PHYS 393 Mini-Project Report

Perrin Waldock

February 24, 2018

Contents

1 Introduction

This program loosely simulates the behaviour of a rocket in outer space, by allowing the user to fly around a binary planet system and shoot projectiles from their craft. The program simulates the effects of gravity, elastic and inelastic collisions, and uses momentum to calculate the force of the thruster on the rocket.It does not take rotational momentum into account at this point (3D elastic collisions with rotational momentum can be complicated to solve), but does allow the user to rotate the rocket.

2 Physics

2.1 Motion

2.2 Gravity

All bodies in the system experience a mutual gravitational attraction. I calculated the gravitational force between two bodies using eq. (1):

$$\vec{F_{12}} = \frac{-Gm_1m_2}{r_{12}^2}\hat{r}_{12} \tag{1}$$

where $\vec{F_{12}}$ is the force that body 1 exerts on body 2, $\vec{F_{21}}$ is the force that body 2 exerts on body 1, G is the gravitational constant ($6.67408 \times 10^{-11} \frac{m^3}{kg^1 s^2}$), m_1 is the mass of the first body, m_2 is the mass of the second body, and $\vec{r}_{12} = \vec{r_1} - \vec{r_2}$ is a vector pointing from the center of mass of body 1 to the center of mass of body 2.

To find the force body 2 exerts on body 1, I applied Newton's laws to get $\vec{F_{21}} = -\vec{F_{12}}$.

2.3 Collisions

There are two types of collisions that bodies in the simulation can undergo: Completely elastic collisions and completely inelastic collisions. 3D partially inelastic collisions were beyond the scope of this project.

2.3.1 Elastic Collisions

In elastic collisions, energy and momentum are both conserved. I used eq. (2) to calculate the new velocity of a body after an elastic collision:

$$\vec{v_1}' = \vec{v_1} - \frac{2m_2(\vec{v_1} - \vec{v_2}) \cdot \hat{r}_{12}}{m1 + m2}\hat{r}_{12} \tag{2}$$

where $\vec{v_1}'$ is the velocity of the first body after the collision, $\vec{v_1}$ is the velocity of the first body before the collision, $\vec{v_2}$ is the velocity of the second body before the collision, m_1 is the mass of the first body, m_2 is the mass of the second body, and $\hat{r_{12}}$ is the same as in . To find $\vec{v_2}'$, I used the same equation but with 1 and 2 swapped.

2.3.2 Inelastic Collisions

In completely inelastic collisions, the bodies stick together. To find the mass and velocity of the new, combined body, I added the masses to find the mass of the new objects, and used eq. (3) to calculate the new body's velocity.

$$\vec{v} = \frac{(m_1\vec{v_1}) + (m_2\vec{v_2})}{m_1 + m_2} \tag{3}$$

where $\vec{v}$ is the velocity of the new body, m_1 and v_1 are the mass and velocity of the first body, and m_2 and v_2 are the mass and velocity of the second body.

2.4 Rocket

Rockets fly forward because they expel mass at a high velocity behind them. As such, the equation for the force a rocket produces is eq. (4).

$$\vec{F}_{rocket} = \frac{dm}{dt}\vec{v}_{exhaust} \tag{4}$$

where $\vec{F}_{rocket}$ is the force the rocket produces, $\frac{dm}{dt}$ is the rate at which the rocket is expelling fuel, and $\vec{v}_{exhaust}$ is the relative velocity of the exhaust of the rocket as compared to the rocket's velocity.

2.5 Projectiles

When the craft launches a projectile, momentum must be conserved. Consequently, the velocity of the craft must change. I used eq. (5) to calculate the craft's new velocity.

$$\vec{v'} = \frac{m\vec{v} - m_p\vec{v_p}}{m} \tag{5}$$

where $\vec{v'}$ is the velocity of the craft after firing, $\vec{v}$ is the velocity of the craft before firing, m is the mass of the craft. $\vec{v_p}$ is the velocity of the projectile after firing, and m_p is the mass of the projectile.

3 Control Panel

On the right side of the window, there should be a series of sliders and buttons that control the craft. They are broken into several major sections.

3.1 Statistics

The top the panel shows useful information about the rocket:

- **Orientation** is a normalized vector showing which direction the rocket is facing. All vectors are displayed as <x y z>, where x, y, and z are the components of the vector.
- **Position** is a vector containing the rocket's position.

- **Velocity** is a vector containing the rocket's velocity.
- **Speed** is the rocket's speed (which is often easier to read than the velocity).
- **Fuel Remaining** shows how much fuel the rocket has left to burn. When the rocket runs out of fuel, the thrusters will no longer fire.
- **Ammo Remaining** shows the number of projectiles that the craft can fire. This limits the number of bodies that the program must simulate.
- **Time Passed** shows the amount of virtual time that has passed in the day:hour:minute:second format.
- **Time Scale** shows the number of virtual seconds that should pass to ever real second.
- **Mass** shows the total mass of the craft, fuel included.
- **Distance From Origin** shows the distance of the craft from the starting position of the non cloud-covered planet (the origin of the coordinate system used in the simulation).

3.2 Simulation Controls

These controls are located in the middle left of the control panel. They adjust how the simulation runs.

- Clicking **run** will cause the simulation to to run; clicking **pause** will cause it to pause. This can also be toggled by using "p" on your keyboard.
- Clicking **Show Trails** will show the trajectories of all of the objects in the system; **Hide Trails** will hide these trajectories.
- **Time Step Size** changes the number of seconds between simulation steps. Smaller time steps offer more accurate results but cause the simulation to run slower.
- **Frames Per Second** changes the maximum frame rate of the simulation. If your computer is not powerful enough, you may not achieve that frame rate.
- **Reset Simulation** clears all projectiles and returns the planets and craft to their starting position without changing other settings. This feature is slightly buggy; I recommend restarting the program unless you need to keey certain settings in place

3.3 Camera Controls

These settings control the camera. I chose to make the camera follow the craft as the craft was too hard to control otherwise.

- **Zoom Level** changes where the 'camera' appears to be positioned away from the craft.
- **Vertical Camera Angle** changes what vertical angle the camera looks at the craft at; 0 means that it looks on the same horizontal plane as the craft. This can also be adjusted using the 'w' and 's' keys on your keyboard.
- **Horizontal Camera Angle** changes what horizontal angle the camera looks at the craft at; 0 means that it looks on the same vertical plane as the craft. This can also be adjusted using the 'a' and 'd' keys on your keyboard.

3.4 Craft Controls

These controls are in the middle and middle-right of the control panel. They change how the craft behaves.

- **Turn Up** rotates the craft upwards. The same effect can be achieved (with a more fine resolution) by pressing and/or holding the up arrow key on your keyboard.
- **Turn Down** rotates the craft downwards. The same effect can be achieved (with a more fine resolution) by pressing and/or holding the down arrow key on your keyboard.
- **Turn Left** rotates the craft left. The same effect can be achieved (with a more fine resolution) by pressing and/or holding the left arrow key on your keyboard.
- **Turn Right** rotates the craft right. The same effect can be achieved (with a more fine resolution) by pressing and/or holding the left arrow key on your keyboard.
- **Fuel Burn Rate** adjusts how quickly the craft burns through its fuel. A higher burn rate goes through fuel more quickly, but also provides greater force.
- **Refuel** magically adds more fuel to the fuel tank of the craft; this both increases the mass of the rocket and allows it to fly for longer.
- **Recentre** magically resets the craft to the origin.
- **Craft Colour** allows you to change the craft colour; it can be useful for tracking where the craft was at certain points of time when looking at its trail.
- **Mass of Craft When Empty** adjusts the empty mass of the ship.
- **Exhaust Velocity** adjusts the exhaust speed of the ship. These values are larger than real-life ships to allow for faster movement; to be realistic, try 2.4 or 4 km/s.

3.5 Projectile Controls

These controls change the settings for the projectiles fired from the craft.

- **Fire** fires a projectile from the craft. This can also be achieved by pressing "f" on your keyboard.
- **Reload** increases the craft's ammunition. Be wary of doing this as it may cause the simulation to need to track too many things at once.
- **Clear** should remove all projectiles from the simulation. This is good to do if your simulation slows down after firing a lot of projectiles.
- **Projectile Colour** allows you to change the colour of the projectile; it can be useful for tracking different projectiles.
- **Projectile Radius** allows you to change the radius of the projectile; this affects how it behaves in elastic collisions.
- **Projectile Mass** allows you to change the mass of the projectile; this affects the craft when firing the projectile as well as the the result of any collisions.

- **Projectile Launch Speed** allows you to change the initial speed of the projectile relative to the craft; this affects the craft when firing the projectile as well as the the result of any collisions.
- **Projectile Collision Type** changes the type of collisions that the particle can undergo (completely elastic or completely inelastic).

4 Documentation

4.1 High-Level Design

This program has two main parts; the setup and the simulation.

4.1.1 Setup

The setup creates the GUI and the objects in the simulation. It executes once at the start of the program.

First, it constructs the GUI. Next, it creates its starting objects (which currently consist of the craft and two planets). Following that, it defines the handler functions for the GUI and binds them to the buttons. I could not define the handler functions as they needed to modify the objects created after setting up the GUI.

4.1.2 Simulation

The simulation actually simulates the motion of the bodies in the system. It is all wrapped inside of a giant while-loop.

First, the program updates parameters such as the camera angle, craft orientation, frame rate, etc from the GUI. Most importantly, it checks whether or not the simulation is paused or running. If paused, it returns to the start of the simulation loop and updates the parameters again. If it isn't paused, it continues on to the physics simulation.

The physics engine first increments the time counter that tracks how many seconds have passed since the start of the simulation. Next, it sees if any objects are touching and handles collisions if they are (see section 4.4.1 for more details). Next, it calculates the gravitational attraction between each of the objects and adds it to a list of forces acting on each object. Finally, it adds the force of the rocket to the force acting on the craft, and makes each object move according to force acting on them.

4.2 Variables and Parameters

4.2.1 Constants

- gravitationalConstant = G, the gravitational constant in eq. (1).
- massEarth is the mass used for the mass of the planets simulated.
- radiusEarth is the radius used for the radius of the planets.
- mCraftI is the starting mass of the craft you control. It and massSlider affect the mass of the craft, which can show up as an m in any of the above equations.

- lCraftI is the animated length of the craft you control. It is not the actual simulated length.
- maxBurnRate is the maximum rate that the craft can expel rocket fuel. It is the maximum value on thrustSlider.
- maxExhaustSpeed is the maximum speed at which the craft can expel rocket fuel. It is the maximum value on exhaustSlider.
- MAX_SIMULATION_TIME is the number of seconds that the simulation runs for before ending. It is used as the compare condition in the while loop.
- L is the height of the window.
- widgetL is the width of a slider in the panel.
- border is the thickness of the border around the window.
- zoomMax is the maximum zoom level allowed.
- colourChoices is a list of allowed colours.

4.2.2 Variables

There are few standard variables in this program, as most values are read directly from the slider objects.

- dt is the number of seconds between updates in the simulation. Its value is read from the timeSlider object.
- frameRate is the maximum number of frames that the simulation shows each second. It is updated by reading from the rateSlider object.
- t is the number of virtual seconds that have elapsed in the simulation, and is used as the value compared to MAX_SIMULATION_TIME in the while loop.
- trails is a boolean that keeps track of whether or not the animation should show the path of all of the bodies in the system. It is generally updated by the showTrails object.

4.2.3 Objects

I used a number of sliders to represent parameters in the program, as well as a few other objects

- objects is the list of all bodies simulated in the system.
- w1 is the window object.
- display1 is the display that shows the 3D animation inside of w1.
- p1 is the panel next to the 3D display.
- stats is a list of text boxes that show information about the craft and the simulation. See section 3.1 for more information.
- startStop is a radiobox that determines whether the animation should be paused or run.

- showTrails is a radiobox that determines whether or not the animation should show trails from objects or not.
- resetButton is a button tied to the reset function; it resets the simulation.
- zoomSlider adjusts the zoom of the 'camera'.
- vSlider adjusts the vertical tilt of the 'camera' from the craft's forward direction.
- hSlider adjusts the horizontal tilt of the 'camera' from the craft's forward direction.
- upButton, downButton, leftButton, and rightButton rotate the craft their respective directions.
- thrustSlider adjusts the rate at which the craft burns its fuel. Its value shows up as $\frac{dm}{dt}$ in eq. (4).
- exhaustSlider adjusts the speed at which the craft expels its exhaust behind it. Its value is the magnitude of $\vec{v}_{exhaust}$ in eq. (4).
- massSlider is a slider that adjusts the empty mass of the craft, which shows up in any equation involving forces on the craft.
- refuelButton is a button tied to the refuel function, which adds fuel to the craft, as the craft tends to burn through its fuel supply very quickly.
- recentreButton is a button tied to the recentre function, which repositions the craft at the origin.
- craftColourBox is a radiobox that adjusts the colour of craft.
- triggerButton is a button tied to the fire function, which launches a projectile from the craft.
- reloadButton is a button tied to the reload function, which increases the number of projectiles that the craft is carrying.
- clearButton is a button tied to the clear function, which clears all projectiles from the simulation.
- projectileColourBox is a radiobox that changes the colour of the projectiles that the craft launches.
- projectileRadiusSlider is a slider that adjusts the radius of the projectiles that the craft launches.
- projectileMassSlider is a slider that adjusts the mass of the projectiles that the craft launches.
- projectileSpeedSlider is a slider that adjusts the relative speed of the projectiles to the craft when launched.
- collisionTypeBox adjusts the type of collisions that future projectiles that the craft launches will undergo.

4.3 Classes and Functions

4.3.1 spaceshipSimulation

This file could be relatively easily converted into a class I wanted to set up a LAN game (or just make everything slightly more object-oriented), but for now is just a file with functions. It acts similarly to the main function in a C program and contains the main body of the code. However, it also contains a few useful functions:

- makeStatsString generates a list of strings that display information about the craft and the simulation.
- makeStartObjects creates a list of objects that the simulation starts with. The first object should be the craft that the simulation controls, but after that, any number of objects can be added (provided that your computer can simulate that many).
- translateColour translates a number from a radio box into a colour.
- refill is an event handler function that increases the amount of fuel in the craft.
- reload is an event handler function that increases the number of projectiles that the craft can fire.
- recentre is an event handler function that sets the craft to the origin of the system.
- clear is an event handler function that deletes all projectiles from the simulation. It is useful when the simulation slows down or when objects disappear because it frees up some RAM.
- fire is an event handler function that generates a projectile and adds it to the objects variable, effectively making the craft fire a projectile.
- turnLeft, turnRight, turnUp, turnDown are event handler functions that turn the craft left, right, up, and down.
- keypress is a catch-all event handler function for when a key on the keyboard is pressed. Depending on the key pressed, it may execute an action. I would have liked to use a key down/key up system rather than handling individual cases of a key being pressed (as it would allow for more complicated combinations of actions at the same time), but I found that deselecting the window then reselecting it disabled keyboard input, and couldn't solve the bug.
- resetSim is an event handler function that resets the simulation to its starting condition.

4.3.2 collisions

This file calculates the new trajectories of the objects after a collision.

- elasticCollision takes two objects and modifies their velocities to be the velocities of two spherical objects after a collision, using eq. (2). It does not take rotational motion into account.
- inelasticCollision takes two objects calculates the velocity of a new object formed from the two given objects. It calculates this using eq. (3). Again, it does not take rotational motion into account.

4.3.3 Thing

Thing is a class used to model a 3D object's behaviour when forces act on it. Although it does not currently support rotational motion or extended bodies, it could be relatively easily added.

Attributes

- collisionTypes is a global list of allowed collision types between objects. Currently, it only supports elastic and completely inelastic collisions.
- shapes is a list of all of the 3D shapes that make up the Thing.
- trail is an additional shape that tracks the path of the Thing through space.
- position is the position of the Thing in space, as none of the objects could form it could be centred at the 'true' centre of the object.
- forward is a vector pointing in the forward direction of the object.
- left is a vector pointing to the left of the object. Together, it and forward specify which direction is up.
- velocity is a vector specifying the velocity of the object
- mass is a scalar specifying the mass of the object
- r is the 'radius of collision' of the object; if another object's edge passes inside of this radius, then the objects are deemed 'touching'.
- name is a useful parameter for identifying Things.
- collisionType specifies what sort of collision the Thing undergoes.

Methods

- getPos, getForward, getLeft, getUp, getVelocity, getMass, getCollisionType, getTrailActive, and getName are all self-explanatory accessor functions.
- setPos, setVelocity, setMass, setCollisionType, setTrail, and setColour are all self-explanatory mutator functions.
- rotate rotates the Thing about a specified axis by a specified angle. This is noteworthy because some objects may not be centred around the object's centre.
- getSep gets the separation distance between two Things' centres. If the two centres overlap, it returns a random unit vector to prevent division by zero.
- autoMove automatically changes the position and velocity of a Thing based on the net force acting on it.
- isTouching returns true if two Things are touching and false if they are not. This function could easily be modified to use a more complicated system then a simple 'radius of collision'. For example, it could specify a 3D function in spherical coordinates from the centre of the Thing, and use that to calculate whether two Thing are overlapping. Or, it could use a series of 'test points' around the Thing's exterior.

- separate separates two Thing that are touching until they are no longer touching. It moves the lighter Thing more than the heavier Thing to more accurately model real life.
- join joins two Things together and updates the new Thing's mass and velocity; it is useful for inelastic collisions.
- gravForce calculates the gravitational attraction between two Things.
- clear acts like a makeshift deconstructor; it deletes a Thing's shapes and trail because VPython shapes will not be deleted unless they are made invisible first.

4.3.4 Craft

This is a child class of Thing that models a spacecraft by adding a rocket to the back and the ability to fire projectiles. It has all of the methods and data of a Thing, plus a few more.
New Attributes

- fuel
- ammo
- exhaustSpeed

New Methods

- getFuel, getAmmo, getExhaustSpeed, and getLength are self-explanatory accessor functions
- setFuel, setAmmo, setExhaustSpeed, setColour are self-explanatory mutator functions
- getMass is an accessor function for the mass of the craft that also adds the weight of the fuel to the craft's mass.
- turnLeft and turnUp turn the craft horizontally and vertically respectively by a specified angle. They are common operations so I made them their own functions rather than relying on the rotate function.
- fireAmmo returns a Thing, which simulates a projectile launched by the craft. It also modifies the craft's velocity to compensate for the momentum given to the projectile.
- rocketForce returns the force that the rocket aboard the craft provides, as well as decrements the amount of fuel in the craft's tank.
- cameraVector returns a vector that is pointed at the craft at a specified angle.
- setCameraAngle centres the camera on the craft at a specifies zoom and angle.

4.4 Hacks and Gotchas

4.4.1 Collisions and Division By Zero

I handled collisions between objects by giving each object a radius. If the distance between two objects' centres was less than the distance between the sum of the two objects' radii, the program deemed the objects touching. However, there is the rare circumstance that two objects could be centred in the same location. To prevent division by zero from happening, I instead set the separation between the objects to a random unit vector. This simulates the potentially random effects of having an object sitting on top of an unstable potential.

4.4.2 Deconstructor Behaviour

A quirk with how Python handles objects made inelastic collisions more complicated. My original plan was to copy the list of shapes in one object over to a second object, adjust the second object's mass and velocity, then delete the first object. However, as those shapes were actually copied by reference, they were deleted when the first object was deleted. Consequently, I added an extra step of setting the list of shapes in the first object to a null list after copying the shapes across. This allowed the first object to be deleted without affecting the shapes that were now part of the second object. Because the second object still used the shapes, the garbage collector didn't kick in, and the shapes remained part of the simulation as part of the second object.

5 Future Improvements

5.1 More Bodies

Two planets is fun. Three planets is better. It would be fun to see a more complicated system than a binary planet system.

5.2 Disappearing Objects

After a while, sometimes the craft or planets will disappear from the simulation. I think this is due to memory issues, as clearing projectiles will sometimes solve this issue. However, it needs more investigation.

5.3 Roll and Translational Motion

It would be fun to allow the craft to roll and move sideways rather than only rotate and go forward at varying speeds.

5.4 Rotational Motion

To implement rotational motion, I would only need to add a moment of inertia matrix and a rotational velocity parameter to the Thing class, as well as a torque term in the autoMove and collision functions.

6 Suggestions

This program offers a near-overwhelming number of options and configurations. In addition, the user can easily go into the makeStartObjects function and add even more bodies to the system. As a result, I've added a few suggestions for things to try to observe some of the physics built into the system.

6.1 Many Body Collisions

While paused (ideally at the start of the simulation), press the fire button multiple times while pointed at a planet. Set the projectile velocity relatively high for ease in aiming. Leave elastic collisions turned on, and for fun, turn on trails. Play the simulation and watch as the projectiles collide with eachother and the planet. If you increase the mass of the projectiles, the craft may

shoot backwards after firing. You can also turn on inelastic collisions, and watch as many of the projectiles stick to the planet.

6.2 Similar Mass Elastic Collisions

Turn on trails, increase the mass of the projectile to a comparable mass with the craft, and fire the projectile with a small initial velocity. Turn on thrusters to maximum, and accelerate into the projectile. Watch the result as you bounce off of the projectile and both the craft and the projectile change paths.

6.3 Orbital Mechanics

Slow the craft down when near the planets. Set the projectiles to a low initial velocity, and fire several around the planets. If you move away and turn your camera around, you should observe the projectiles orbiting the planets in a non-trivial manner due to the constantly-changing gravitational field.

6.4 Rocket Behaviour

Set the craft's mass to a minimum, and watch the stats section of the panel. Watch how the craft's velocity dramatically increases as fuel runs out, as the craft's total mass will become negligible compared to the thrust of the rocket.

6.5 Have fun!

Fly around, use inelastic collisions to stick multi-coloured projectiles to your craft, or enjoy bouncing off of planets! This simulation is meant to be fun, so enjoy it!