Naev Development Manual

Version 0.10.0-alpha.1

Naev DevTeam

November 23, 2022

Contents

1 Introduction **5**
- 1.1 Getting Started . . . 5
- 1.2 Plugins . . . 6

2 Plugin Framework **7**
- 2.1 Directory Structure . . . 7
- 2.2 Plugin Meta-Data `plugin.xml` . . . 9
- 2.3 Plugin Repository . . . 10
- 2.4 Extending Naev Functionality . . . 10
  - 2.4.1 Adding News . . . 10
  - 2.4.2 Adding Bar NPCs . . . 10
  - 2.4.3 Adding Derelict Events . . . 11
  - 2.4.4 Adding Points of Interest . . . 11
  - 2.4.5 Adding Personalities . . . 11

3 Missions and Events **13**
- 3.1 Mission Guidelines . . . 14
- 3.2 Getting Started . . . 14
- 3.3 Basics . . . 17
  - 3.3.1 Headers . . . 17
  - 3.3.2 Memory Model . . . 21
  - 3.3.3 Mission Variables . . . 23
  - 3.3.4 Hooks . . . 23
  - 3.3.5 Translation Support . . . 25
  - 3.3.6 Formatting Text . . . 26
  - 3.3.7 Colouring Text . . . 27
  - 3.3.8 System Claiming . . . 28
  - 3.3.9 Mission Cargo . . . 29
  - 3.3.10 Ship Log . . . 31
  - 3.3.11 Visual Novel Framework . . . 31
- 3.4 Advanced Usage . . . 34
  - 3.4.1 Handling Aborting Missions . . . 34

3.4.2 Dynamic Factions . . . 34
3.4.3 Minigames . . . 35
3.4.4 Cutscenes . . . 35
3.4.5 Unidiff . . . 36
3.4.6 Equipping with `equipopt` . . . 36
3.4.7 Event-Mission Communication . . . 36
3.4.8 LuaTK API . . . 37
3.4.9 Love2D API . . . 37
3.5 Tips and Tricks . . . 38
3.5.1 Optimizing Loading . . . 38
3.5.2 Global Cache . . . 39
3.5.3 Finding Natural Pilots NEW . . . 40
3.5.4 Making Aggressive Enemies . . . 40
3.5.5 Working with Player Fleets . . . 40
3.6 Full Example . . . 41

4 Systems and System Objects 47
4.1 Systems . . . 47
4.1.1 Universe Editor . . . 47
4.1.2 System XML . . . 47
4.1.3 System Tags NEW . . . 47
4.1.4 Defining Jumps . . . 48
4.1.5 Asteroid Fields . . . 48
4.2 System Objects (Spobs) . . . 48
4.2.1 System Editor . . . 48
4.2.2 Spob XML . . . 48
4.2.3 Spob Tags NEW . . . 48
4.2.4 Lua Scripting . . . 48
4.2.5 Techs . . . 48

5 Outfits 49
5.1 Slots . . . 49
5.2 Ship Stats . . . 49
5.3 Outfit Types . . . 49
5.3.1 Modification Outfits . . . 49

6 Ships 51
6.1 Ship Classes . . . 51
6.2 Ship XML . . . 51

Chapter 1

Introduction

Welcome to the Naev development manual! This manual is meant to cover all aspects of Naev development. 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.

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.

1.1 Getting Started

This document assumes 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[2]. 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	TODO

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/tree/main/docs/manual>
[2] <https://github.com/naev/naev>

1.2 Plugins

Naev supports arbitrary plugins. These are implemented with a virtual filesystem based on PHYSFS[3]. 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:

```
missions/
  my_mission.xml
```

This will cause `my_mission.xml` to be loaded as an extra mission.

[3] <https://icculus.org/physfs/>

Chapter 2

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.

2.1 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.
- `damagetype/`: contains all the potential damage types.

[1] <https://icculus.org/physfs/>

- `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 equip-

ment 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.

2.2 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>
</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.

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 bas egame, 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 explicity 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.

2.3 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.

2.4 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.

2.4.1 Adding News

TODO

2.4.2 Adding Bar NPCs

TODO

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

2.4.3 Adding Derelict Events

TODO add engine support

2.4.4 Adding Points of Interest

TODO

2.4.5 Adding Personalities

TODO

Chapter 3

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 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>

3.1 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.

3.2 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 3.3.1.

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 3.3.5.
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

```
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 3.3.11). 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 3.3.6.
- 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 3.3.2.
- You have to pass function names as strings to the family of `hook.*` functions. More details on hooks in Section 3.3.4.

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 3.6 that covers more advanced functionality.

3.3 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.

3.3.1 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 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.

[6] <https://regexr.com/>

Finally, there is a `<notes>` section that contains optional meta data about the meta data. 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>
--]]
```

3.3.2 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 wit 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.

3.3.3 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 to obtain 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.

3.3.4 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:

```
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

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

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.

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.

3.3.5 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

[9] <https://naev.org/api/modules/hook.html#pilot>
[10] <https://hosted.weblate.org/projects/naev/naev/>

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 envelope 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.

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:

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

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

3.3.6 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:

[11] <https://www.gnu.org/software/gettext/>

```
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].

3.3.7 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"`.

3.3.8 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")}, nil, true ) then
      evt.finish(false)
   end

   -- ...
end
```

In this case, the second parameter is set to `nil`, which defaults to trying to claim the system instead of just testing it, and more importantly, the third 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.

3.3.9 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 functiosn 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 )
```

3.3.10 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.

3.3.11 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.

3.4 Advanced Usage

TODO

3.4.1 Handling Aborting Missions

TODO

3.4.2 Dynamic Factions

TODO

3.4.3 Minigames

TODO

3.4.4 Cutscenes

Cutscenes are a powerful 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 same 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. Further-

more, pilots can be controlled and made to do all sorts of actions. There is no limit to what is possible!

3.4.5 Unidiff

TODO

3.4.6 Equipping with `equipopt`

TODO

3.4.7 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 to define 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 to trigger 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.

3.4.8 LuaTK API

TODO

3.4.9 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. 3.3.11), minigames (Sec. 3.4.3), and LuaTK (Sec. 3.4.8) 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 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.

[17] <https://love2d.org/>

```
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.

3.5 Tips and Tricks

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

3.5.1 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.

3.5.2 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.

3.5.3 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.

3.5.4 Making Aggressive Enemies

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

3.5.5 Working with Player Fleets

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

3.6 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.
   In 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 {reward} when we get there. How's that
        sound?"]]), {pnt=misplanet, sys=missys, reward=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 {reward} for it! What
       do you say?"]]),
        {pnt=misplanet, sys=missys, reward=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 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 on {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 4

Systems and System Objects

An important aspect of Naev is the universe. The universe is formet 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 editting 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.

4.1 Systems

TODO

4.1.1 Universe Editor

TODO

4.1.2 System XML

TODO

4.1.3 System Tags

TODO

4.1.4 Defining Jumps

TODO

4.1.5 Asteroid Fields

TODO

4.2 System Objects (Spobs)

TODO

4.2.1 System Editor

TODO

4.2.2 Spob XML

TODO

4.2.3 Spob Tags new

TODO

4.2.4 Lua Scripting

TODO

4.2.5 Techs

TODO

Chapter 5

Outfits

TODO

5.1 Slots

TODO

5.2 Ship Stats

TODO

5.3 Outfit Types

TODO

5.3.1 Modification Outfits

TODO

Chapter 6

Ships

TODO

6.1 Ship Classes

TODO

6.2 Ship XML

TODO