SolarSystem Project Documentation

Introduction

The `SolarSystem` project is a simulation of our solar system that reads planet data from a JSON file and visualizes the solar system using OpenGL and GLUT. This guide will help you compile and run the project on both macOS and Linux platforms.

Prerequisites

- CMake version 3.10 or above
- A C++17 compliant compiler
- The Boost library, with the `system` component
- OpenGL and GLUT libraries

Compilation

1. **Set up the Build Directory**
   It's recommended to create a separate directory for building the project, typically named `build`. Navigate to your project root and do:

   ```
   mkdir build
   cd build
   ```

2. **Run CMake**
   While inside the `build` directory, run CMake to generate the necessary Makefiles.

   ```
   cmake ..
   ```

3. **Compile the Project**
   After successfully running CMake, compile the project with:

   ```
   make
   ```

   This will produce an executable named `solar_system`.

Running the Program

To run the `solar_system` program, you need to provide a JSON file that contains the data of the solar system. An example JSON file `simple.json` is provided with the project.

From the `build` directory, run:

```
./solar_system ../simple.json
```

Interacting with the Animation

Once the program is running, you can drag the animation with your mouse. This allows you to view the solar system from different angles and get a closer look at individual planets.

Notes

- The JSON file structure is hierarchical with planets orbiting the sun and satellites orbiting the planets. Each celestial body has properties like `size` , `color` , `orbit` , and `period` that determine its appearance and movement in the simulation.
- The CMakeLists.txt has been set up to differentiate between macOS and Linux platforms and will compile the appropriate source file (`my_solar_system_mac.cpp` for macOS and `my_solar_system_linux.cpp` for Linux) based on the platform detected.

Description

1. **Purpose**: The script is a simulation of a solar system using OpenGL to visualize it. Planets in the system revolve around their parent celestial objects, such as moons around planets or planets around stars.

2. **Libraries**:

   - Standard libraries like `<cmath>` , `<fstream>` , `<iostream>` , `<sstream>` , `<chrono>` , and `<map>` .
   - OpenGL and GLUT libraries to handle graphical rendering and windowing system interaction.
   - Boost's Property Tree for JSON parsing and manipulation.

3. **Global Variables**:

   - `main_window` : keeps track of the main OpenGL window's ID.
   - `world_P0` and `world_P1` : Define the corners of the world (or viewing) space.
   - `planetAngles` : A map that associates the name of a planet with its current angle of rotation.
   - `solar_system` : Represents the entire solar system using Boost's property tree.

- `t0` : The starting point for the timer.
- `time_scale` : Scaling factor for time.
- `ViewPort` : Contains the viewport settings for the OpenGL window.
- `InvPrj` : Likely an inverse projection matrix.
- Drag and drop related variables: `isDragging` , `lastX` , `lastY` , `offsetX` , and `offsetY` .

4. **Key Functions**:

- `circle()` : Draws a circle in OpenGL.
- `initiate_planets()` : Initiates planets and their initial angles.
- `draw_hierarchy()` : A recursive function that draws celestial objects and their children (e.g., moons).
- `simulate()` : Sets up a timer callback to continuously redraw the solar system.
- `matrix4x4_inv()` : Computes the inverse of a 4x4 matrix.
- `reshape()` : Called when the OpenGL window is resized. Adjusts the viewport and potentially the projection to fit the new window size.

5. **Implementation Details**:

- The script uses recursive techniques to traverse the celestial hierarchy and draw each object.
- It appears to handle mouse dragging operations, though the functionality isn't visible in the provided code.
- The script uses a timer to continuously simulate and redraw the solar system.
- Planets and other celestial objects can have properties like size, color, orbit distance, and rotation period, which are stored in the `solar_system` property tree.
- The simulation updates the angle of rotation for each planet based on elapsed time and the planet's rotation period.

However, this is just a partial script. Depending on how the rest of the script is structured, it might handle user interactions, loading data from external sources, or additional rendering effects.

Setting up SolarSystem Project on Fedora 38

Installing Dependencies

To successfully set up and run the SolarSystem project on Fedora 38, you'll first need to install all necessary dependencies.

1. **Ensure Basic Development Tools**
   Make sure you have `cmake` and the necessary compilers installed:

   ```
   sudo dnf install make gcc-c++
   ```

2. **Installing OpenGL Dependencies**
   For OpenGL support, you will need to install the following packages:

   ```
   sudo dnf install glew-devel SDL2-devel SDL2_image-devel glm-devel freetype-devel
   ```

3. **Installing GLUT**
   For GLUT (OpenGL Utility Toolkit), install the `freeglut-devel` package:

   ```
   sudo dnf install freeglut-devel
   ```

4. **Installing Boost Library**
   Boost provides free, peer-reviewed, portable C++ source libraries. To install:

   ```
   sudo dnf install boost-devel
   ```

After installing all these packages, you should be ready to compile and run the SolarSystem project on Fedora 38.