PhysicsSim - a configurable gravity sandbox

Arseny Uskov

PhysicsSim is a customisable simulation of gravitational interactions between 'planets' depicted on a 2D plane. The simulation is controlled via a series of shortcut keys (remappable with a config file) the cursor and a scrollwheel (or gestures imitating scrolling). It uses object-oriented programming techniques to remain efficient, despite having O(n^2) complexity due to the calculation of gravitational forces on every pair of planets. It runs at a fixed time step (60 Hz) and doesn't slow down on a modern processor, even with a large number of planets. The simulation itself was constructed with care to improve efficiency and readability of the code, but no optimisation is done on the gravitational calculation, although this is possible by using clustering of planets, for example.

Usage and customisation

`config.xml` is a user-editable configuration file containing two types of elements - bindings and globals. Bindings are used to change the simulation controls; globals are used to change the underlying properties of the simulation. The config file is pre-populated with the default value for each attribute, which can be changed by the user at will. If certain options are missing, the simulator will change their values to these defaults automatically. However, if an invalid configuration is read, the simulator will not launch, which is why it is advisable to change the configuration options one by one, making configuration errors easier to catch.

Controls

Action	Key (default)	Description
Pause	NumPad0	Pause the simulation
New	NumPad1	Advance planet creation
Grid	NumPad2	Toggle between three grid modes
Clear	NumPad3	Clear existing planets/cancel planet creation
Debug	NumPad4	Toggle debug view
Trail	NumPad5	Toggle visibility of planet trails

The above is a the list of customisable keyboard shortcuts used to control the simulation; `config.xml` contains these default key bindings in a format understood by the simulator. Any shortcut key can be remapped by replacing the `key` attribute with a valid keycode: a single capital letter denotes the corresponding letter on the keyboard, D* denotes a number in the top row (where * is replaced by the intended digit) and NumPad* denotes a number on the number pad (where * is replaced by the intended digit); an exhaustive list of keycodes can be found here.

One must tap the 'New' shortcut three times to introduce a new planet into the simulation. The first tap introduces it to the canvas. Now, the position (controllable by moving he mouse to the desired location) and size (controllable with the mousewheel or a scrolling gesture on a laptop trackpad) can be set, before locking these properties with another tap of the same shortcut. The initial velocity of the planet can now be set by moving the mouse to the desired location relative to the position of the planet (the indicator is an exact representation of the distance covered in one second) before locking this final property with a final tap of the 'New' shortcut key.

The toggleable values provide additional functionality: the ability to pause the simulation, toggling between two grid sizes (the mouse position is automatically snapped to the nearest horizontal and vertical gridlines, if they are visible), toggling the visibility of planet trails and toggling a debug view which presents additional information about the planet being created as well as showing the real-time velocity of any existing planet.

Globals

The `config.xml` file also permits the user to change a number of 'global' values used primarily to change the properties of the gravitational forces in the simulation. Each `value` attribute of each of the globals *must* be an integer.

`Gglobal` (gravitational global) stores the value of the gravitational constant (G) used to calculate gravitational forces.

`Tglobal` (trail global) represents the number of seconds that are recorded as part of each planet's trail; trails are hidden by default and are toggleable by the 'Trail' shortcut. Larger values of this global are more resource-intensive; it is the only configuration option which impacts performance.

The mass of each planet is calculated according to the following `PMglobal` (planet mass global) values, making a crucial yet configurable link between the size of a planet and its mass:

```
mass = (radius ^ index) * coefficient + constant
```

Intended usage

The simulation is intended to be used as a demonstration of gravitational interactions between bodies and *not* as a general-purpose physics simulation. For this reason there is only basic collision detection, which aims to maintain the course of the 'colliding' planets by ignoring the gravitational force between any two planets which intersect one another. This feature only exists so that the simulation is not ruined by the centres of two planets approaching each other in such a way as to produce a huge force on both planets, sending them out of view. This workaround allows two planets to briefly touch without interrupting the general course of either of their orbits. The event of a planet spending a large portion of its orbit in this state falls outside the realm of the simulator's intended use. In this case, the simulator stops producing realistic results and must be reset by using the 'Clear' shortcut.

The simulator allows easy visualisation of systems such as binary stars and also more conventional systems where one or more small planets orbits a single star. It can also serve to demonstrate the chaotic nature of such systems by simulating three bodies (all of similar mass) being launched into orbit, where even a small change in starting conditions can result in a large difference in the state of the simulation at a later stage.

There is little UI to introduce the user to the simulator and its controls/operation: developing a full interface would take a disproportionate amount of programming time to the amount of functionality it provides to the user, replacing the brief user guide above. After having understood the controls and customisation options, the minimal UI shown at the corners of the screen is enough for the user to confidently operate the simulator at its full potential.

Development

The role of object-oriented programming (OOP)

OOP, a major component of the design of the simulation, is the practice of dealing with a collection of classes and objects, where each is responsible for one small aspect of the whole, as opposed to a structure where the whole program is treated as one long series of functions. Programs written with OOP in mind usually have a number of defining characteristics, describing the way in which objects are to communicate.

The fact that the simulation concerns an arbitrary number of discrete objects (planets) makes it easy to conceptualise each planet as an instance of the planet class, with its own properties such as mass, radius, position and velocity. Furthermore, each planet must be able to update its own location and velocity on every rendered frame, so it was intuitive to introduce a method of that class to handle updating these fields on every frame for a given planet. The implementation of OOP in C# allow each instance of the Planet class to access data of all other instances, so any given planet can calculate the force on itself from all the other individual planets.

OOP is also used elsewhere in the simulation - to differentiate between the two available modes, for example: an idle mode and a mode for planet creation. Each one is a class, implementing the `IMode` interface, which allows the swapping out of these classes inside a "current mode" variable. Both have an `Update()` method (as per the interface they implement), allowing only certain actions to be performed in each mode, without a lengthy if else statement to determine the current mode and perform the appropriate function - this would have to be run every frame.

A similar technique is used to swap between the three 'modes' of possible mouse input. Because the co-ordinates of the mouse (as visible to the Planet class) have to be rounded depending on the current grid setting, there is a separate class for each of the grid levels, all implementing a common `IMouseInput` interface. In this way, the correct methods can be run to both get the current mouse co-ordinates (whether or not they are snapped to the nearest gridlines) and to draw the grid, without wasting an if statement to check the current grid level on every time step.

The project benefitted greatly from an OOP approach, ranging from the implementation of retrieving keyboard and mouse input to the concept of a Planet object itself. Having previously never used OOP principles (or even classes), this approach improved readability and greatly decreased the time it took to find the location where a certain feature is implemented (compared to previous projects), so that the simulator's features could be added faster - just a few benefits of having a well-defined structure.

One must recognise that not all concepts relating to object-oriented design were strictly followed. For example, few fields are encapsulated in each class, meaning that a lot of them are necessary to be made accessible to other classes for various purposes. UI doesn't play a huge role here, so the on-screen elements are drawn directly in the Simulator class (instead of their own class), requiring rather low-level access to the Planet class for details such as a planet's radius and position in case the debug switch is turned on by the user. One could move the drawing of these two particular elements to the Planet class, but this would separate these lines from the rest of the code responsible for drawing more general debug information like the number of planets and whether the simulator is reaching its framerate goal. A whole separate class could be created

solely for the purpose of drawing UI elements, but that wouldn't solve the issue of encapsulation as this class would still have to have the same amount of low-level access to the Planet class for displaying this debug information. A few fields are in fact properly encapsulated, such as those responsible for storing the trail of the planet and its current velocity - the methods which require those fields are already in the Planet class. Ultimately, the process of drawing UI is rather simple and compact, so this particular issue is only that of design, not of readability or functionality.

On the other hand, the Planet class also requires access to the main spriteBatch (`Simulator.spriteBatch` - this is used in every function call where a shape is to be drawn to the canvas) for drawing its planet on every frame. Hence, both classes must have public, static elements to be able to perform their duties in the correct method, according to their purpose, as opposed to what the scope/hierarchy allows. At the very least this makes the simulator more maintainable, by allowing similar functionality to be achieved in the same method, regardless of scope.

Use of version control

Git was used in the form of a GitHub plugin to maintain a record of changes and the additions of various features. Although only a small fraction of git's functionality was used in this one-man project, it allowed synchronisation between multiple devices without losing progress, and most importantly the ability to rollback changes as needed. On two occasions the functionality of the simulator was crippled by a lack of testing after the development of multiple features in quick succession, and reverting to a previous commit and inspecting the changes one by one (this time building the application to test it throughout this process) allowed easy debugging of the error. The power of git would have made much more of an impact on a group project or one where contributions are taken from the community, but its usefulness was not missed here.

Future improvements

A number of features could still be thought of as necessary in a simulation like this but aren't included here. For example, a number of crucial graphics function calls could be configured to allow the "camera" to be moved and the "zoom level" changed, so that orbits going out of frame could be still be easily followed.

As a final point, one must acknowledge the difficulty of simulating celestial bodies and the gravitational forces between them (to scale): the gravitational constant is orders of magnitude smaller than in the default configuration and real celestial bodies are separated by distances vastly greater than their radii, completely unlike the values set in the default configuration. The simulator can be adjusted to near those values somewhat, but the effect of gravity is less noticeable at this scale. In the end, a compromise must be made between realistic parameters and a visually appealing demonstration of the gravitational forces at play.