Grafos Burrito — Technical Manual

Version: 1.0

Audience: Developers and maintainers who want to understand, extend, or maintain the codebase.

Introduction

Grafos Burrito is a didactic and exploratory software project designed to visualize, analyze, and interact with graph-based data structures through a dynamic interface built using **Pygame**. The project originated as a practical exercise in graph theory and object-oriented programming, merging technical depth with creative visualization.

This manual provides developers and maintainers with a comprehensive understanding of the system's architecture, components, and conventions. It also explains how to set up the environment, extend the application with new views or models, and troubleshoot potential issues. The document is intended to serve as both an onboarding guide and a long-term reference for the project's technical foundation.

1 Project Summary

Grafos Burrito is a Pygame-based desktop application to visualize and edit graph-based representations of constellations and related domain objects (a "burro" object and mission parameters). The project follows a clear separation of concerns:

- `main.py` — bootstraps the application and initializes the view manager and views.
- `screens/` — contains view classes that implement UI screens and input handling.
- `models/` — defines domain objects (Graph, Star, Burro, etc.).
- `config/loader.py` — JSON loader that populates Graphs and Stars from `data/constellations.`
- `assets/` — images and audio used by the UI.

The app uses a `ViewManager` (in `screens/manager.py`) to register, switch, and dispatch events to views.

2 How to Run (Developer)

1. Create a Python 3.10+ environment.
2. Install dependencies:

```
pip install -r requirements.txt
```

3. From the project root, run:

```
python main.py
```

Development notes:

- Use an editor that supports Python 3.10+ type checking and linting.
- The project has no automated tests by default; add unit tests under `tests/` when adding new logic.

3 Repository Layout and Responsibilities

- `main.py` — application entrypoint. Initializes Pygame, loads data using `config/loader.py`, creates and registers views, and runs the main loop.
- `config/loader.py` — loads `data/constellations.json` and returns a tuple `(constellations, burro_data, mission_params)`.
- `data/constellations.json` — primary data store.
- `models/graph.py` — Graph class and edge management.
- `models/star.py` — Star model with properties like `id`, `label`, `coordinates`, etc.
- `models/burro.py` — Burro domain model (project-specific behavior).
- `screens/*.py` — UI views and `ViewManager`.
- `utils/` — helper utilities.
- `assets/` — images and audio.

4 JSON Data Schema (Summary)

```
{
  "constellations": [
    {
      "name": "Name",
      "color": [R,G,B],
      "stars": [
        {
          "id": <unique id>,
          "label": "optional label",
          "coordinates": {"x": <number>, "y": <number>},
          "radius": <number>,
          "timeToEat": <number>,
```

```
          "amountOfEnergy": <number>,
          "hypergiant": <bool>,
          "timeToResearch": <number or null>,
          "linkedTo": [
            {"starId": <id>, "distance": <number>}
          ]
        }
      ]
    }
  ],
  "burro": { ... },
  "missionParams": { ... }
}
```

The loader returns `(constellations, burro_data, mission_params)` where:

- `constellations` is a list of `Graph` instances.
- `burro_data` contains the JSON object under `"burro"`.
- `mission_params` holds mission configuration data (defaulted if absent).

5 Key APIs and Conventions

ViewManager (used in `main.py`):

- `register_view(name, instance)` — register a view instance.
- `set_view(name)` — change current view.
- `handle_event(event)` — forward input events.
- `update(dt)` and `render(screen)` — update and draw current view.

Graph and Star models:

- Graph contains a list of stars and edges.
- Star instances hold geometry and domain attributes (`timeToEat`, `amountOfEnergy`, etc.).

6 Extending the App — Add a New View

1. Create a new file in `screens/`, e.g., `my_view.py`, defining:

```
class MyView:
    def handle_event(self, event): ...
    def update(self, dt): ...
    def render(self, screen): ...
```

2. In `main.py`, register the view:

```
manager.register_view("my_view", my_view_instance)
```

3. Switch to it:

```
manager.set_view("my_view")
```

7 Editing and Saving Data

- The application reads `data/constellations.json` at startup.
- Some views allow in-UI editing; ensure persistence by implementing a save helper if necessary.

8 Debugging Tips

- Print or log inside `update`, `handle_event`, and `render` for debugging.
- Check asset paths if missing files cause errors.
- Validate JSON files before loading them.

9 Suggested Improvements and Future Work

- Add unit tests for loader logic and graph algorithms.
- Implement save/export functionality.
- Add type hints and static analysis (mypy).
- Package for distribution (e.g., PyInstaller).

10 Contact and Contribution

Follow repository contribution guidelines (if available). Open issues or pull requests for bugs or feature requests.

End of Technical Manual