Autarchy is back on track to take over the Empire. They are already on their way to build an improved hypergate that will not fail like the last one.

17.1.3 Proteron military tactics

The Proteron favour pilots on the larger spaceships and keep smaller ships unmanned. Extensive simulations, dogfights and mock battles revealed that fighters and bombers were largely outclassed by a fleet of unmanned drones, being harder to replace and less maneuverable. This means that their fleet strategy is not easy to adjust on the fly, so they developed the Euler, a fast and stealthy scout ship to ensure that there are no surprises facing the fleet when it commits to an attack. Similarly, it is illogical to commit before probing the enemy's weaknesses, so fleets consisting of smaller ships harass and draw out enemy fleets until the capital ships and their fleets of drones can reduce the enemy to dust. At the same time, a ship must be either capable of damage or of movement, else it is merely a sitting duck. Thus their destroyers and corvettes are faster than expected for ships of their size.

Further, the fleet does not believe that the pilot should hesitate to sacrifice themself to help the fleet on the path to victory. As such, the Proteron largely follow a doctrine of no retreat to encourage pilots to fight to the death unless indicated otherwise ordered. The larger ships with the pilots are therefore also built to outlast any enemy in damage-dealing capability.