A1G Code Review - By Stephen Davies

Documentation:

The code is well documented, without being over the top. The comments are descriptive and mention describe not only "what" parts of the code do, but "why" as well. For example `dialog.cpp:129`:

```
// use a QPointF and radius so the body is centered correctly
```

The only file where a lack of comments is noticeable is `config_section.cpp`, but the code in this file is fairly self-explanatory anyway.

There is a slight lack of consistency with comment types. Some comments use inline comment syntax (`//`) some use block comment syntax (`/* */`). For example `config_section.h:21-23`:

```
/*
 * Creates a new ConfigSection with the given name
 */
```

Whereas `config_keyvalue.h:20-21`:

```
// Key and value getters:
// Keys and values are stored as strings, so these return std::string.
```

Extensibility:

The code was well designed for extensibility in general. Specifically the configuration file system is very easily extended to include clusters and zodiacs. So much so that I didn't need to modify the configuration file system much at all. This is because the code to parse the configuration is broken down into the classes:

- `Config` (which handles the tokenisation using regexes)
- `ConfigSection` (which groups key & value pairs together for each heading `[Object]` in the config file), and
- `ConfigKeyValue` (which stores the value with the key and deals with type conversions of that value)

Then a Builder class was used to build bodies from the `ConfigSections`

Design

The code used two design patters: Prototype and Builder.

Using the Builder patter was a good choice because it moves the complexity of building the `Body` class from a `ConfigSection` into a separate class. It also makes dealing with missing config parameters easy, as you don't need to have many different constructors to allow for parameter defaults.

Prototype was also a good choice because it hides the complexity of building a body class using the constructor. It works well with the Builder pattern as well, because Prototype can clone a typical instance for Builder to build upon.

Implementation

The implementation was mostly excellent, but there are a couple of times where the SOLID principles are broken including:

- `Dialog` has many too tasks. It coordinates the building of the planetary system, owns the bodies, loops over & accumulates forces for each body during physics calculations and displays the planetary configuration
- `Body` is open rather than closed for the two functions `Body::addAttraction` and `Body::update`. The caller of these function is relied on to accumulate the forces on the `Body`, when this could be done with private variables which accumulate the force applied by each other `Body`, then applied to the body and zeroed when `Body::update` is called.

Style

The layout used is consistent, with every file using 2 spaces for indentation. Code cliches used such as the C++11 range-based for loop are used consistently. Naming conventions such as:

- `m_` prefix for private member variables
- UPPERCASE defines and globals
- camelCase local variable names
- CamelCaps class names

are applied consistently throughout the code. A couple of style inconsistencies are:

- In `body.h`, the private member variables are declared above the public functions. Elsewhere private variables are always at the end of class definitions, below the public and protected sections
- In `config.h` the static variables `REGEX_SECTION`, `REGEX_PAIR` and `REGEX_COMMENT` are named in UPPERCASE even though they are not global variables or defines. However they act as constants, so it's not clear whether this truly is a breach of the aforementioned naming conventions followed throughout the rest of the code, or just a slightly different interpretation of them.

Written in Markdown and converted to pdf with http://www.markdowntopdf.com.