Analysis

Introduction

This project is in the field of astrodynamics, the study of the mechanics of orbits in solar systems. This subject is very difficult to give a visual intuition of – the textbooks are full of difficult equations, and few give a real demonstration of how the different variables affect orbits.

This project aims to provide a useful simulation where users can input and edit conditions and see a visual representation of various orbits. This can be used for students trying to learn more about astrodynamics, and teachers trying to give a visual demonstration.

Research

In conversations with potential users I found that the most important aspect of this program is the modelling accuracy, but also particularly the graphical representation during the simulation – since it is modelling 3D space, it should be possible in some way to navigate the space to fully picture the results of the simulation.

There are two existing programs which provide a similar service to what is outlined in the Introduction – Kerbal Space Program, and Universe Sandbox. Kerbal Space Program primarily aims to provide a tool for creating your own space program – building rockets and space stations, managing resources, and so on, but the astrodynamics of the planetary orbits and spacecraft orbit is based on real physics – the patched-conic approximation (See below). It is also limited in that the planets follow fixed paths and cannot be perturbed by other bodies. Universe Sandbox provides a full gravitational simulation based on small timesteps of Newton's Law of Gravitation, but it is relatively expensive (~£20) which does not provide for teachers in underfunded schools who want to show a simulation for maybe only a lesson or two each year.

Modelling

There are two main methods of approximating gravitational motion, which were briefly mentioned above.

The patched conic approximation:

Since in most planetary systems there is one body that is much heavier than the others, (e.g. the sun), the simplest approximation would be to ignore all interactions between pairs of lighter bodies and model only interactions between each body and the most massive body. This would be a conic approximation, since in this instance all paths would be conic sections (or degenerate conic sections, such as a fixed point or a straight line)

The patched conic approximation is slightly more complex than this, in that for each body it defines a "Sphere of Influence", which is the sphere within which this body is the most gravitationally significant body. Then interactions for a less massive body inside this sphere of influence can be modelled as a conic section with the gravitationally significant body as a focus.

For example, for a spacecraft orbiting the moon, gravitational attraction from Earth and the sun is very small, so its orbit can be modelled as an ellipse with the moon as a focus. If this spacecraft were to accelerate away from the moon, it would soon exit the “sphere of influence” of the moon and it could be modelled as following the path of a conic section with Earth as the focus.

Newtonian timestep:

The instantaneous acceleration vector on each body can be determined precisely as $\underline{a_b}=\sum_{i=2}^{i=n}(\underline{r_i}-\underline{r_b})\frac{Gm_i}{|r_i-r_b|^3}$, where n is the number of bodies in the system, $\underline{r_b}$ is the position vector of the body, $\underline{r_i}$ is the position vector of another body, m_i is the mass of the other body, and G is the gravitational constant. Since $\underline{r_i}=\underline{r_{i0}}+\iint_{t=0}^{t=t_{now}}\underline{a_i}\,dt$, it is quite quickly becoming clear that the equation of the position of a body at a certain time may be unsolvable (and indeed it is!).

The Newtonian timestep approximation attempts to approximate a solution to this double integral via numerical methods:

$$\underline{r_{b,n+1}}=\underline{r_{b,n}}+\iint_{t=t_n}^{t_{n+1}}\underline{a_b}\,dt \approx \underline{r_{b,n}}+\underline{v_{b,n}}(t_{n+1}-t_n)+\frac{1}{2}\underline{a_{b,n}}(t_{n+1}-t_n)^2$$

, where

$$\underline{v_{b,n+1}}=\int_{t=t_n}^{t=t_{n+1}}\underline{a_b}\,dt \approx \underline{v_{b,n}}+\underline{a_{b,n}}(t_{n+1}-t_n)$$

, where $\underline{a_{b,n}}$, $\underline{v_{b,n}}$, and $\underline{r_{b,n}}$ are the acceleration, velocity, and position vectors of body b at time t_n. In this case, we must define a “timestep” $t_{n+1} - t_n$. The lower this timestep is, the more accurate the approximation will be, but the more calculations (and hence more real-world time) required for the same amount of in-simulation time.

Here is a comparison

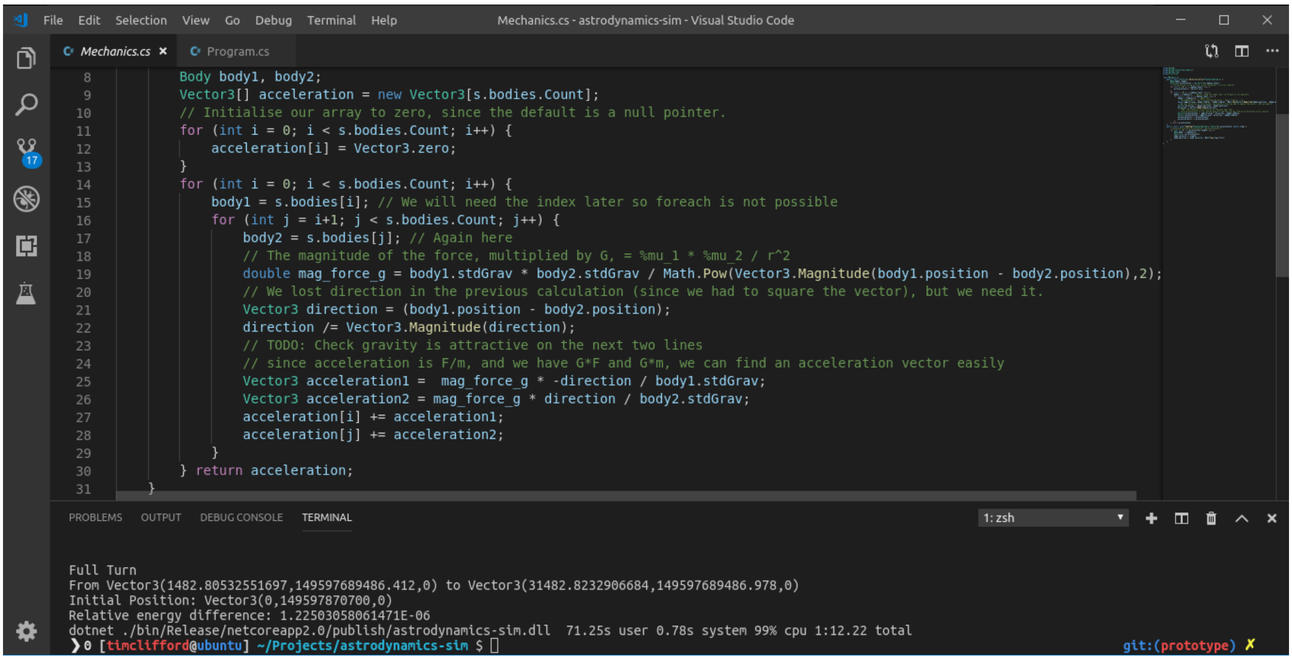
Newtonian Timestep	Patched-Conic Approximation
Can model complex interactions accurately (e.g binary star systems)	More accurate for most simple systems (eg. our solar system)
Without enough processing power, it will either run slowly or inaccurately	Can run arbitrarily fast
	Less processor intensive

If the mechanics can be done directly from Newton’s law of gravitation and remain accurate and fast, then it will be a much better option than the patched conic approximation, since it gives much greater flexibility in the types of situations which can be modelled. To test this, I built an early prototype of the mechanics system to find out. The test was as follows: Approximate initial conditions of an Earth-Sun system by hand, enter the conditions into the mechanics algorithm, and test running time against the change in specific energy, since the change in specific energy is the most important consequence of an inaccurate algorithm – the planets will begin to spiral away from the object they should be orbiting rather than remaining bound.

The result was that at a real-life orbit period of a few seconds the change in energy was +0.1%, and at an orbit period of a minute, it dropped to +0.001%. This was on a very low-powered machine, so this amount of error is more than could be expected on a modern computer.

In a conversation with a potential user, we agreed that this would be an acceptable amount of error accumulation, since it will not be noticeable to the human eye, and since the program would be primarily used as an educational tool rather than professionally.

Prototype Screenshot:


File Edit Selection View Go Debug Terminal Help
Mechanics.cs - astrodynamics-sim - Visual Studio Code
Mechanics.cs ×
Program.cs
17
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
1: zsh

```
        Body body1, body2;
        Vector3[] acceleration = new Vector3[s.bodies.Count];
        // Initialise our array to zero, since the default is a null pointer.
        for (int i = 0; i < s.bodies.Count; i++) {
            acceleration[i] = Vector3.zero;
        }
        for (int i = 0; i < s.bodies.Count; i++) {
            body1 = s.bodies[i]; // We will need the index later so foreach is not possible
            for (int j = i+1; j < s.bodies.Count; j++) {
                body2 = s.bodies[j]; // Again here
                // The magnitude of the force, multiplied by G, = %mu_1 * %mu_2 / r^2
                double mag_force_g = body1.stdGrav * body2.stdGrav / Math.Pow(Vector3.Magnitude(body1.position - body2.position),2);
                // We lost direction in the previous calculation (since we had to square the vector), but we need it.
                Vector3 direction = (body1.position - body2.position);
                direction /= Vector3.Magnitude(direction);
                // TODO: Check gravity is attractive on the next two lines
                // since acceleration is F/m, and we have G*F and G*m, we can find an acceleration vector easily
                Vector3 acceleration1 =  mag_force_g * -direction / body1.stdGrav;
                Vector3 acceleration2 = mag_force_g * direction / body2.stdGrav;
                acceleration[i] += acceleration1;
                acceleration[j] += acceleration2;
            }
        } return acceleration;
    }
```

```
Full Turn
From Vector3(1482.80532551697,149597689486.412,0) to Vector3(31482.8232906684,149597689486.978,0)
Initial Position: Vector3(0,149597870700,0)
Relative energy difference: 1.22503058061471E-06
dotnet ./bin/Release/netcoreapp2.0/publish/astrodynamics-sim.dll  71.25s user 0.78s system 99% cpu 1:12.22 total
❯0 [timclifford@ubuntu] ~/Projects/astrodynamics-sim $                                          git:(prototype) ✗
```

We also agreed that for ease of use it would be best to describe the bodies as following conic orbits, and then convert into a pure position and velocity before starting the simulation.

Since users will want to be able to tweak different variables a lot, we agreed that it would be very good also, if possible, to provide a tool to save configurations to the hard drive and load them again later.

Final Objectives:

1. Accurately model simple Keplerian orbits – circular, elliptical, parabolic, and hyperbolic
2. Provide a tool to add, remove, and edit bodies with classical orbital elements, and provide an explanation:

Name	Description	Symbol	Range
Semi-latus rectum	the distance between two bodies at right angles to the "periapsis" (minimum point)	ρ	[0,+∞)
Eccentricity	A measure of the shape of the orbit	e	[0,+∞)
Inclination	the angle between the orbital plane and the reference plane	i	[0,180º]
Longitude of the ascending node	the angle from the reference direction anticlockwise to the point where the orbiting body rises above the reference plane	Ω	[0,360º)
Argument of periapsis	the angle from the ascending node anticlockwise to the periapsis.	ω	[0,360º)
True anomaly	The angle from the periapsis anticlockwise to the current position of the body.	ν	[0,360º)

3. Simulate motion in non-classical systems – e.g a rogue planet entering the solar system
4. Render the 3D system into 2D convincingly.
5. Be able to place one or more cameras anywhere in the system and provide controls for zoom/rotation/etc.
6. Be able to pause the simulation and modify variables at any point.
7. Provide a capability to save and load systems to/from the hard drive.

Design

This is split into three main areas: Mechanics, Graphics, and UI, which have been approached separately. Mechanics deals with calculations of how the bodies are affected with time, Graphics deals with the 3D projection of the simulation to the user while it is running, and UI deals with the process of adding, removing, and editing bodies.

Mechanics

Mathematical Structures:

We will need some basic mathematical structures to make 3D calculations easier: the 3-Vector, which stores 3 real numbers, and can be used for storing positions, velocities, and so on, and the 3x3 Matrix, which makes operations on 3-Vectors like rotation easier. These will also be useful for the Graphics process. These data types should be able to perform simple mathematical functions implicitly and provide static functions for more complex operations such as the dot and cross product.

Since the double type will be used for the simplest part of these structures, and since after repeated computation some small errors will accumulate, the equality of these structures will check that each component value is within a small value of each other rather than exactly the same. By experimentation, a value of 10^{-10} was deemed suitable.

The implementation of these structures can be seen in Appendix 1.

Bodies:

There must be a data structure which describes a body in the system. This will be stored in main memory as the program is running, but also must be able to be saved to disk.

Since we chose to use a Newtonian timestep approximation, the three variables required for mechanics are:

StdGrav – the standard gravitational parameter of the body.
Position – the current position of the body (as a vector)
Velocity – the current velocity of the body (as a vector)

For identification of bodies and initial orbit calculations the following two are also required:

Name – the name of the Body
Parent – the body this body is orbiting

And for the Graphics process:

Color – the color of the body

This class should be able to be constructed from the six orbital elements, so an “OrbitalElements” class is required, storing these elements.

The process of converting an orbit determined by the six orbital elements into an initial position and velocity is as follows:

```
// First, find the position and velocity in “perifocal coordinates”, which means that
// the orbital plane is the X-Y plane, and the periapsis is along the x axis
```

$$\underline{r_p} \leftarrow \frac{\rho}{1+e\cos\nu}\begin{pmatrix}\cos\nu\\ \sin\nu\\ 0\end{pmatrix} \quad ; \quad \underline{v_p} \leftarrow \sqrt{\frac{\mu}{\rho}}\begin{pmatrix}-\sin\nu\\ \cos\nu+e\\ 0\end{pmatrix}$$

```
// Now we must convert these into common IJK coordinates. This is in fact several
// rotations, from which a matrix was calculated by hand.
```

$$M \leftarrow \begin{pmatrix}\cos\Omega\cos\omega-\sin\Omega\sin\omega\cos i & -\cos\Omega\sin\omega-\sin\Omega\cos\omega\cos i & \sin\Omega\sin i\\ \sin\Omega\cos\omega+\cos\Omega\sin\omega\cos i & -\sin\Omega\sin\omega+\cos\Omega\cos\omega\cos i & -\cos\Omega\sin i\\ \sin\omega\sin i & \cos\omega\sin i & \cos i\end{pmatrix}$$

$$\underline{r} \leftarrow M\,\underline{r_p} \quad ; \quad \underline{v} \leftarrow M\,\underline{v_p}$$

The implementation of the Body class can be found in Appendix 2

Since we also want to be able to pause the simulation and view/edit variables, we need also to be able to run this algorithm in reverse. This is more complex, since it will contain arcos(), so we must ensure that our angles are in the correct quadrant.

We will first calculate the “Fundamental Vectors”, which are a sort of interim state between position-velocity and orbital elements. This will be implemented as its own class, with the calculation in the constructor. These vectors are as follows:

```
// Specific Angular Momentum: This is the angular momentum of the body around its parent, divided by the mass of the parent. It can be shown to be constant in any conic orbit.
  $\underline{h} \leftarrow \underline{r} \times \underline{v}$
// Eccentricity: a vector in the direction of the periapsis, with magnitude of the eccentricity of the orbit
  $\underline{e} \leftarrow \underline{r}(\frac{v^2}{\mu} - \frac{1}{r}) - (\underline{r}.\underline{v}) \times \underline{v}$
// Node: The direction of the node where the orbital plane intersects the I-J plane
  $\underline{n} \leftarrow \underline{h} \times \underline{K}$  // K := (0,0,1)
```

From these vectors we can then calculate the orbital elements:

```
  $e \leftarrow |\underline{e}|$  ;  $\rho \leftarrow \frac{h^2}{\mu}$  ;  $i \leftarrow \arccos(\hat{\underline{h}}.z)$
  $\cos\Omega \leftarrow \hat{\underline{n}}.x$
IF (n.y >= 0) THEN Ω ← arcos(cos Ω)
ELSE Ω ← 2π – arcos(cos Ω)
  $\cos\omega \leftarrow \hat{\underline{n}}.\hat{\underline{e}}$
IF (e.z >= 0) THEN ω ← arcos(cos ω)
ELSE ω ← 2π – arcos(cos ω)
  $\cos\nu \leftarrow \hat{\underline{e}}.\hat{\underline{r}}$
IF (|r|.|v| >= 0) THEN ν ← arcos(cos ν)
ELSE ν ← 2π – arcos(cos ν)
```

In implementation this algorithm must be modified, since double precision floating point error leads to nonsense values for ν at small values of e and Ω. It is assumed that if e is small, the periapsis is at Ω, and if Ω is also small, it is the I vector.

The implementation of this algorithm can be seen in Appendix 3

PlanetarySystem:

We also need a class to store the planetary system as a whole. This should inherit from the standard IEnumerator interface, which allows the bodies which are stored in it to be iterated through, indexed, added to, and removed from. It will store the bodies as a protected list. This simplifies access to the system from other objects.

Bodies – a list of the bodies in the system
Centers – the index of each body in the system which can be focused on when the system is drawn.

The planetary system class will also handle all of the mechanics of the system. As it was decided to use Newton’s laws, the instantaneous acceleration on each body must be calculated first.

```
foreach body A in bodies:
        foreach body B in bodies:
                accelerationA ← accelerationA + b.stdGrav/(distance between them squared)
                accelerationB ← accelerationB + a.stdGrav/(distance between them squared)
```

This runs in O(n^2) time, which is acceptable. This calculation should be in a function called GetAcceleration().

The PlanetarySystem class will include a method TimeStep(step), which takes the instantaneous accelerations and assumes they are constant for a small time “step” in seconds, and modifies the bodies.

```
foreach body B in bodies:
    B.position += B.velocity*timestep + (timestep^2/2)*accelerationB
```

The calculations should be encapsulated within the class and run asynchronously to the other processes. Public Start(), StartAsync(), and Stop() functions should be provided. These rely on a flag in the class which says whether the program is running.

The implementation of the PlanetarySystem class can be seen in Appendix 4

Graphics:

Gtk# was chosen for this project as a graphics library since it is used professionally, has good support, and runs well on Linux with the Mono runtime.

The graphics processor should iterate through the planetary system class asynchronously from the mechanics, and, in order of distance to the camera, draw each object (and it’s associated trail), passing it through a camera transformation to project it onto the screen.

Required classes:
Camera – contains an orientation and distance to the origin, as well as transformation function
SystemView – overloads an existing GTK class and defines what happens when the screen is drawn.
Variables:
    active planetary system
    list of previous positions of bodies (to draw path)
Variables which are modified by Input class:
    planetary radius multiplier
    trail length
    active camera
Constants:
    line width (as a fraction of planetary size)
Pseudocode:

```
Fill screen with black
Set (0,0) as centre of screen
Scale coordinates to screen
order ← order of bodies according to distance from camera (closest first)
foreach Body in system according to order:
    DrawCircle(radius Body.radius*radius multiplier, position: camera transformation of Body.position of color Body.color
    lastPath ← first point in paths[Body]
    foreach (point in paths[Body]):
        DrawLine(camera transform of lastPath to camera transform of point, width: Body.radius*radius multiplier*line width, color Body.color)
    if len(paths) > maximum trail length:
        paths ← last (maximum trail length) elements of paths
```

GTK is event based, which means that ordinarily the screen would only be refreshed if an event happens, eg the player clicks on something. Therefore we must create our own method to manually trigger the draw function quickly so that the simulation can be seen. The starting and stopping is achieved via a flag – it is very important to terminate all processes when the program is stopped, since we have many asynchronous processes which could theoretically carry on even if the main thread is stopped.

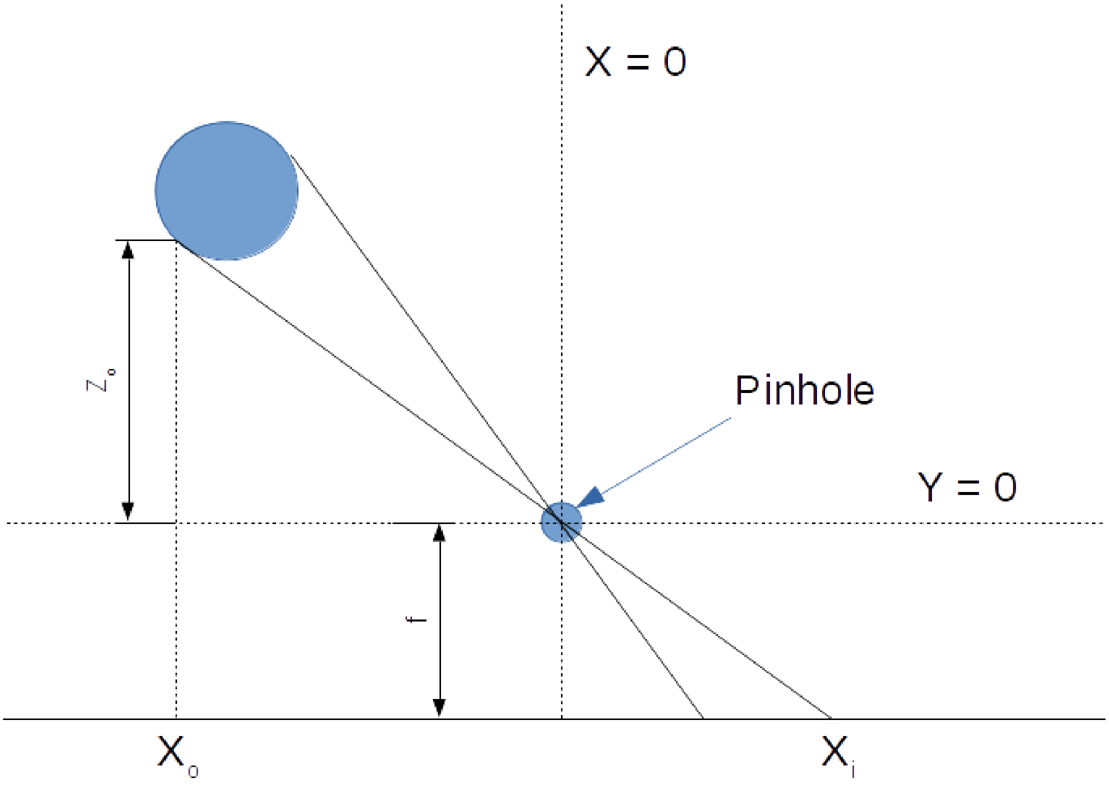
The implementation of the SystemView class can be seen in Appendix 5

Camera Transformation:

Since we want to be able to focus on a certain body and still move the camera around, a camera is modelled as a point on a sphere which looks directly at the origin. This requires two variables: an angle (relative to the IJK vector system), and a distance from the origin.

I chose to model the camera as a pinhole camera, because the implementation is relatively simple and it provides a projection which looks very accurate. This is where light to the camera passes through a “pinhole” and lands on a screen behind it. The location of each point on the screen is it’s projection onto our 2D screen.

Diagram:



By similar triangles, $X_i = -X_o \frac{f}{Z_o}$, but since we do not want to flip directions in our simulation, we will use $X_i = X_o \frac{f}{Z_o}$. This formula is exactly the same for a 3rd dimension: $Y_i = Y_o \frac{f}{Z_o}$.

By the same argument the radius of a sphere which is projected in this way is $r_i = r_o \frac{f}{Z_o}$ in the case that $r_o << d$, so this will be the radius of our objects.

To apply this algorithm we must first transform the coordinates such that the position of the camera is (0,0,0) and the vector (0,0,1) points towards the "origin". This can be achieved simply using our Matrix3 class.

Pseudocode:

```
SUB CameraTransform(position, camera, origin)
# position is a Vector3 object, camera is a Camera object
# using Tait-Bryan angles
cameraLocalPosition ← Matrix3.IntrinsicZYXRotation(camera.angle)
       *Vector3(0,0,-camera.focalLength)
localPosition ← position - cameraLocalPosition - origin
localPositionRotated ← Matrix3.ExtrinsicZYXRotation(camera.angle) *localPosition
OUTPUT (camera.focalLength/localPositionRotated.Z)*localPositionRotated
# the Z value of the output will not be used,
# so it does not matter that it is transformed like X and Y
```

The camera transformation can be made "orthographic" (i.e. distances are not distorted), by increasing the focal length to a very large value. This will be an option to the user via a key command.

The implementation of the camera can be seen in Appendix 6

UI:

The UI must have functionality to set variables such as the mechanics timestep, trail length, and planetary radii multiplies, as well as add and remove bodies, set body names, set body variables, save and load the current state.

Since most of the data in this menu will be specific to a body, it makes sense to define a separate class that describes a box containing the body variables/sliders etc.

The required properties:

name: text box
parent: drop down menu
mass: slider
radius: slider
semilatusrectum: slider
eccentricity: slider
inclination: slider

ascendingNodeLongitude: slider
periapsisArgument: slider
trueAnomaly: slider
focusable: check button (allows the body to be set as the centre of the camera projection)
delete: button

This class should also reference the body it describes, and be able to set the body variables based on the sliders, or set the sliders based on the body. It should also be able to send a message to the parent menu class to remove it if the delete button is pressed.

It is linked to the body via aggregation. This decouples the UI process from the Mechanics which happen later – when the UI Process terminates itself on the "Done" button click, the bodies it created will not also be destroyed.

The UI Implementation can be seen in Appendix 7

Since the variables are so difficult to understand, we should also have some sort of help function. The best way to do this would be to link to a web page, since this allows for much better formatting than a textbox inside a program. Since web design is not the focus of this project, I decided to create a markdown file containing the help and convert it to HTML via 3rd party software.

Final Markdown and rendered HTML is in Appendix 8

Save/Load:

I chose XML as the file format to save into, since it is human readable but still offers good flexibility as to the types that are saved. I decided a custom save file class should be created, acting as a wrapper for the data that must be saved.

Required data:

- Mechanics Timestep
- Planetary Radii multiplier'
- Orbit Trail Length
- Body variables
- which bodies can be focused on

Several Example systems are provided as XML files. These are outlined in the help file, and provide a tool for new users to understand the capabilities of the system without having to create their own system. An example (Our solar system) is shown in Appendix 9

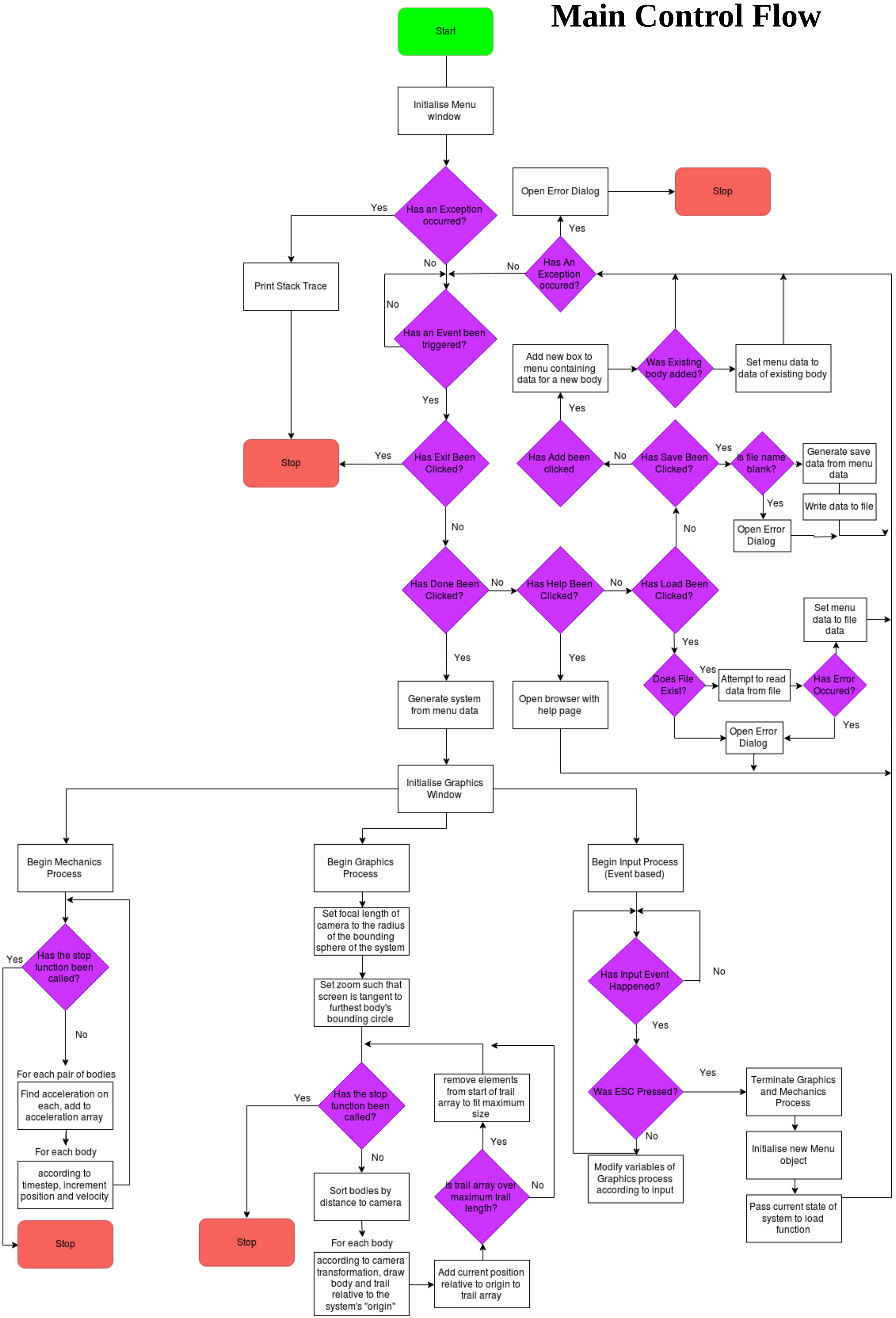
Input

In simulation input will be achieved via Event-based triggering. EventHandlers are defined for each type of input, which have access to the active PlanetarySystem and SystemView objects via the Program class, which starts the program and stores the active objects as public properties.

The implementation can be seen in Appendix 10

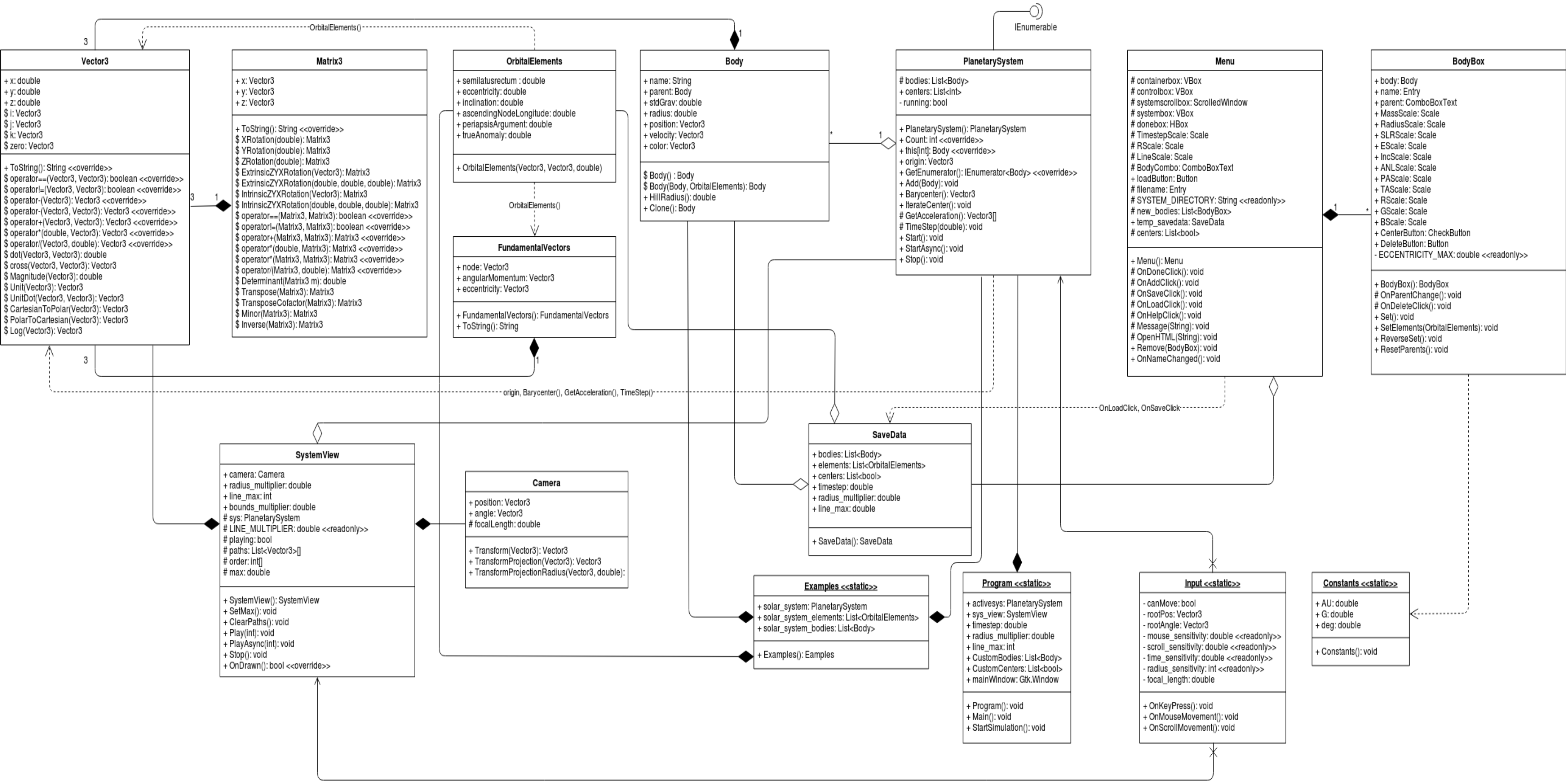
Pausing

I decided that the best way to allow users to stop and edit the variables of the system would be to take them back to the original setup UI screen. To take advantage of the already defined functions I decided to generate a new SaveData object, but instead of writing it to a file, to create a new UI object and run the Load command, passing this SaveData as an object. Therefore the user can stop the simulation whenever he/she wants and modify the orbit variables. This control flow can be seen in the main control flowchart below.



Main Control Flow

Start
Initialise Menu window
Has an Exception occurred?
Yes
No
Print Stack Trace
Stop
Open Error Dialog
Stop
Yes
Has An Exception occured?
No
Has an Event been triggered?
No
Yes
Has Exit Been Clicked?
Yes
Stop
No
Add new box to menu containing data for a new body
Was Existing body added?
Set menu data to data of existing body
Yes
Has Add been clicked
No
Has Save Been Clicked?
Yes
Is file name blank?
Generate save data from menu data
Write data to file
Yes
Open Error Dialog
No
Has Done Been Clicked?
No
Has Help Been Clicked?
No
Has Load Been Clicked?
Yes
Set menu data to file data
Yes
Yes
Does File Exist?
Yes
Attempt to read data from file
Has Error Occured?
Yes
Open Error Dialog
Generate system from menu data
Open browser with help page
Initialise Graphics Window
Begin Mechanics Process
Has the stop function been called?
Yes
No
For each pair of bodies
Find acceleration on each, add to acceleration array
For each body
according to timestep, increment position and velocity
Stop
Begin Graphics Process
Set focal length of camera to the radius of the bounding sphere of the system
Set zoom such that screen is tangent to furthest body's bounding circle
Has the stop function been called?
Yes
No
Stop
Sort bodies by distance to camera
For each body
according to camera transformation, draw body and trail relative to the system's "origin"
Add current position relative to origin to trail array
Is trail array over maximum trail length?
Yes
No
remove elements from start of trail array to fit maximum size
Begin Input Process (Event based)
Has Input Event Happened?
No
Yes
Was ESC Pressed?
Yes
No
Modify variables of Graphics process according to input
Terminate Graphics and Mechanics Process
Initialise new Menu object
Pass current state of system to load function


OrbitalElements()
3
1
IEnumerable
Vector3
+ x: double
+ y: double
+ z: double
$ i: Vector3
$ j: Vector3
$ k: Vector3
$ zero: Vector3
+ ToString(): String <<override>>
$ operator==(Vector3, Vector3): boolean <<override>>
$ operator!=(Vector3, Vector3): boolean <<override>>
$ operator-(Vector3): Vector3 <<override>>
$ operator-(Vector3, Vector3): Vector3 <<override>>
$ operator+(Vector3, Vector3): Vector3 <<override>>
$ operator*(double, Vector3): Vector3 <<override>>
$ operator/(Vector3, double): Vector3 <<override>>
$ dot(Vector3, Vector3): double
$ cross(Vector3, Vector3): Vector3
$ Magnitude(Vector3): double
$ Unit(Vector3): Vector3
$ UnitDot(Vector3, Vector3): Vector3
$ CartesianToPolar(Vector3): Vector3
$ PolarToCartesian(Vector3): Vector3
$ Log(Vector3): Vector3
3
1
Matrix3
+ x: Vector3
+ y: Vector3
+ z: Vector3
+ ToString(): String <<override>>
$ XRotation(double): Matrix3
$ YRotation(double): Matrix3
$ ZRotation(double): Matrix3
$ ExtrinsicZYXRotation(Vector3): Matrix3
$ ExtrinsicZYXRotation(double, double, double): Matrix3
$ IntrinsicZYXRotation(Vector3): Matrix3
$ IntrinsicZYXRotation(double, double, double): Matrix3
$ operator==(Matrix3, Matrix3): boolean <<override>>
$ operator!=(Matrix3, Matrix3): boolean <<override>>
$ operator+(Matrix3, Matrix3): Matrix3 <<override>>
$ operator*(double, Matrix3): Matrix3 <<override>>
$ operator*(Matrix3, Matrix3): Matrix3 <<override>>
$ operator/(Matrix3, double): Matrix3 <<override>>
$ Determinant(Matrix3 m): double
$ Transpose(Matrix3): Matrix3
$ TransposeCofactor(Matrix3): Matrix3
$ Minor(Matrix3): Matrix3
$ Inverse(Matrix3): Matrix3
OrbitalElements
+ semilatusrectum : double
+ eccentricity: double
+ inclination: double
+ ascendingNodeLongitude: double
+ periapsisArgument: double
+ trueAnomaly: double
+ OrbitalElements(Vector3, Vector3, double)
OrbitalElements()
FundamentalVectors
+ node: Vector3
+ angularMomentum: Vector3
+ eccentricity: Vector3
+ FundamentalVectors(): FundamentalVectors
+ ToString(): String
1
3
origin, Barycenter(), GetAcceleration(), TimeStep()
Body
+ name: String
+ parent: Body
+ stdGrav: double
+ radius: double
+ position: Vector3
+ velocity: Vector3
+ color: Vector3
$ Body() : Body
$ Body(Body, OrbitalElements): Body
+ HillRadius(): double
+ Clone(): Body
*
1
PlanetarySystem
# bodies: List<Body>
+ centers: List<int>
- running: bool
+ PlanetarySystem(): PlanetarySystem
+ Count: int <<override>>
+ this[int]: Body <<override>>
+ origin: Vector3
+ GetEnumerator(): IEnumerator<Body> <<override>>
+ Add(Body): void
+ Barycenter(): Vector3
+ IterateCenter(): void
# GetAcceleration(): Vector3[]
# TimeStep(double): void
+ Start(): void
+ StartAsync(): void
+ Stop(): void
Menu
# containerbox: VBox
# controlbox: VBox
# systemscrollbox: ScrolledWindow
# systembox: VBox
# donebox: HBox
# TimestepScale: Scale
# RScale: Scale
# LineScale: Scale
# BodyCombo: ComboBoxText
+ loadButton: Button
# filename: Entry
# SYSTEM_DIRECTORY: String <<readonly>>
# new_bodies: List<BodyBox>
+ temp_savedata: SaveData
# centers: List<bool>
+ Menu(): Menu
# OnDoneClick(): void
# OnAddClick(): void
# OnSaveClick(): void
# OnLoadClick(): void
# OnHelpClick(): void
# Message(String): void
# OpenHTML(String): void
+ Remove(BodyBox): void
+ OnNameChanged(): void
1
*
BodyBox
+ body: Body
+ name: Entry
+ parent: ComboBoxText
+ MassScale: Scale
+ RadiusScale: Scale
+ SLRScale: Scale
+ EScale: Scale
+ IncScale: Scale
+ ANLScale: Scale
+ PAScale: Scale
+ TAScale: Scale
+ RScale: Scale
+ GScale: Scale
+ BScale: Scale
+ CenterButton: CheckButton
+ DeleteButton: Button
- ECCENTRICITY_MAX: double <<readonly>>
+ BodyBox(): BodyBox
# OnParentChange(): void
# OnDeleteClick(): void
+ Set(): void
+ SetElements(OrbitalElements): void
+ ReverseSet(): void
+ ResetParents(): void
OnLoadClick, OnSaveClick
SystemView
+ camera: Camera
+ radius_multiplier: double
+ line_max: int
+ bounds_multiplier: double
# sys: PlanetarySystem
# LINE_MULTIPLIER: double <<readonly>>
# playing: bool
# paths: List<Vector3>[]
# order: int[]
# max: double
+ SystemView(): SystemView
+ SetMax(): void
+ ClearPaths(): void
+ Play(int): void
+ PlayAsync(int): void
+ Stop(): void
+ OnDrawn(): bool <<override>>
Camera
+ position: Vector3
+ angle: Vector3
# focalLength: double
+ Transform(Vector3): Vector3
+ TransformProjection(Vector3): Vector3
+ TransformProjectionRadius(Vector3, double):
SaveData
+ bodies: List<Body>
+ elements: List<OrbitalElements>
+ centers: List<bool>
+ timestep: double
+ radius_multiplier: double
+ line_max: double
+ SaveData(): SaveData
Examples <<static>>
+ solar_system: PlanetarySystem
+ solar_system_elements: List<OrbitalElements>
+ solar_system_bodies: List<Body>
+ Examples(): Eamples
Program <<static>>
+ activesys: PlanetarySystem
+ sys_view: SystemView
+ timestep: double
+ radius_multiplier: double
+ line_max: int
+ CustomBodies: List<Body>
+ CustomCenters: List<bool>
+ mainWindow: Gtk.Window
+ Program(): void
+ Main(): void
+ StartSimulation(): void
Input <<static>>
- canMove: bool
- rootPos: Vector3
- rootAngle: Vector3
- mouse_sensitivity: double <<readonly>>
- scroll_sensitivity: double <<readonly>>
- time_sensitivity: double <<readonly>>
- radius_sensitivity: int <<readonly>>
- focal_length: double
+ OnKeyPress(): void
+ OnMouseMovement(): void
+ OnScrollMovement(): void
Constants <<static>>
+ AU: double
+ G: double
+ deg: double
+ Constants(): void

Tests

Tests were included as a part of the design of data structures. These are outlined below.

Vector/Matrix

Class	Test	Type	Expected	Pass/Fail
Vector3	(2,3,6) + zero = zero + (2,3,6) = (2,3,6)	Normal	True	Pass
Vector3	5 * (2,3,6) / 5	Normal	(2,3,6)	Pass
Vector3	(2,3,6) + (2,3,6) = 2 * (2,3,6)	Normal	True	Pass
Vector3	-a = 0 - a	Normal	True	Pass
Vector3	Magnitude(2,3,6)	Normal	7	Pass
Vector3	(1,2,0) dot (-2,1,0)	Normal	0	Pass
Vector3	(2,3,6) dot (2,3,6)	Normal	49	Pass
Vector3	(3,−3,1) cross (4,9,2).	Normal	(−15,−2,39)	Pass
Vector3	Unit(2,3,6)	Normal	(0.285…,0.428…,0.857...)	Pass
Vector3	Unit(0,0,0)	Erroneous	DivideByZeroException	Pass
Vector3	PolarToCartesian(CartesianToPolar((2,3,6))	Normal	(2,3,6)	Pass
Vector3	Null = null	Erroneous	True	Pass
Vector3	Null = (2,3,6)	Erroneous	False	Pass
Matrix3	I*I = I	Normal	True	Pass
Matrix3	I*[1,3,1],[0,4,1],[2,-1,0] = [1,3,1],[0,4,1],[2,-1,0]*I = [1,3,1],[0,4,1],[2,-1,0]	Normal	True	Pass
Matrix3	5* ([1,3,1],[0,4,1],[2,-1,0]) / 5	Normal	([1,3,1],[0,4,1],[2,-1,0])	Pass
Matrix3	[1,3,1],[0,4,1],[2,-1,0] + [1,3,1],[0,4,1],[2,-1,0] = 2 * [1,3,1],[0,4,1],[2,-1,0]	Normal	True	Pass
Matrix3	Inverse([1,3,1],[0,4,1],[2,-1,0])	Normal	([-1,1,1],[-2,2,1],[8,-7,-4])	Pass
Matrix3	Inverse(Inverse([1,3,1],[0,4,1],[2,-1,0]))	Normal	([1,3,1],[0,4,1],[2,-1,0])	Pass
Matrix3	Inverse([0,0,0],[0,0,0],[0,0,0])	Erroneous	DivideByZeroException	Pass
Matrix3	([1,3,1],[0,4,1],[2,-1,0])*([1,3,1],[0,4,1],[2,-1,0])	Normal	([3,14,4],[2,15,4],[2,2,1])	Pass

Body/OrbitalElements Tests

Test	Type	Expected	Pass/Fail
Identical bodys created orbiting static and moving bodies	Normal	Created bodies' position and velocity differ by that of the moving body	Pass
OrbitalElements(Body(all possible values of variables, in increments of 0.2))	Normal	Equal to the original variables, and all circular orbits of the same radius have the same velocity	Pass
OrbitalElements(angles greater than 2pi)	Erroneous	Angles are implicitly converted to be within bounds	Pass

PlanetarySystem tests

Test	Type	Expected	Pass/Fail
Construct system from list of bodies	Normal	System contains bodies	Pass
Add body to system	Normal	System contains body	Pass
Barycenter of two bodies on line	Normal	Returns position on line in ratio of gravities	Pass
Barycenter of bodies on vertices of equilateral triangle	Normal	Returns center of triangle	Pass
Specific Energy drift while simulating (tested as prototype)	Normal	Not Significant	Pass

The implementation of these tests is in Appendix 11

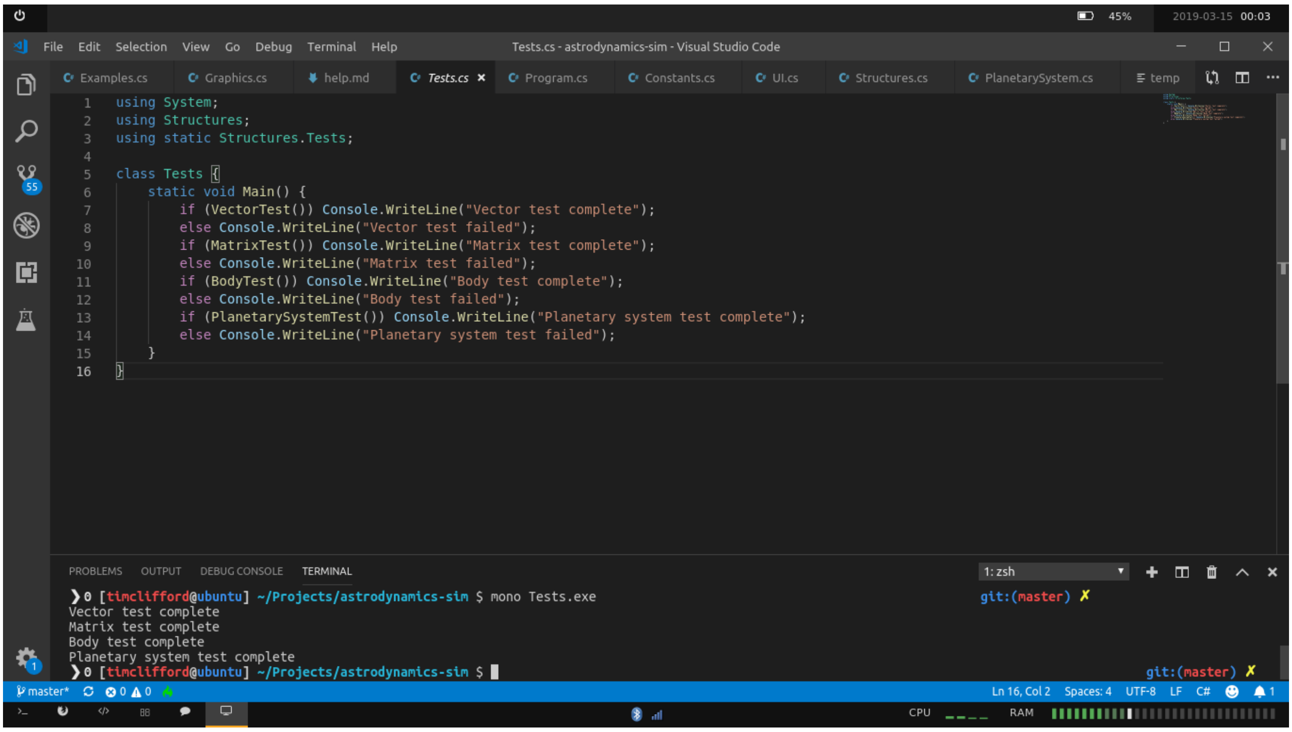
System Tests

The Graphics and UI were tested qualitatively as part of the entire system.

Test	ID	Type	Expected	Pass/Fail
UI provides functional buttons/sliders for all options	1	Normal	Buttons work as labelled	Pass
Invalid filename is entered and load button is pressed	2	Erroneous	UI shows error message	Pass
File is saved, program is exited, and file is loaded again	3	Normal	Options are exactly the same	Pass
ESC is pressed during simulation	4	Normal	Simulation exits and UI menu is loaded with current system parameters	Pass
F is pressed during simulation	5	Normal	Camera focuses on next body	Pass
C is pressed during simulation	6	Normal	Camera switches to stereoscopic mode	Pass
Scroll wheel and mouse are used during simulation	7	Normal	Camera zooms and moves	Pass
Arrow keys and PgUp/PgDown are pressed	8	Normal	Graphics variables are changed	Pass
System is started with rogue planet in hyperbolic orbit	9	Normal	Believable non keplerian motion, orbital parameters change.	Pass

Test Screenshots/Videos:

Data structures test screenshot:



```
using System;
using Structures;
using static Structures.Tests;

class Tests {
    static void Main() {
        if (VectorTest()) Console.WriteLine("Vector test complete");
        else Console.WriteLine("Vector test failed");
        if (MatrixTest()) Console.WriteLine("Matrix test complete");
        else Console.WriteLine("Matrix test failed");
        if (BodyTest()) Console.WriteLine("Body test complete");
        else Console.WriteLine("Body test failed");
        if (PlanetarySystemTest()) Console.WriteLine("Planetary system test complete");
        else Console.WriteLine("Planetary system test failed");
    }
}
```

```
❯0 [timclifford@ubuntu] ~/Projects/astrodynamics-sim $ mono Tests.exe                git:(master) ✗
Vector test complete
Matrix test complete
Body test complete
Planetary system test complete
❯0 [timclifford@ubuntu] ~/Projects/astrodynamics-sim $                                     git:(master) ✗
```

System Test documentation was taken as screen recordings and can be found at this link:

https://www.dropbox.com/sh/3gn5k2rxhtp207k/AADQsOLul0vZfNrF8RVAU8pIa?dl=0

Evaluation

Interview with Prospective User

An interview with Sophie Williams, A level physics student and prospective engineer.

“How was the user interface of the program?

Having the help button was extremely useful. The user interface was clear and concise, and simple enough to not overcrowd the screen. It’s meant to be a scientific demonstration so it was beneficial having all the different options on display.

What about in the simulation?

It would have been nice to see the values on display while running the simulation, even just saying how much it changes when the buttons are pressed, particularly the speed and trail length.

Did you enjoy the program?

Yes! It was very educational, especially about binary star systems which I had never really considered before – it was fascinating to see the movement of different bodies in that binary star system. It was really useful having the preloaded systems to see what works and what doesn't before I tried to create my own system.

Do you understand orbital mechanics more now?

Definitely.

Would you recommend this program to other like minded people?

Yes! It's useful and helpful despite some inaccuracies at high simulation speed. Also fun to watch!

Do you think this would be a good tool for any of your teachers to teach about orbital mechanics?

I think students benefit greatly from seeing what the maths they are learning leads to, and believe this would be a great tool in being able to do that.

How convincing is the 3D projection?

Very. It's useful to be able to move the camera around, which gives a very good impression of the 3D projection onto the 2D screen. Of course it's impossible to get a realistic 3D projection until you have a hologram but it's doing pretty good for a computer program.

Any other comments?

I thought being able to change the color of planets gave it a nice touch."

Comparison against initial objectives:

1. Accurately model simple Keplerian orbits – circular, elliptical, parabolic, and hyperbolic
    - Success. See eccentricity demo in Appendix 8

2. Provide a tool to add, remove, and edit bodies with classical orbital elements, and provide explanation.
    - Success. See System Test 1 and 2
3. Simulate motion in non-classical systems – e.g a rogue planet entering the solar system
    - Success. See RoguePlanet1 (System Test 9)
4. Render the 3D system into 2D convincingly.
    - Success. Agreed by interviewee.
5. Be able to place one or more cameras anywhere in the system and provide controls for zoom/rotation/etc.
    - Success. See System Tests 5, 6 and 7
6. Be able to pause the simulation and modify variables at any point.
    - Success. See System Test 4
7. Provide a capability to save and load systems to/from the hard drive.
    - Success. See System Test 3

Conclusion

Overall, I think this project was a success. The objectives were met, the potential users were generally satisfied, and the program itself is fun to play with and very educational. As in any program, there are certainly things to be improved. Like a user suggested, the interface during the simulation could be more extensive, perhaps with a sidebar showing all the options with sliders etc. This might have been looked at more if I started the project again.

I am very pleased with the 3D projection, but it could be improved – maybe with realistic simulation of light sources via raytracing, or texture maps on the surface of planets. The mechanics, too – pertubations from radiation pressure and relativistic calculations could be added, or the numerical integration could be improved via a more efficient algorithm.

In total, however, I am very happy with the end result. I will certainly use it myself in the future, I hope others will too.