GRAVITAS

EJC March 18, 2009

Introduction

Gravitas is a fully scriptable, real-time 3D gravity simulator. Scripts written in the integrated development environment (IDE) are interpreted by the simulator and allow users to create any imaginable planetary setup.

Once inside the simulated system, the user can move around in 3D space to observe the gravitational interactions of the celestial bodies. With the aid of the head-up display (HUD), users can track bodies and view live information about the simulation as it progresses.

Gravitas makes use of the powerful yet simple-to-use Lua scripting language, which offers the perfect introduction to programming for beginners, and extensive features for more experienced users. The rendering of simulations is handled by OpenGL, the industry standard interactive 3D graphics environment.

Gravitas is an Open Source application which means that you can download the full source code for the simulator from the project website. It is programmed and compiled using FreeBASIC.

Usage

The first step in creating a gravity simulation with Gravitas is to make a Lua script which describes the initial setup of the system. This script specifies the number of bodies as well as their positions, velocities, masses and radii, and also contains general commands which define various aspects of the simulation.

Once this script is complete, you can run the simulation by either pressing the 'Run' button on the toolbar or the 'F5' key. This will open a full screen window in which the simulation will take place. If there are any problems with your script, the simulation will stop and an error message will be displayed. As soon

as the script has been processed (this will normally take less than a second), the simulation will start.

During the simulation, your viewpoint (the camera) can be moved around in space using the keyboard and mouse (see Controls Reference). If at any point during the simulation you want to reset the camera to its initial position, press the 'Home' key. The simulation can be paused by pressing 'F1', and many options can be toggled on or off through other keyboard actions.

To exit the simulation, press 'Esc' on the keyboard. This will close the full screen window and return to the Gravitas main editor. You are now in a position to continue editing your script.

There are many example scripts included with Gravitas, which can be opened from the 'Welcome' screen of the program when it is first run. Alternatively, they can be found in 'Examples' under the File menu. To learn more about creating your own Lua scripts, open some of the tutorial files which can also be found on the 'Welcome' screen or under the Help menu.

Simulation Concepts

Gravitas applies Newton's Law of Gravitational Attraction to all of the bodies in a simulation.

$$F = G\frac{m_1 m_2}{r^2}$$

It calculates the overall force on each body and hence the acceleration (by Newton's Second Law). From this, it works out what the velocity of the body will be a very short time later (this time is called the step length of the simulation), and also the position of the body after this time, given that it is travelling at the velocity just calculated.

This process is then repeated, with the force on each body re-calculated at the new positions. The assumption made during the simulation is that the force acting on a body during the step length is constant (so that its acceleration is constant also). Clearly, the larger the step length, the more uncertainty there is in the simulation as the assumption above is less valid. It is also true to say that the size of the step length affects the speed of the simulation – the smaller the step length the more calculations your computer has to make.

The gravity calculations are summarised in this pseudo-code below:

```
Repeat

      For i From 1 to NumBodies

            Body(i).Force = 0

            For j From 1 to NumBodies
                  If j <> i then
                        D = Body(i).Position-Body(j).Position
                        F = (G * Body(i).Mass * Body(j).Mass)/(D^2)
                        Body(i).Force += F
                  End If
            Next j

            Body(i).Velocity += StepLength * (Body(i).Force / Body(i).Mass)
            Body(i).Position += StepLength * Body(i).Velocity

      Next i

Until GravityStop = 1
```

In the Gravitas source code, the calculations are optimised so that as few operations have to be computed as possible. For example, you can see above that the overall force is eventually divided by the mass to give acceleration, so it is quicker to just calculate the acceleration due to gravity in the first place (ie. by not multiplying by `Body(i).Mass`).

Controls Reference

This section lists the simulation controls on the keyboard/mouse.

Camera Controls

Mouse	Look around
W	Forwards thrust
S	Reverse thrust

A	Strafe left
D	Strafe right
R	Rise
F	Fall
Q	Roll left
E	Roll right
SPACE	Brake
HOME	Reset camera

Simulation Controls

F1	Pause simulation
F2	Hide/show background sphere
F3	Enable/disable nice rendering
F4	Hide/show HUD
F12	Save image[1]
PLUS (+)	Increase step length
MINUS (-)	Decrease step length
LEFT / RIGHT	Change tracked body
RETURN	Highlight tracked body
T	Show/hide history
ESC	Exit simulation

Lua Scripting

There are plenty of resources online detailing programming with Lua, but this section serves as a very brief introduction to some of the commands you may encounter while using Gravitas.

In Lua, just like any other programming language, you can store information in variables. Within the context of Gravitas, these variables will generally contain numbers. For example:

[1] Images are saved to the 'My Pictures' directory in a folder named 'Gravitas'.

```
velocity = 0.5          --Sets the variable 'velocity' to 0.5
result = velocity+1     --'result' contains the number 1.5
number = result/3       --'number' now contains 0.5
```

There are many predefined functions that are used to send information to the simulator – these are detailed in the Functions Reference below. For example, you would use the `addbody` function to add a body to the simulation, and so forth. There are also many functions which are not specifically related to Gravitas, but may be useful when calculating the positions or velocities of bodies in your initial setup, such as mathematical functions. For example:

```
theta = (math.random()*2-1)*math.pi
x = 200*math.cos(theta)
y = 200*math.sin(theta)
```

And a Gravitas function:

```
mass = 10000000000000
vx, vy, vz = circularorbit(mass, 100, 100, 100, 1, 1)
--The function above returns 3 variables which are stored in vx, vy and
--vz respectively.
```

There are also several different control structures that you can use within a Lua script. Examples of these are conditional statements and loops:

```
particles = 200
for i = 1, particles, 1 do
      addbody(0, 0, 2*i, 0, 0, 0, 10000, 1)
end

mass = math.random()*100000000000000
id = addbody(0, 0, 0, 0, 0, 0, mass, 10*(mass/100000000000000))
if mass < 10000000000000 then
      setcolour(id, 255, 255, 255)
else
      setcolour(id, 255, 0, 0)
end
```

There are many example scripts as well as tutorials which are part of Gravitas that will guide you through creating your own script; these also demonstrate how to use the functions detailed in the Functions Reference below.

Note: Lua is a case-sensitive programming language, so the variable `velocity` will not be the same as `Velocity`. All of the Gravitas functions are lower case.

Functions Reference

`settitle(text)`

Sets the text that is displayed at the top of the simulator screen.

`setinterval(interval)`

Sets the step length (in seconds) for the iterations of the gravity calculations in the simulation. The smaller this value is, the more accurate (and slower) the simulation will be. The step length can be altered within the simulation by holding the plus and minus keys.

`setcamera(x, y, z)`

Sets the starting position in Cartesian coordinates of the camera in space. The default is (`0, 0, 500`).

`setdirection(x, y, z)`

Sets the direction in which the camera is initially facing. For example, a vector of (`1, 0, 0`) sets the camera looking in the x direction. By default, the direction vector is (`0, 0, -1`).

Note: (`x, y, z`) must be a non-zero vector.

`setscale(scale)`

Sets the scale of the simulation. For example, `setscale(1000)` results in 1 unit in 3D space being equivalent to 1000 metres. By default, the scale is 1 unit = 1 metre.

`sethistorylog(number)`

Sets history logging on. This plots the position of a body every `number` iterations of the gravity calculation so that a virtual trail will be drawn in

space. Pressing the 'T' key during a simulation in which history logging is enabled will toggle the display of these trails on and off.

Note: The ninth parameter of `addbody` must be set to 1 for each body you wish to record history for.

`enablecollisions(1 or 0)`

If the parameter of this function is set to 1, collisions will be enabled (default). If set to 0, collisions between bodies will be disabled and they will not merge if they touch (this will potentially increase the simulation speed as well).

`enablesphere(1 or 0)`

If the parameter of this function is set to 1, the large sphere grid in the background will be rendered (default). If set to 0, it will not be rendered. This can be toggled on and off in the simulation by pressing 'F2'.

`startpaused()`

Starts the simulation in a paused state. Press the 'F1' key to un-pause.

`addbody(x, y, z, vx, vy, vz, mass, radius, history)`

This command adds a body to the simulation. You need to specify its initial Cartesian coordinates (`x, y, z`), the components of its velocity (`vx, vy, vz`), its `mass` and its `radius`. The final (optional) parameter, `history (1 or 0)`, specifies whether or not the body should have its history stored (only valid when `sethistorylog` has been called). The function returns a unique ID for the body, which can be used to reference it in other functions.

`setcolour(id, r, g, b)`

Sets the colour of a body. `id` is a unique body identifier returned from the `addbody` function, and (`r, g, b`) are red, green, and blue colour components (from 0 to 255). For example `setcolour(1, 1, 0, 0)` sets the colour of the body with `id = 1` to red.

Note: By calling this function the default colouring of ALL the bodies is cancelled and will be rendered as white unless this function is specifically called with their ID.

`settext(id, text)`

Sets the name of a body, as displayed on the HUD (head-up display). By default the name is "[Body ID]". For example, the command `settext(1, "Sun")` will set the name of the first body created to "Sun".

`settrack(id)`

Specifies which body the HUD (head-up display) shows statistics for. By default this is the first body created (`id = 1`). The tracked planet can be changed within the simulation by pressing the left or right arrow keys.

`circularvelocity(mass, radius)`

Returns the velocity required for a body at a distance `radius` from another body of mass `mass` to execute a perfect circular orbit. This assumes that the orbiting body is much less massive than the centre body.

Note: Use the `circularorbit` function to return the 3D components of velocity for a specified circular orbit.

`circularorbit(mass, x, y, z, zn, yn)`

Returns three variables (`vx, vy, vz`), the three components of velocity for a circular orbit about a centre body of mass `mass`. The vector (`x, y, z`) is the position vector of the orbiting body relative to the centre body.

As there are infinitely many possible directions in which a circular orbit can take place in 3D (around all circumferences of a sphere), the remaining two parameters specify the x and y components of this direction. The function will automatically calculate a z component given that the velocity vector and position vector must be perpendicular, and hence determine the velocity components in that direction.

If the specified values for `zn` and `yn` are inconsistent with the position vector (ie. position vector (`0, 0, 100`) and velocity components (`0, 0`)),

the function will adjust the velocity components so that a valid velocity vector can be returned.

Troubleshooting

Script Errors

If your script gives an error when you attempt to run the simulation, then you are using some of the Lua syntax incorrectly, or may have misspelt one of the function names.

Remember that Lua is case-sensitive, so it is probably best to write everything in lower case to avoid confusion. If a Gravitas function doesn't show up with green highlighting, then you have spelt it incorrectly.

If you have problems with a specific function or Lua command structure, take a look at some of the examples which implement similar script.

Simulation Errors

If the simulation doesn't render correctly, or all you see is a black screen, then it is likely that your computer has an out-dated graphics card which doesn't support OpenGL fully. To fix this problem update your graphics card driver software (which can be downloaded for free online), or move to a newer computer!

On some graphics cards, OpenGL is only partially supported and as such certain parts of the simulation may not display correctly, for example the shading. In order to fix this, unselect 'Smooth Rendering' under the 'Simulate' menu.

General Problems

As Gravitas is fairly processor intensive, it may be that it runs quite slowly on your computer. If this is the case, it is suggested that you close all other applications while running Gravitas. Alternatively you may decide to sacrifice accuracy by increasing the step length in your script which will result in a faster simulation.

Appendix A – Circular Orbit Calculations

This section explains how the functions `circularvelocity` and `circularorbit` work. For a body of mass `m` to execute a circular orbit or radius `r` around a larger body of mass `M`:

$$\frac{GMm}{r^2} = \frac{mv^2}{r} \Rightarrow v = \sqrt{\frac{GM}{r}}$$

The function `circularvelocity` simply carries out this calculation and returns the value of `v`. The second function takes this velocity and converts it into a 3D vector based on the values of `xn` and `yn` specified by the user; these are the x and y components of a vector in the direction of the velocity. As the position vector of the body and its velocity vector must be perpendicular:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} \cdot \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} = 0$$

It is therefore also true to say that:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} \cdot \begin{bmatrix} x_n \\ y_n \\ z_n \end{bmatrix} = 0$$

$$x \times x_n + y \times y_n + z \times z_n = 0$$

This statement can be rearranged to give an expression for `zn`, the z component of the vector in the same direction as the velocity.

$$z_n = -\frac{x \times x_n + y \times y_n}{z}$$

From here it is now a matter of simply normalising the vector (`zn, yn, zn`) and multiplying it by the calculated velocity to give the velocity vector.

$$\begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} = \frac{v}{\sqrt{x_n^2 + y_n^2 + z_n^2}} \begin{bmatrix} x_n \\ y_n \\ z_n \end{bmatrix}$$

The three components of this vector, `vx`, `vy`, and `vz`, are returned from the function `circularorbit` respectively.

Note: You will have realised that there are a few situations in which the function will fail; for example, if `z = 0`. In this specific case, Gravitas will make a small adjustment to the value of z so that it is just above zero. In other situations where the specified values of `xn` and `yn` are inconsistent with the position vector, Gravitas will choose another suitable direction vector so that a valid velocity vector can be returned.