SOFT-356_Solar-System

Solar system simulator with accurate gravity **without** using a physics engine as a dependency.
This repository is at https://github.com/tobysmith568/SOFT-356_Solar_System
Video report is at https://drive.google.com/open?id=1nx_LxbCpoDKki8jtrdrLewbKNt7wg1j8

For the quickest setup:

- **If you clone this repository in full, then the software will work instantly in debug mode from within Visual Studio after following the steps in the Compilation section.**
- **If you compile and wish to use the `.exe` then you should take the `config.dat` (Solar%20System/config.dat) and `planets.dat` (Solar%20System/planets.dat) files from this repo as well as the `Models` (Solar%20System/Models) and `Shaders` (Solar%20System/Shaders) directories. Both files and both folders should be placed next to the `.exe`.**

Software and Libraries

- Visual Studio 2019 v16.3.9
- GLM v0.9.9.600
- nupenGL v0.1.0.1

Usage

Compilation

The only setup for compiling this solution is to restore the NuGet packages. This software has been written only with Windows in mind.

Execution

The `.exe` can be run by double-clicking it, but should any errors occur and the program shut down then you will not be able to see the error message. As such, it is recommended that you open a command prompt, navigate to the directory of the `.exe` and run it from within that prompt. This will keep the window open when the program closes, and then any error messages can be read.

Config

Config.dat

All the functionality of this software is configurable via the [config.dat file (Solar%20System/config.dat)] located next to the `.exe`. Should the file not exist, then the program will attempt to create it on startup. If the program has to create the `config.dat` file, then it will also create the default shaders and default planet configuration alongside it. Config within that file is a list of key/value pairs and sits in the following categories:

- Window Setup
- Shader Setup
- Planet and Model Setup
- Behaviour Setup
- Keybindings

Boolean options evaluate to `true` if they are equal to `1`, else they are `false`.

Planet and Model Setup

This section of the config file configures the planets. `planetFile` should be a path to a planet file. `sunModel`, `planetModel`, and `backgroundModel` should all be paths to `.obj` files for the scene to use. By default this repository comes with some lower-poly and higher-poly models, either can be commented in/out to change the functionality of the program. The low-poly models should take about 3 seconds to load in while the higher-poly models should take around 10-12 seconds. While they take longer to load, the higher-poly models are recommended. `sunScale` and `backgroundScale` tell the program how it should scale up/down these models; the default values given are ideal for the default model files.

Behaviour Setup

- `physicsEnabled`: `True` means that physics will be enabled when the program starts. This can also be toggled at runtime with a keybinding
- `lockSun`: This will lock the sun in place; other objects will have no gravitational effect on it, but it will still affect them. While this is not realistic, it creates more stable orbits
- `startOnFPSStyle`: `True` will start the camera in an FPS style mode, `False` will start in an orbit mode. This can also be toggled at runtime with a keybinding
- `movementSpeed`: The movement sensitivity when in FPS mode
- `mouseSpeed`: The mouse 'looking' sensitivity for both camera types
- `scrollSpeed`: The zooming in and out sensitivity

Keybindings

All of the key bindings within the config file begin with the prefix `KeyBinding_`. In this README, key bindings are shown without the prefix and then give the default key in brackets, e.g., `MoveForward (W)` or `Reload (R)`. In the config file, the bound key is represented by a number, the relationships between the keys on a keyboard and these numbers can be found here on the GLFW website (https://www.glfw.org/docs/latest/group__keys.html).

- `Quit (Esc)`: Quits the application
- `Reset (R)`: Resets all the models to their initial state
- `MoveForward (W)`: Moves the camera forwards when in FPS mode
- `MoveBackward (S)`: Moves the camera backwards when in FPS mode
- `MoveLeft (A)`: Moves the camera left when in FPS mode
- `MoveRight (D)`: Moves the camera right when in FPS mode
- `ToggleFPSStyle (T)`: Toggles the camera between an FPS mode and an orbit mode
- `TogglePhysicsEnabled (P)`: Enables or disables physics

The mouse scroll wheel can also be used to zoom in and out.

Planets.dat

This file describes the solar system that the program will create. The first line of the file is to set up the sun. This is only a single number, and it is the weight of the sun in kilograms. Each following line in the file represents a planet. Planet lines can be commented out by placing a hash (`#`) at the very beginning of the line. The keys can be in any order or can be omitted, they are:

- `name`: Not currently used by the program but allows you to label the planet within the file
- `mass`: This is the mass of the planet in kilograms
- `distance`: This is the initial distance for the centre of the planet to be from the centre of the sun in metres
- `radius`: This is a multiplier for how large this planet should be represented in the scene relative to the sun. E.g.: `1` means the planet will graphically be the same size as the sun, `0.5` means the planet will be shown to be half the size of the sun
- `initialForce`: This is a force applied to the planet one time when after physics is initially enabled, or after the scene is reset. This force is applied into the calculations before the mass of the planet is factored. This means the same force can be applied to two different planets of different masses

Code Structure

Utilities

Large amounts of functionality within the program is broken down and encapsulated within utility classes:

- `CameraUtil`: Used to manipulate the view matrix
- `ConfigUtil`: Used to read from the config.dat file
- `ConsoleUtil`: Used to read from and write to the console window
- `FileUtil`: Used for many different actions for interacting with files, folders, and file paths
- `GLEWUtil`: Used to interact with the GLEW library
- `GLFWUtil`: Used to interact with the GLFW library
- `InputManager`: Used to register mouse and keyboard interactions as well as execute callback functions when those actions occur
- `ModelLoaderFactory`: Used to return a model loader based on the file path it is given
- `MVPBuilder`: Used for a few different calculations involving a model's MVP. This also keeps track of a model's position, rotation, and scale
- `PlanetFactory`: Used to load in all the data contained within the planets.dat (Solar%20System/planets.dat) file into `Planet` objects
- `ShaderProgramBuilder`: Used to compile and register shader programs
- `TimeUtil`: Used to keep track of the current delta time

The majority of utility classes are instantiated once within the main method and are then passed around wherever they are needed via constructor dependency injection. Many of these classes rely on others.

Core

The core program is broken down into the following structure:

- Scene
    - Shader Program
    - Gravity logic
    - Planets[]
        - Model
            - Materials[]
            - MVP
            - Objects[]
                - Meshes[]

Details:

- There is a single `Scene` within the program; this represents what the user sees

- The `Shader Program` sits at the `Scene` level and remains constant after init
- The different `Planet`s all sit within the `Scene`. While the word 'planet' is always used, this technically also refers to the sun and the background.
- Once per game tick, when the `Scene` is updated, it manipulates the `Planet`s using its gravity implementation and then also calls an update method on all of the `Planet`s
- A `Planet` stores information used in physics calculations as well as stores a single `Model`
- A `Model` holds all the different `Material`s that any of its child `Object`s might use
- A `Model` has an MVP, and it sets this as the current one each game tick before it then calls an update method on all of it's `Objects`
- `Objects` only act as a container for `Mesh`es. They exist to mirror the structure found within different model file types. When an `Object` is updated, it must update all of it's `Mesh`es
- A `Mesh` contains the data required for OpenGL to render something. It also has references to a single `Material` and the `Shader program`

Project Motivation

While having never worked directly with one in C++ and OpenGL, I have assumed that implementing gravity using a physics engine library would be reasonably straightforward. I wanted to see how easy it would be to create a realistic gravity system without using such a library. It was critical to the development of this system that it worked with real measurements, and as such, it works with kilograms and metres. This system only works with small units of measurements due to the orbit sizes needed for larger numbers. Still, it stands to reason that if were to configure it to use accurate planetary masses and distances then it would create stable orbits.

While all the measurements are accurate, the 'physical' sizes of the objects are not. By this, I mean that the mass of a planet does not affect its size, and its size does not affect the mass or movements. This allows for a user to configure the program to use any model file without it affecting the physics. Any model can also be rendered in any proportional size.

This project is originally based off of my model loading project found on Github here (https://github.com/tobysmith568/SOFT-356_Model_Loader).