Stellar Magnate Documentation

Release 0.1

Toshio Kuratomi

Sep 30, 2017

Contents:

In the late 1980s I played a game called Planetary Travel by Brian Winn. The game was a textual trading game where you had to take a small amount of money and a space ship and travel between the planets of our solar system to make money.

Stellar Magnate is a game written in that same genre but updated and enhanced. Unlike the original Integer Basic that Planetary Travel was written in, this is written in Python and makes use of asynchronous programming, abstractions to allow multiple user interfaces, and other modern programming practices.

Stellar Magnate was written both to enjoy a little bit of nostalgia and to have a practical problem on which to experiment with new technologies. I hope that the game is somewhat enjoyable and the core is simple enough that new aspiring programmers can take a look at how it works to make it their own.

CHAPTER 1

Design Documentation

These pages document the design decisions for Stellar Magnate

Mechanics

This document describes the game mechanics

Market

Current

Random pricing based upon a normal distribution.

Future

Need a cyclical market pricing. This way prices rise or fall for a certain duration. This makes the user take more care to judge whether the item's price is on its way up or down.

Cycle should be mostly time based so that we can recalculate the price when a ship actually lands there rather than for every tick.

Will need to save the current state of the market so that loading a game starts in the same place.

- Markets only have so many goods that they're willing to buy or sell
- Markets should be rated for population, industrialization, agriculture, mining. These affect supply and demand of categories of goods.
- Markets will accept categories of product even if they don't sell them.
  - Supply and demand drives prices. So if a market does not sell an item, there will be a low supply. However, demand needs to be based off of how different it is from what is already there.

    * So perhaps categories need to be hierarchical. The farther away on the hierarchy from something already sold, the less demand

Ship

Current

- Purchase more cargo space

Future

- Purchase different ships
- Each ship that you can purchase has different characteristics

Fleets

- We can organize ships into separate fleets
- Can send fleets off to trade in different locations
- Give fleets different orders about buying and selling

Finances

Current

One bank. Accessible whenever the user is on Earth.

Future

Multiple banks.

- The Syndicate: * Can do business everywhere * Loans money * High interest rates * Deposit low interest rate * 100% safe * May be inaccessible 50% of the time but never twice in a row
- System wide bank – Solarian National Bank * May not be present on pirate worlds. * Standard loans * Deposit low-medium interest * Safe unless war
- Capital planet bank – First Terran Bank and Trust * Available on a subset of system worlds * May open/close branches. Pick X% most industrial/civilized in-system worlds. * May sometimes bankrupts but bailout 75-95% of owed money * standard loan * moderate interest * War may lose all
- Planetary bank – Mercury Savings and Loan * Available 1 world * Low interest loans * Deposits moderate interest * Sometimes bankrupts bailout 0-50% * War may lose all

Movement

Current

Travelling from one location to another constitutes one turn

Future

- Markets have distances associated with them. takes longer to travel from Mercury to Pluto than it does to travel from Venus to Earth
- Orbits? Planets change positions?
- Fuel costs to travel

Combat

Current

- Compare number of enemies with number of weapons
- Weapons hit X enemies per turn
- Enemies cause X damage per turn

Future

- Different ships can protect in differing amounts
- Different ships have different speeds
- Order ships on a grid
- Enemies encounter the grid of ships

Events

PubMarine is used to pass events from the UI to the Dispatcher. These are the events that are emitted over the course of the program. Events normally report information about changes but we also use it as a general bus between the frontend and backend to pass commands and return data as well. This allows the frontend and backend to operate asynchronously in some cases, decouples the precise APIs from each other, and allows us to pass a single PubPen object around to handle communication instead of having to pass references to each object whose methods we wanted to invoke.

User events

User events return information about user objects.

`user.cash.`**`update`**(*new_cash: int*, *old_cash: int*)
Emitted when a change occurs in the amount of a user's cash on hand

Parameters

- **`new_cash`** (*int*) – The amount of cash a user now has
- **`old_cash`** – The amount of cash the user had before

`user.`**`login_success`**(*username: string*)
Emitted when a user logs in successfully.

Parameters **`username`** (*string*) – The username that successfully logged in

user.**login_failure**(*msg: string*)
Emitted when a login attempt fails

Parameters **msg** (*string*) – A message explaining why the attempt failed

user.**info**(*username: string*, *cash: int*, *location: string*)
Emitted in response to a *query.user.info()*. This comtains all relevant information about a user.

Parameters

- **username** (*string*) – The user who the information is about
- **cash** (*int*) – The amount of cash the user has on their person
- **location** (*string*) – The location that the user is in currently

user.**order_failure**(*msg: string*)
Emitted when an order attempt fails

Parameters **msg** (*string*) – A message explaining why the attempt failed

Ship Events

Ship events return information about ship objects.

ship.cargo.**update**(*amount_left: ManifestEntry*, *free_space: int*, *filled_hold: int*)
Emitted when a ship's cargo manifest changes (commodities are bought and sold or transferred to a warehouse)

Parameters

- **amount_left** (*ManifestEntry*) – A magnate.ship.ManifestEntry that shows how much of a commodity is left on board.
- **free_space** (*int*) – Amount of the hold that's free
- **filled_hold** (*int*) – Amount of the hold that's filled

ship.**destinations**(*destinations: list*)
Emitted when the destinations a ship can travel to changes. This usually means that the ship has moved to a new location which has different options.

Parameters **destinations** (*list*) – A list of strings showing where the ship can travel from here.

ship.equip.**update**(*holdspace: int*)
Emitted when a ship's equipment changes

Parameters **cargo_space** – The total cargo space in the ship currently has

ship.**info**(*ship_type: string*, *free_space: int*, *filled_space: int*, *manifest: dict of ManifestEntry*)
Emitted in response to a query.ship.info(). This contains all relevant information about a ship.

Parameters

- **ship_type** (*string*) – The type of ship
- **free_space** (*int*) – How much hold space is available
- **filled_space** (*int*) – How much hold space is used
- **manifest** (*dict*) – The commodities that are in the hold. This is a dictionary of ManifestEntry types

ship.**moved**(*new_location: string*, *old_location: string*)
Emitted when a ship changes location.

Parameters

- **new_location** (*string*) – The location that the ship moved to
- **old_location** (*string*) – The location that the ship moved from

ship.**movement_failure**(*msg: string*)
Emitted when a ship attempted to move but failed.

Parameters **msg** (*string*) – A message explaining why the movement failed

Market Events

Market events carry information about a specific market to the client.

market.**event**(*location*, *commodity*, *price*, *msg: string*)
Emitted when an event occurs at a market. This is for informational purposes. The client may choose to display the message for game flavour. Once markets become stateful, this may become more useful.

Parameters **msg** (*string*) – A message about the market

market.{location}.info(prices: dict)
Emitted in response to a `query.market.{location}.info()`. This carries information about prices of all commodities in a market.

Parameters **prices** (*dict*) – A mapping of commodity name to its current price

market.{location}.purchased(commodity: string, quantity: int)
This contains information when a user successfully purchases a commodity at a specific market.

Parameters

- **commodity** (*string*) – The name of the commodity that was bought
- **quantity** (*int*) – The amount of the commodity that was purchased

market.{location}.sold(commodity: string, quantity: int)
This contains information when a user successfully sold a commodity at a specific market.

Parameters

- **commodity** (*string*) – The name of the commodity that was sold
- **quantity** (*int*) – The amount of the commodity that was sold

market.{location}.update(commodity: string, price: int)
Emitted when the price of a commodity changes.

Parameters

- **commodity** (*string*) – The name of the commodity being operated upon
- **price** (*string*) – The new price of the commodity

Action Events

Action events signal the dispatcher to perform an action on behalf of the user.

action.ship.**movement_attempt**(*destination: string*)
Emitted when the user requests that the ship be moved. This can trigger a *ship.moved()* or *ship.movement_failure()* event.

Parameters **destination** (*string*) – The location to attempt to move the ship to

action.user.**login_attempt**(*username: string*, *password: string*)
Emitted when the user submits credentials to login. This can trigger a *user.login_success()* or *user.login_failure()* event.

Parameters

- **username** (*string*) – The name of the user attempting to login
- **password** (*string*) – The password for the user

action.user.**order**(*order: magnate.ui.event_api.Order*)
Emitted when the user requests that a commodity be bought from a market. Triggers one of market.{location}.purchased(), market.{location}.sold(), or *user.order_failure()*.

Parameters **order** (*magnate.ui.event_api.Order*) – All the details necessary to buy or sell this commodity.

See also:

magnate.ui.event_api.Order

Query Events

These events are requests from the frontend for information from the backend. This could simply be to get information during initialization or it could be to resynchronize a cache of the values if it's noticed that something is off.

query.cargo.**info**()
Emitted to retrieve a complete record of the cargoes that are being carried in a ship. This triggers a ship.cargo() event.

query.market.{location}.info()
Emitted to retrieve a complete record of commodities to buy and sell at a location.

query.user.**info**(*username: string*)
Emitted to retrieve a complete record of the user from the backend.

Parameters **username** (*string*) – The user about whom to retrieve information

query.warehouse.{location}.info()
Emitted to retrieve a complete record of the cargoes being held in a location's warehouse.

UI Events

UI events are created by a single user interface plugin for internal communication. For instance, a menu might want to communicate that a new window needs to be opened and populated with data. All UI events should be namespaced under ui.[PLUGINNAME] so as not to conflict with other plugins.

Urwid Interface

These are UI Events used by the Urwid interface. Urwid has its own event system but using it requires that the widget that wants to observe the event must have a reference to the widget that emits it. When dealing with a deep hierarchy of widgets it can be painful to pass these references around so the Urwid interface makes use of our pubmarine event dispatcher for some things.

ui.urwid.**message**(*msg: string*, *severity=MsgType.info: magnate.ui.urwid.message_win.MsgType*)
Emitted to have the message window display a new message.

Parameters

- **msg** – The message to display to the user
- **severity** – Optional value that tells whether the message is merely informational or informs the error of some error. The message_win will display more severe messages with special highlighting.

ui.urwid.**order_info**(*commodity: string*, *price: int*)

Emitted to inform the transaction dialog what commodity and price the user is interested in.

Parameters

- **commodity** (*string*) – Name of the commodity to buy or sell
- **price** (*int*) – Price of the commodity

Data Model

The data in Stellar Magnate is divided into two categories: static data that defines base information and dynamic data that changes over the course of the game.

Static data is shipped in yaml files with the program. When a new Stellar Magnate game is started, the yaml files are loaded into a database to initialize the program.

The dynamic data is generated by the game as it moves along. It is persisted into separate tables in the database as the game progresses.

Static data and dynamic data reside in different tables in the database but the dynamic data can reference the static tables.

Static Data

Shipped format

Game Dependent

These are various records that only the game ships because there needs to be new game logic whenever something is added to them.

location_type enumeration of known location types. :surface: On the surface of a celestial body :orbital: Orbiting a celestial body in the system :dome: Exists under a dome because the ecosystem is otherwise non-habitable

planetoid_type enumeration of planetoid types - oceanic - gas giant - rocky - oxygen - methane

moon_type enumeration of planetoid types

commodity_type enumeration of known commodities. - food - metal - fuel - low bulk chemical - low bulk machine - high bulk chemical - high bulk machine

finance_type enumeration of known bank types. - mafia - system - capital - planetary

order_status_type The status of a transaction - presale - submitter - rejected - finalized

ship_type

- Passenger
- Cruiser
- Etc

equipment

ship

- Cargo hauler

property

- Warehouse

ship parts

- Cargo module
- Laser

Extendable by Server Owners

system list of stellar systems

name name of the stellar system

possessive If a location's name is not unique, then the possessive is applied to the name for display purposes

planetoid A list of bodies orbiting the system's sun

name name of the planetoid

type type of the planetoid

moon A list of bodies orbiting the planetoid

name name of the moon

type type of the moon

location A list of inhabited locations in the Stellar System

name A location's human name

place

type Type of the location

Someday locations will have other characteristics like:

- Orbit velocity
- Initial position
- Orbit radius
- type jump will have locations that can be jumped to
- population. Population affects the amount of commodities that a location needs or produces
- **population_origin: list of stellar systems that the population originates in. Affects** commodities for sale
- **commodity consumption and production information**
  - Production should be of a specific commodity
  - Consumption can be of a commodity or commodity_type
  - These numbers should be weightings. Production weighting affects how likely (or how much) a place is to sell an item

- Consumption affects how much of an item a place is likely to take

commodity list of commodities that can be bought and sold :name: name of the commodity :type: Type of the commodity :mean_price: average price of the commodity :standard_deviation: How much money is there in one std deviation :depreciation_rate: How quickly the value of a product degrades :space: Volume the commodity consumes

financial_institution list of financial institutions in these stellar systems :name: name of the institution :type: type of institution

Someday commodity should grow a weight attribute. Weight will affect a fee for transporting the commodity to the ship. (Perhaps. Perhaps there needs to be a way for this to be offset. Rockets are inefficient weight. Space elevators are volume constrained. Counter grav can land a whole ship but only high tech worlds and weightless environments (orbital facilities) have it?) To implement this, locations would need to grow a surface_to_orbit type

event list of events that can change the price of goods :msg: Description of the event. Like headline and subheading of a news article :affects: list of commodities affected by the event

commodity_name The specific commodity that this event effects. One of name or type is filled

commodity_type A type of commodity that this event affects. One of name or type is filled

adjustment A numeric value to adjust the price by

adjustment_type How to treat the adjustment. Can be offset to add the adjustment value to the price or percent to change the price to adjustment percent of the current price.

ship A user's ship :type: Type of ship :mean_price: Average price of the ship :standard_deviation: How much the ship's price fluctuates :depreciation_rate: How quickly the resale value of the ship drops :holdspace: How much space the ship has for cargo :weaponmount: How many weapons the ship can have ready to fire

Database Schema

This closely mirrors the Shipped format. The static data should live in separate db tables from dynamic data and be associated via foreign keys. This allows for easier changes to the static data if an update occurs.

Tables for static data should all have _data as a suffix

Linter

There should be a static_data linter that does the following:

- Verifies the schema
- Checks everything for spelling
- Assembles a list of commodity types (in commodity and in event) and asks for confirmation if any of the commodity_types are unknown
- All location names must be unique within a stellar_system
- All stellar_system names must be unique within the game
- All commodity names must be unique within their system
- All types (for locations, commodities, etc) must be known to the game

Dynamic Data

These pieces of data are per game instance. They change as the game progresses

Database Schema

All records have a system created id

players Table of players :username: Player's handle :display_name: If set, an alternate name the player can go by :password: hashed password that the player can use to verify themselves :cash: Amount of cash on hand

epoch Number of "ticks" since the game was created. Eventually a number of calculations will include the epoch

bank_accounts Table of bank accounts that players have :bank_id: foreign key to bank :loan amount: amount on loan :loan_rate: interest rate

commodity

price The present price this sells for

commodity_data_id

location_id location that this is selling at

order

location location at which the sale takes place

commodity_id commodity being bought or sold

price Amount at which the user is agreeing to buy or sell

hold_quantity Amount of the commodity to place in the ship's hold

warehouse_quantity Amount to place in or draw from the player's warehouse

buy True if this is a buy order. False if it's a purchase order

status status of the order

originated timestamp for when the order was placed

manifest

commodity_data_id Commodity which this is

quantity Amount of the commodity

price_paid Average price paid for this entry. Can be used to show profit and loss reports

average_age Average age this was bought ago. Used for depreciation calculations

ship_id What ship this cargo is a part of

ship

ship_data_id Which type of ship this is

location_id Where the ship is at

ship_parts

ship_part_data_id Link to the ship part that this implements

ship_id Link to the ship that this is mounted on

Urwid UI

The Urwid UI backend is a text based backend. It implements a series of static screens and menus that the user interacts with to manage their fleet.

See also:

Displaying on the console uses the Urwid library

Naming Conventions

Toplevel UI elements in the Urwid code are called Screens. Most Screens are single area forms. The `MainScreen` is the exception to this. This complex Screen is composed of various constant status windows displaying information to the user about their ship and surroundings, a menubar to select options from and an area for contextual content which the user can interact with. The widgets displaying information are calld Windows and those that display the contextual content are called Displays.

Each Display shows the user information upon which they can act. For instance, the `MarketDisplay` allows the user to select `Commodities` to buy and sell.

Sometimes a Display needs to popup a subelement that allows the user to input more information. These widgets are named Dialogs.

Mockups

Here there are various mockups of screens in the Urwid UI

Splash Screen

```
+------------------------+
      Stellar Magnate
      1.0
      (c) Toshio Kuratomi
+------------------------+
```

Status Bar

This is part of the frame around the Main Window:

```
+=Name: Hiormi -------------- Location: Earth -+
```

Menubar

This is the top level user interaction for the Main Window:

```
+-------------------------------------------------------------------+
| (P)ort District  Ship(Y)ard  (F)inancial  (T)ravel  (M)enu        |
+-------------------------------------------------------------------+
```

Travel Display

This is simply a text menu that allows the user to choose a destination planet:

```
+----------------------+
| (1) Mercury
| (2) Venus
| (3) Earth
| (4) Luna
| (5) Mars
| (6) Jupiter
| (7) Saturn
| (8) Uranus
| (9) Neptune
| (0) Pluto
|
| (!) Jump
+----------------------+
```

Menu Dialog

The Menu Dialog lets the user perform out-of-character functions like quitting the game:

```
+----------+
| (s)ave
| (l)oad
| (q)uit
| (c)ontinue
+----------+
```

Market Display

Displays `Commodities` that the user can buy and sell to turn a profit:

```
+- Commodity -- Price - Quantity --- Hold ------ Warehouse --+
| (1) Grain     $10     7.5E+200     7.5E+200    7.5E+200
| (2) Metal     $100    1.7E+5       10          100
| (3) Weapons   $2000   3.4E+21      1000        0
| (4) Drugs     $10.1K  2.0E+10      10          0
+-----------------------------------------------------------+
```

Cargo Order Dialog

The Cargo Order Dialog lets the user input quantities of a `Commodity` that they wish to buy or sell. The Dialog has a way to toggle between buying or selling the commodity.

Buy Mode:

```
+------------------------+
| (o) Buy ( ) Sell
| Hold: XXX  Warehouse: YYY
| Total cost: $XXX
| hold/warehouse    (H)/(W)
```

```
| Quantity [_____] [MAX]
+------------------------+
```

Sell Mode:

```
+------------------------+
| ( ) Buy (o) Sell
| Hold: XXX  Warehouse: YYY
| Total sale: $XXX
| Quantity [_____] [MAX]
+------------------------+
```

Port Display

We will eventually have distinct types of ships to buy and sell but for the initial release we'll just adopt the Planetary Travel style of having a spaceship that we can buy additional cargo modules for. The Port Display lets the user buy equipment (additional cargo modules, warehouse space, shipboard weapons, etc):

```
+- Equipment -------- Price --- Current --+
| (1) Cargo Space     $10K     1000
| (2) Lasers          $5000       1
| (3) Warehouse       $15K    20000
+-----------------------------------------+
```

Equipment Order Dialog

The Equipment Order Dialog lets the user fill in additional information for buying equipment for their ship:

```
+------- Hold space - $42 ---+
| Total Sale: $0
| (o) Buy (o) Sell
| Current Amount:
| [MAX] Quantity [_____]
|              [Place Order][Cancel]
+------------------------+
```

Info Window

The Info Window sits alongside the Display in the Main Screen and shows an overview of statistics about the player and ship:

```
+-----------------+
| Ship:
|   Minnow
| Type:
|   Freighter
| Free Space:
|   1000
| Cargo:
|   500
| Warehouse:
|   10000
| Transshipment:
```

```
|   10
| Bank:
|   $1K
| Cash:
|   $1.5Mil
| Loan:
|   $0
|
+-----------------+
```

Financial Display

This allows the user to deposit money and take out loans:

```
+-----------------+
| (1) The Syndicate
| (2) Solarian National Bank
| (3) First Terran Bank and Trust
| (4) Mercury Savings and Loan
+-----------------+
```

The Syndicate Mob; high limit; high interest loan. Present anywhere. Deposit: low interest, 100% safe, but may be 0-50% inaccessible at any given time

System-wide bank May not be present on pirate world. std loan. Deposit: low interest, Safe unless war.

Capital planet bank Available on subset of system worlds. May open/close branches. std loan. Deposit moderate interest. Sometimes bankrupts but bailout 75-95%. War, may lose all

Planetary bank Available 1 world. low interest loan. Deposit moderate interest. Sometimes bankrupts. bailout 0-50%. War, may lose all.

Financial Dialog

Allows the user to select what they want to do at the financial institution they selected:

```
+--------------+
| (d)eposit
| (w)ithdraw
| (i)ncrease loan
| (r)epay loan
+--------------+
```

Ship Update

These are some thoughts on how to change the Ships for later versions

Fleet

```
+------------+
| (s)hip
| (w)eapons
+------------+
```

Ship

```
+- (B)uy ---------------+- (S)ell -----------+
|                       |
| (1) Scout [x2]  $1K   | (1) Tug [x3]        $1K
| (2) Tug   [x1]  $1.2K | (2) Freighter [x1]  $500
| (3) Freighter   $1.3K |
| (4) Cruiser     $1M   |
| (5) Carrier     $8T   |
+-----------------------+--------------------+
```

Purchase Ship

```
+------------------------+
| Ship Type: Scout
| Cargo:  100
| Weapon Space:  5
| Upkeep:  $20K/yr
| Cost per: $1,000
| Supply 3
|
| Total cost: $XXX
| Purchase Quantity [_____] [MAX]
+------------------------+
```

Weapons

```
+- (B)uy----------------+--- (S)ell ------------+
| (1) Laser [x10] $1K   |  (1) Laser [x2] $500
+-----------------------+-----------------------+
```

Purchase Weapons

```
+------------------------+
| Weapon Type: Laser
| Space:  5
| Upkeep:  $1K/yr
| Cost per: $1,000
| Supply: 10
|
| Total cost: $XXX
| Quantity [_____] [MAX]
+------------------------+
```

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Index