A Simulation of Orbital Bodies

Computer Science NEA

Name: Cristian Stefan IONEL
Candidate Number: 1113

Centre Name: Barton Peveril College
Centre Number: 58231

Contents

Contents 2

1 Analysis 3

1.1 Statement of Problem 3

1.2 Background 3

1.3 End User 3

1.4 Initial Research 4

1.4.1 Existing, similar programs 4

1.4.2 Potential algorithms 6

1.4.3 First interview 6

1.4.4 Key components from First interview 7

1.5 Further Research 7

1.5.1 Prototype 7

Code, with explanation highlighted: 8

1.5.2 Prototype Results 14

1.5.3 Second interview 18

1.6 Objectives 19

2 Design 20

Main: 20

User input: 21

Pause: 23

Import/Export/Load: 23

Trails: 24

Setting up trails/Cleaning trails: 25

Display Planets: 26

Real Distance to On-Screen: 26

Centred Screen: 28

Display Planets Flow-chart 29

Adding Planet: 30

3 Technical Solution 32

Contents page for important code 32

4 Testing 35

5 Evaluation 40

Overall 40

User Feedback 40

Objectives Review 41

6 Code 42

1 Analysis

1.1 Statement of Problem

The concept of orbits and the movement of bodies in space is a concept hard to grasp.
It takes what we know of movement; you constantly need to push something to move, what comes up must come down; and removes them. In space once you start moving in a direction with a speed you will continue going at that same speed until you accelerate in another direction. Everything accelerates from the pull of a planet at the same rate, regardless of mass due to a lack of atmosphere. A hammer on the moon dropped from 10 metres off the surface will hit the ground no sooner than a feather.
More than just being hard to grasp, this kind of environment is not something that most of us will ever experience, making it infeasible to experiment in such an environment to better understand it.

1.2 Background

The concept of movement outside of Earth's pull has been something that we as humans do not intuitively understand. The first successful attempt at a gravitational simulation was done by Swede Erik Holmberg in 1941, by using the equations of motion (SUVAT) and light bulbs with different light intensities as a representation of their gravitational pull and photocells measuring the intensity of the light, the result from the cells being used as the gravitational strength at that place. Later simulations also implemented general relativity, which is necessary if the simulation is between galaxies, which are 5.8*10^18 miles apart on average.

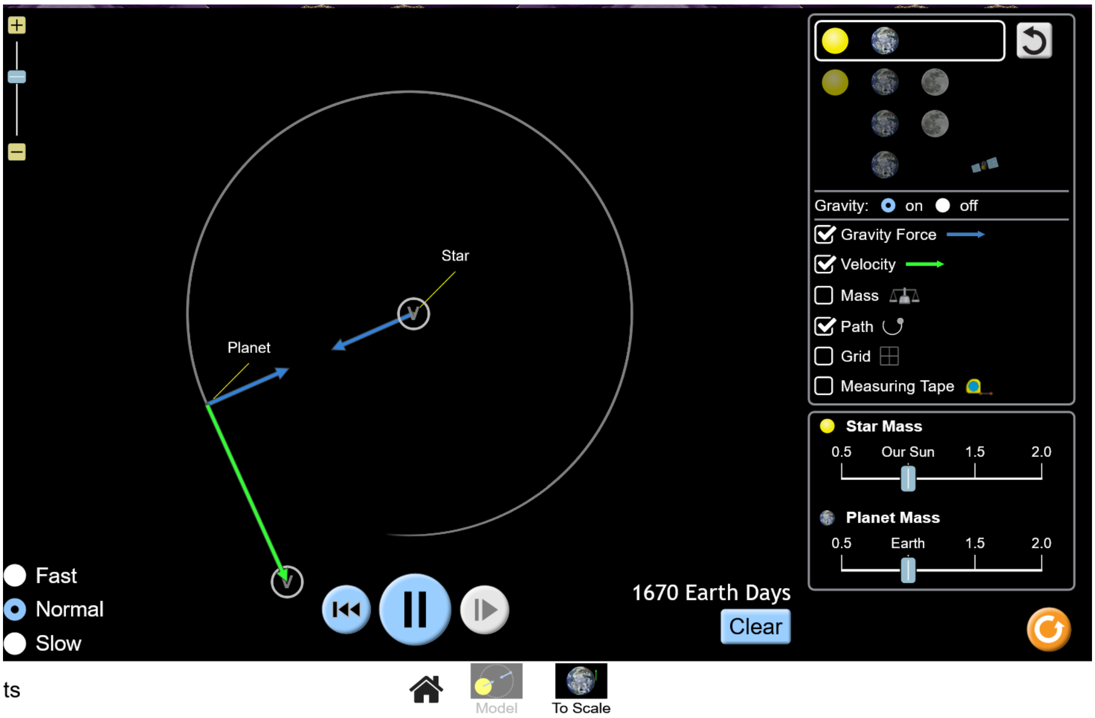
The problem of understanding the movement of planets or other astronomical things is caused mainly by two things: 1) by the change from having Earth as a complete reference frame (having everything move with it and falling towards it), to having no reference frame and 2) The Earths' atmosphere making different things fall at different speeds due to their mass. This leads to people interested in or studying orbits to struggle with the concept.

1.3 End User

I have a friend, Alex, who is struggling with understanding orbits in general but has a good grasp of physics. He will be the main user of this program.

1.4 Initial Research

1.4.1 Existing, similar programs

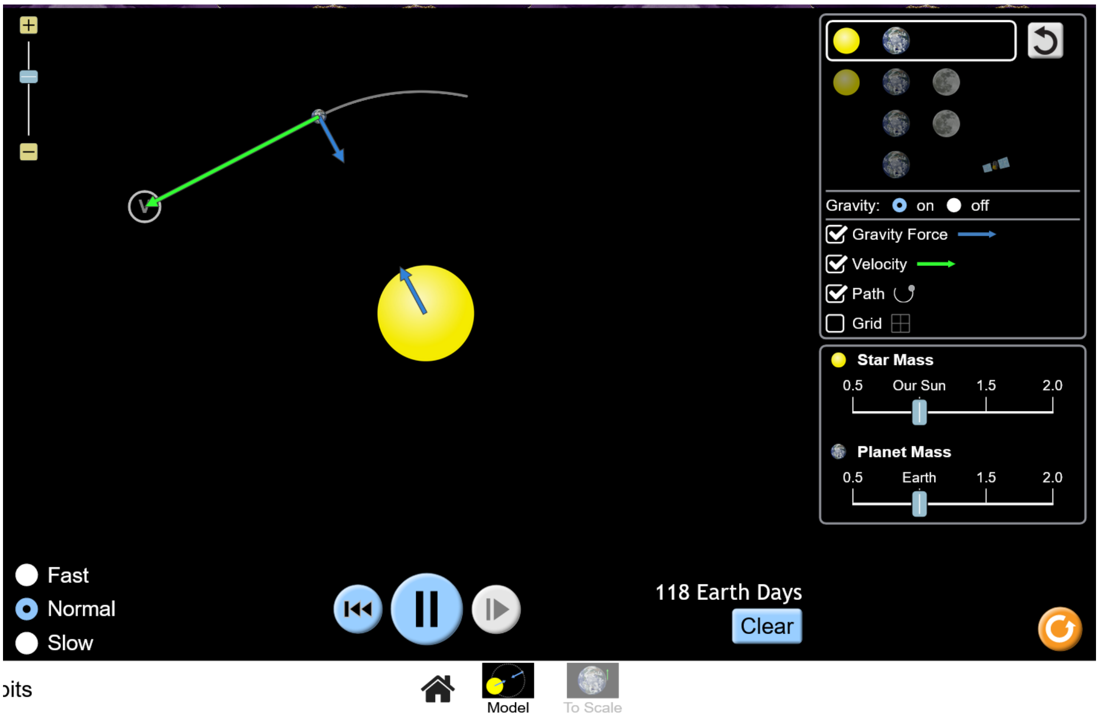


Gravity: ● on ○ off
☑ Gravity Force
☑ Velocity
□ Mass
☑ Path
□ Grid
□ Measuring Tape

Star Mass: 0.5, Our Sun, 1.5, 2.0
Planet Mass: 0.5, Earth, 1.5, 2.0

○ Fast
● Normal
○ Slow

1670 Earth Days



Gravity: ● on ○ off
☑ Gravity Force
☑ Velocity
☑ Path
□ Grid

Star Mass: 0.5, Our Sun, 1.5, 2.0
Planet Mass: 0.5, Earth, 1.5, 2.0

○ Fast
● Normal
○ Slow

118 Earth Days

https://phet.colorado.edu/sims/html/gravity-and-orbits/latest/gravity-and-orbits_en.html

Pros:

Both to-scale and model versions

Allows for forces and path to be shown

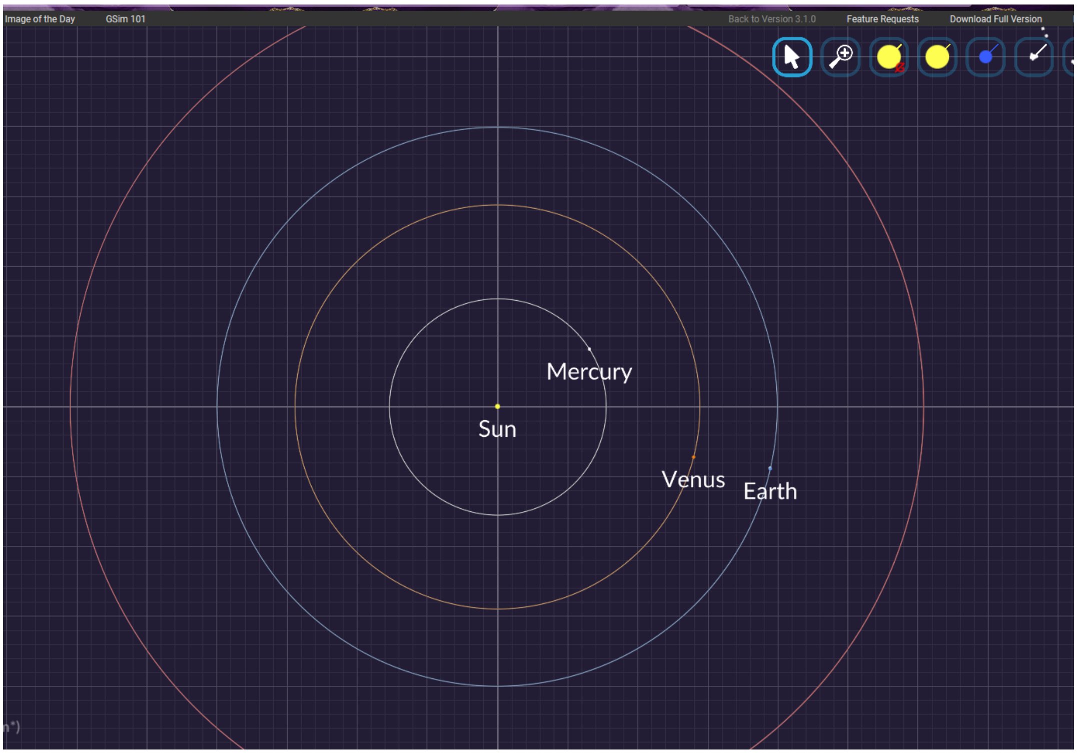
Simple sliders to adjust settings

Lack of features makes it more beginner friendly

Cons:

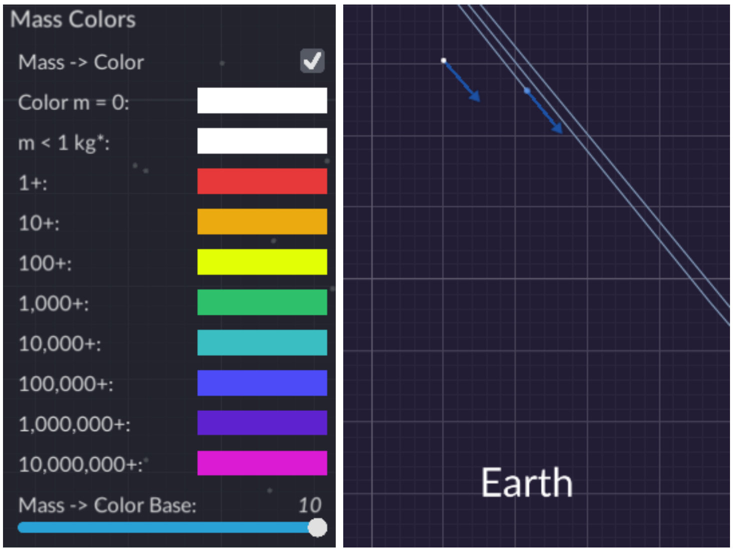
Lack of features e.g. adding more celestial bodies, no full solar system and limited mass adjustment means that the amount that can be learnt from this program is limited

https://www.testtubegames.com/gravity.html



Pros:

Clear visuals, allows for custom colours or for planets to have their colour relative to their mass as seen in this screenshot:



Mass Colors

Mass -> Color	☑
Color m = 0:	⬜
m < 1 kg*:	⬜
1+:	🟥
10+:	🟧
100+:	🟨
1,000+:	🟩
10,000+:	🟦
100,000+:	🟦
1,000,000+:	🟪
10,000,000+:	🟪
Mass -> Color Base:	10

Also allows for forces and paths to be shown

Can load from pre-made systems or store custom ones

Can click and drag objects around

Cons:

Tools are somewhat difficult to use

1.4.2 Potential algorithms

Using F = ma, the gravitational constant and vectors and doing it procedurally is one option. It would mean that I cannot do predictions for where objects will be. However, it would be relatively light on the computer and would be able to handle any amount of objects.
I could not find any other reasonable alternatives, as they would either implement general relativity which would be useless on a solar scale or use equations far too complicated for simulations with low numbers of bodies.

- So, I will have the simulation run continuously, one tick at a time, where a tick can represent any time interval (1second, 1min, etc.).
- Each tick the program will check the gravitational attraction of the planets on one another in m/s^2 (how much the gravitational attraction causes the planets to accelerate)
- Then, the program will update the velocities of the planets based on their acceleration times the time each tick represents.(E.g. if a planet accelerates at 5 m/s^2 for 10 seconds its velocity will change by 50 meters per second)
- Finally, the program updates the position of each planet based on the velocity multiplied by the time each tick represents again.
- Program goes to next tick

This is the simulation part of the program.

1.4.3 First interview

Q1) How important would the visual aspect of the program be for you?
I think that graphics in general are an important aspect, being able to view and differentiate things from one another is a strong feature of any program.

Q2) What options have you tried in the past? Any features that were especially useful?
Some time ago, one of my teachers showed us his orbit simulation program he had made. I played around with it for a few days, but overall it was very complicated to use and hard to understand what did what on it. It wasn't very well displayed and I I often found myself annoyed by it. Other than that I have mostly just used online simulations I found on google.

Q3) How many objects would you consider as the standard number for a simulation?
Around 2-3 objects I think should be good.

Q3) How important would it be for you to be able to save and load the configuration of systems, made by you and some pre-sets?
This is a very important feature in my opinion. Reopening a program and having to redo the configuration from scratch is a loathsome process and during the journey I might actually not understand what happened and how I got there (that is, of course, in case I made a mistake). Being able to save and load configurations while making things a lot easier to work with and understand.

Q4) What is the main problem in trying to understand orbits; the way that gravity still interacts with the body, like the concept of an orbit in itself or the way that changes in the body's velocity affects the shape of its orbit?
What attribute does what to an orbit, how does it affect it? Even after getting an explanation about orbits and playing around with programs, I just found it more and more confusing and complex.

Q5) Do you then think that being able to change the velocity of objects gradually would be important?
Yes, this feature does actually seem important to me. It would help me understand changes in orbits more easily as things are not only using constant values. The more changes I am able to make and test out different things, the easier I am able to learn and understand.

Q6) What about the importance of being able to directly add new objects in an orbit of a desired shape?
Same as before, I think it's very important that I can, within limitations, create my own scenario using new objects that would in fact prove beneficial to understanding orbits more easily.

1.4.4 Key components from First interview

- Clear visuals
- Easy to add objects
- Customisable objects
- Saving/Loading Systems
- Capability of adding new objects
- Objects should have different colours at least to be able to be easily differentiated

1.5 Further Research

1.5.1 Prototype

For a prototype it would be best to handle the simulation side of the program first, as in the calculations of the movement of bodies, leaving out the display and user input.
This will be good to check if the potential algorithmI thought of will work, or if the time in between ticks where the planets move effectively without gravity will cause problems.
It will also be useful to see how much time can be simulated per tick without a significant loss in accuracy.

Since I intend to make the end program object-oriented, I will make the prototype using functions and procedures wherever possible to make it easier later on to adapt the code to classes.
I will write the program in C# as it can handle classes easily and I already know how to use it.
Notes added as comments so I can still copy and paste the code, notes are in **BOLD**

Code, with explanation highlighted:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Sencond_attempt__different_equasions
{
    class Program
    {
```

// The structure “Body” will be used to store the (x, y) coordinates, velocity vector, mass and name of the bodies within the simulation. These will be stored as a Body[] array in the main, though in the final program a list will be used instead of an array to allow for a changing number of bodies..

```
        struct Body
        {
            public double posX;
            public double posY;
            public double xVelocity;
            public double yVelocity;
            public double mass;
            public string name;
        }
```

// To be able to add new bodies to an array of “Body” I will use a function that makes a temporary Body named temp with the desired attributes and then returns that to be placed in the array.

```
        static Body addObject(double X, double Y, double xVel, double yVel, double mas, string na)
        {
            Body temp;
            temp.posX = X;
            temp.posY = Y;
            temp.xVelocity = xVel;
            temp.yVelocity = yVel;
            temp.mass = mas;
            temp.name = na;
            return temp;
        }
```

// makePositive takes a reference of a number and if it is negative, it multiplies it by -1 to make it positive (since it is a reference the original variable will also be changed), it will be needed to make sure there aren’t negative distances.

```
        static void makePositive(ref double num)
        {
            if (num < 0)
            {
```

```
            num = num * -1;
        }
    }
```

// Using a^2+b^2=h^2 to find the distance between two objects, where a is the difference in the x axis and b is the difference in the y axis. Since I cannot do a double to the power of two I instead need to multiply it by itself

```
    static double distanceCalc(double x1, double y1, double x2, double y2)
    {
        double distance = Math.Sqrt((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2));
        makePositive(ref distance);
        return distance;
    }
```

// Taking the x and y coordinates of two objects, you can find the vector from one to the other. This function finds the vector from (x1, y1) to (x2, y2).

```
    static double[] vectorClac(double x1, double y1, double x2, double y2)
    {
        double[] vector = new double[2];
        vector[0] = x2 - x1;
        vector[1] = y2 - y1;
        return vector;
    }
```

// Using gravitational pull = mass1*mass2*gravitationalConstant/distance^2, mass1 and mass2 representing the mass of the two objects pulling on each other and the gravitational constant being an arbitrary number, used to find how much gravitational pull an object has

```
    static double gravPullForceCalc(double mass1, double mass2, double distance)
    {
        double gravConstant = 6.67e-11;

        double gravitationalPull = (mass1*mass2) / (distance*distance);
        //gravitational pull = gravitational constant * (mass1*mass2) / distance^2

        gravitationalPull = gravConstant * gravitationalPull;
        //adding the gravitational constant

        return gravitationalPull;
    }
```

// This function uses two for loops to compare every body from the array two at a time. The function uses distanceCalc and vectorCalc functions to find the distance between two objects and the vector with length one that points from body to the other. Then it uses the gravPullForceCalc function to find the gravitational pull and a for loop to return how much the objects have accelerated along each axis. The for loop used when updating the acceleration allows for implementation of a z axis.

```
    static double[,] accelerationCalc(Body[] things)
    {
        double[,] accelerations = new double[things.Length, 2];
        double distance;
```

```
            double gravPullForce;
            double[] unitVector = new double[2];

            for (int i = 0; i < things.Length; i++)
            {
                for (int j = 0; j < things.Length; j++)
                {
                    if (i != j)
                    {

                        distance = distanceCalc(things[i].posX, things[i].posY, things[j].posX,
things[j].posY);
                        unitVector = vectorClac(things[i].posX, things[i].posY, things[j].posX,
things[j].posY);

                        unitVector[0] = unitVector[0] / distance;
                        unitVector[1] = unitVector[1] / distance;

                        gravPullForce = gravPullForceCalc(things[i].mass, things[j].mass, distance);

                        //accelerationVector = 1/m * Sum(force vector)
                        //array accelerations used as accelerationVector, will slowly add 1/m * force
vector onto it

                        for (int l = 0; l < unitVector.Length; l++)
                        {
                            accelerations[i, l] += (unitVector[l]*gravPullForce)/things[i].mass;
                        }

                    }
                }
            }

            return accelerations;
        }
```

// velocityChange uses the acceleration along the x and y axis of every body and the time simulated per tick to update the velocities of the bodies in the array. The timePerTick value is necessary because the acceleration is measured in m/s^2, as in how much the velocity will change in one second, however if we simulate multiple sends at a time we need to multiply for that, e.g. for a minute we multiply by 60

```
        static Body[] velocityChange(Body[] things, double[,] accelerations, double timePerTick)
        {
            for (int i = 0; i < things.Length; i++)
            {
                things[i].xVelocity = things[i].xVelocity + accelerations[i, 0] * timePerTick;
                //x axis velocity += x axis acceleration
                things[i].yVelocity = things[i].yVelocity + accelerations[i, 1] * timePerTick;
                //y axis velocity += y axis acceleration
            }
            return things;
        }
```

// A function to update the position of the bodies in the array. Since the velocity of the bodies is already in the body structure all that the function needs is the timePerTick. Again, this is necessary because the velocity is in metres per second but we may simulate multiple seconds at a time, so we need to multiply for that.

```
        static Body[] positionChange(Body[] things, double timePerTick)
        {
            for (int i = 0; i < things.Length; i++)
            {
                things[i].posX = things[i].posX + things[i].xVelocity * timePerTick;
                //x axis position += x axis velocity
                things[i].posY = things[i].posY + things[i].yVelocity * timePerTick;
                //y axis position += y axis velocity
            }
            return things;
        }
```

// A simple function that displays the position of the bodies in the console as (x, y) coordinates and relative to their orbit radius. Since this prototype simulates the solar system, it has the orbit radii inbuilt. It also uses a series of if statements to change the colours used for the text representing the planets which makes it easier to differentiate the bodies apart. The movement relative to radius is calculated by dividing its current x or y coordinates and dividing them by the radius of its orbit.

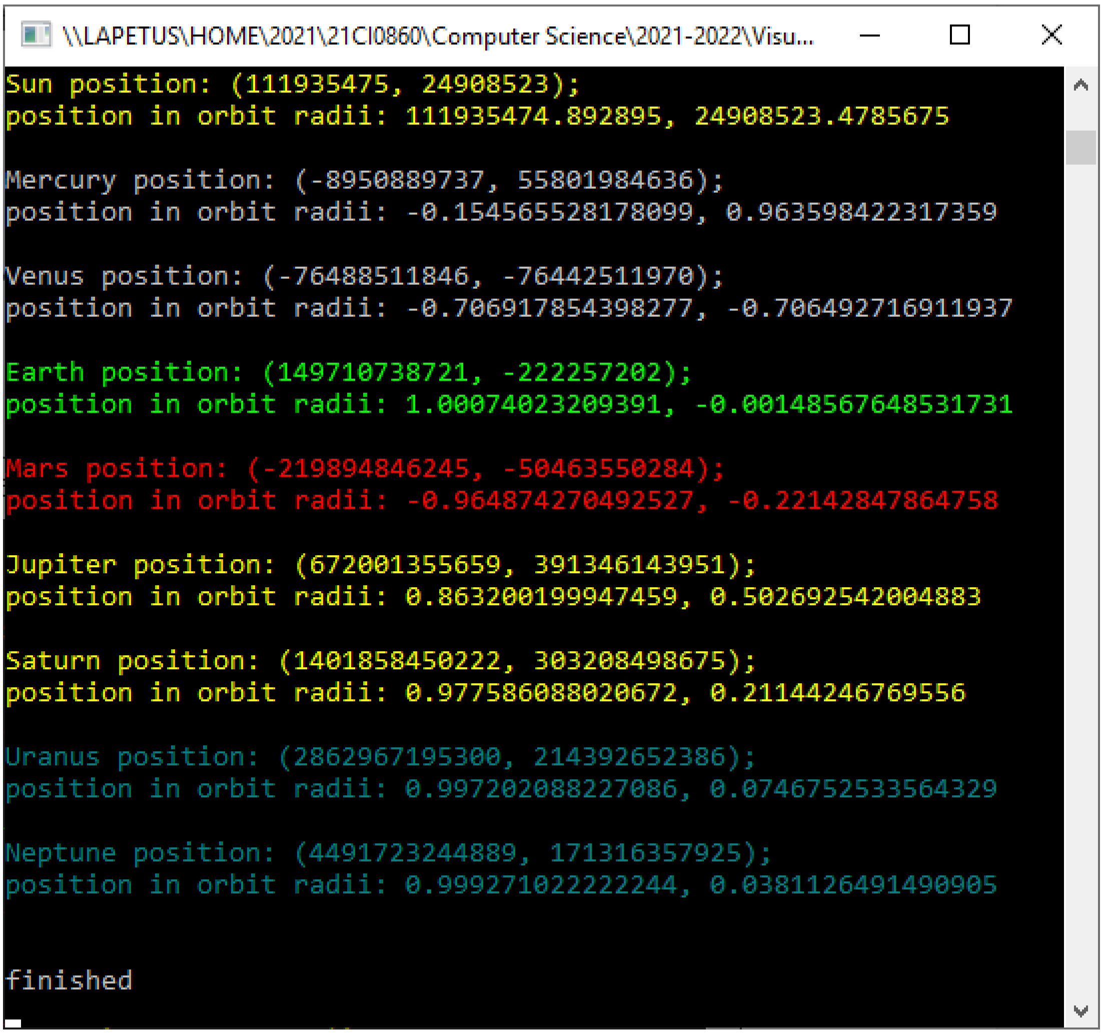
```
        static void displayPositions(Body[] things)
        {
            double[] orbitRadii = new double[] {1, 5.791e10, 1.082e11 , 1.496e11 , 2.279e11, 7.785e11 , 1.434e12 , 2.871e12 , 4.495e12 };

            for (int i = 0; i < things.Length; i++)
            {
                if (i == 0)
                {
                    Console.ForegroundColor = ConsoleColor.Yellow;
                }
                else if (i == 3)
                {
                    Console.ForegroundColor = ConsoleColor.Green;
                }
                else if(i == 4)
                {
                    Console.ForegroundColor = ConsoleColor.Red;
                }
                else if (i == 5 || i == 6)
                {
                    Console.ForegroundColor = ConsoleColor.Yellow;
                }
                else if (i > 6)
                {
                    Console.ForegroundColor = ConsoleColor.DarkCyan;
                }
                Console.Write(things[i].name + " position: (" + Math.Round(things[i].posX) + ", " +
Math.Round(things[i].posY) + ");\nposition in orbit radii: " + things[i].posX/ orbitRadii[i] + ", " +
things[i].posY/ orbitRadii[i] + "\n");
                Console.WriteLine();
                Console.ForegroundColor = ConsoleColor.Gray;
            }
            Console.WriteLine();
        }
            // In the main I set up the Body array to have 9 bodies, the 8 planets in the solar
system and the sun. The bodies are added using the addObject function.
        static void Main(string[] args)
        {

            Body[] things = new Body[9];

            things[0] = addObject(0, 0, 0, 0, 1.989e30, "Sun");

            things[1] = addObject(5.791e10, 0, 0, 47.36e3, 3.285e23, "Mercury");
            things[2] = addObject(1.082e11, 0, 0, 35.02e3, 4.867e24, "Venus");
            things[3] = addObject(1.496e11, 0, 0, 29.78e3, 5.972e24, "Earth");
            things[4] = addObject(2.279e11, 0, 0, 24.07e3, 6.39e23, "Mars");

            things[5] = addObject(7.785e11, 0, 0, 13e3, 1.898e27, "Jupiter");
            things[6] = addObject(1.434e12, 0, 0, 9.68e3, 5.683e26, "Saturn");
            things[7] = addObject(2.871e12, 0, 0, 6.80e3, 8.681e25, "Uranus");
            things[8] = addObject(4.495e12, 0, 0, 5.43e3, 1.024e26, "Neptune");

            double[,] acceleration;

            double percentDone = 0;

            double time;
            double endTime = 60*60*24*365.25; // time to end set to 1 year
            double timePerTickInSec = 1; // time simulated per tick, currently set to 1 second

            //loop that continues until the time simulated reaches endTime, increasing time
each time by the time per tick variable, making it go one tick at a time
            for (time = 1; time <= endTime; time += timePerTickInSec)
            {
                acceleration = accelerationCalc(things);
                //acceleration 2d array stores the accelerations in x and y of each planet
relative to the things array. So if things[0] refers to the sun, acceleration[0, 0] refers to
the acceleration of the sun in the X axis. Acceleration [0, 1] would then be the
acceleration in the Y axis.

                things = velocityChange(things, acceleration, timePerTickInSec);
                //update the velocity of objects using the acceleration found

                things = positionChange(things, timePerTickInSec);
                //update the velocity of objects using the velocities

                if (time/endTime >= percentDone)
                {
                    //checks if the time completed has increased by 10% and if so, it displays
the position of all the planets
                    percentDone += 0.1 ;
                    displayPositions(things);
                }
            }
            //after the simulation has completed the amount of time
            displayPositions(things);
            Console.WriteLine("finished");
            Console.ReadKey();
        }
    }
}
```

1.5.2 Prototype Results

All bodies start the simulation to the right of the sun and orbit anti-clockwise.
I will run the program with different values for “timePerTickInSec”, checking for one second, per tick, one minute, one hour, and one day to check how much time it takes to simulate one year of the solar system and how much information is lost from skipping more from tick to tick.
Since the planets do not move in a perfect centre around the sun (planets do not return to their exact original position after going around the sun), and the sun is not supposed to be at the centre of the solar system I will assume that when time per tick is one second the simulation is perfect.
I will write the deviation that each time per tick setting has from the one second per tick setting as a percentage.

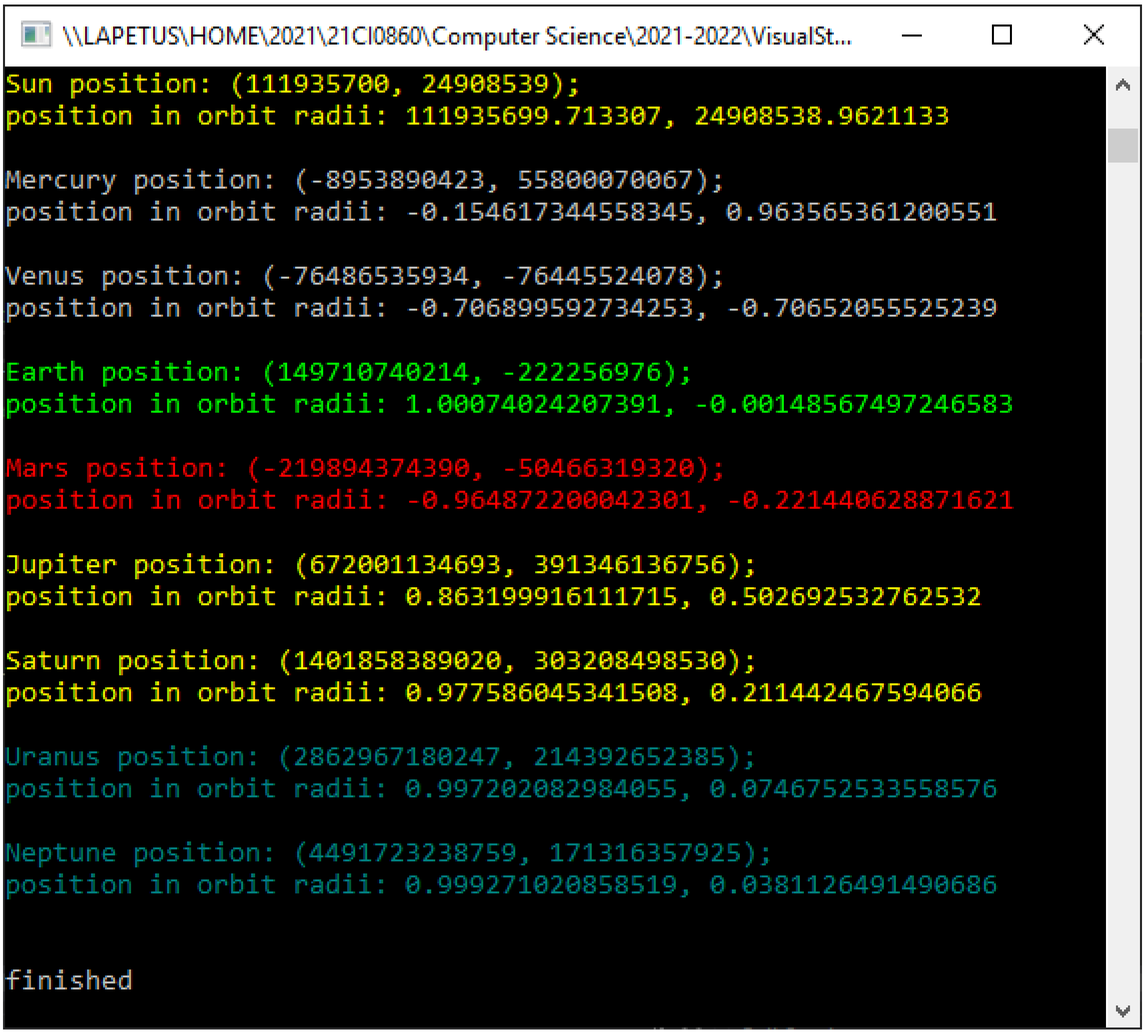
The screenshots below show the final position of each planet and th position relative to its radius.

Second: 3 min 7 sec for 1 year (assuming to be perfectly accurate)


\\LAPETUS\HOME\2021\21CI0860\Computer Science\2021-2022\Visu...

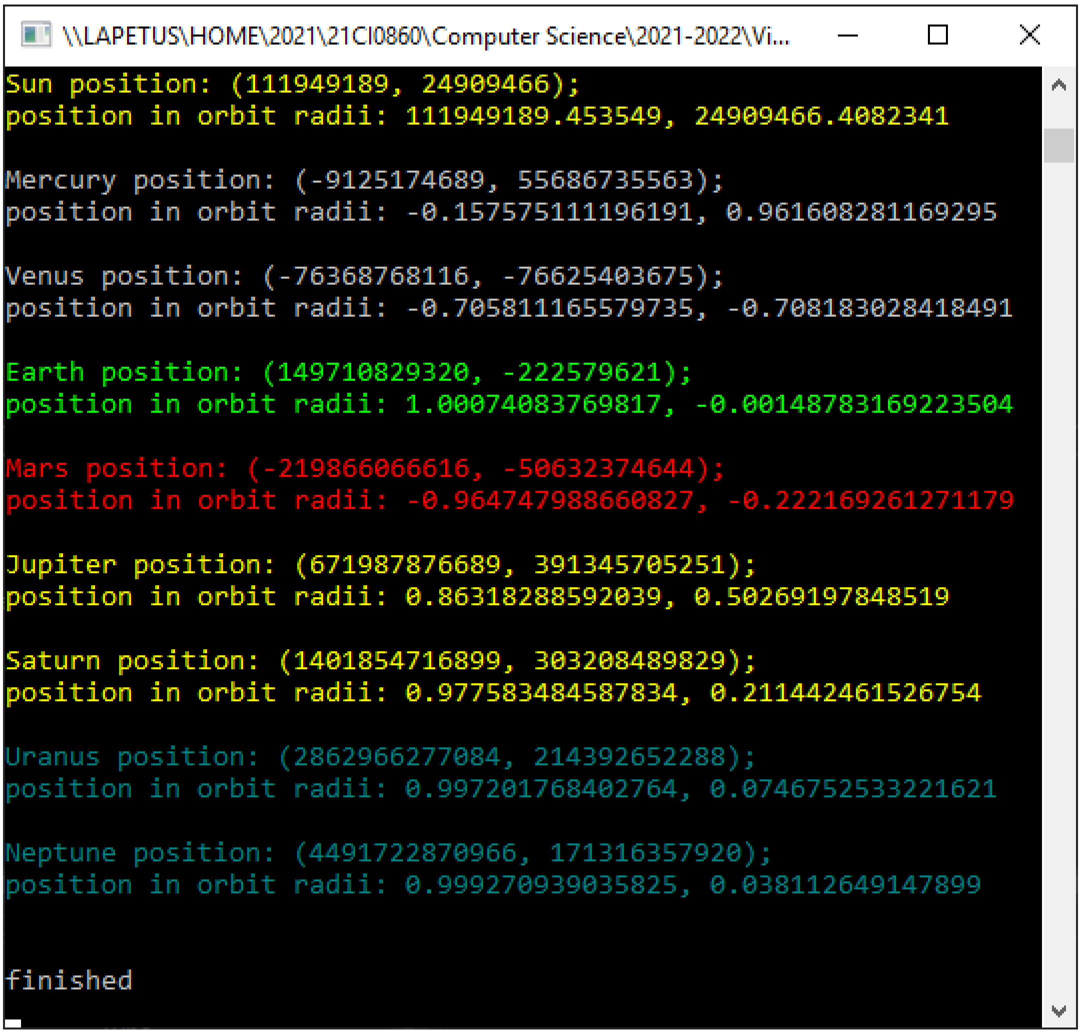
```
Sun position: (111935475, 24908523);
position in orbit radii: 111935474.892895, 24908523.4785675

Mercury position: (-8950889737, 55801984636);
position in orbit radii: -0.154565528178099, 0.963598422317359

Venus position: (-76488511846, -76442511970);
position in orbit radii: -0.706917854398277, -0.706492716911937

Earth position: (149710738721, -222257202);
position in orbit radii: 1.00074023209391, -0.00148567648531731

Mars position: (-219894846245, -50463550284);
position in orbit radii: -0.964874270492527, -0.22142847864758

Jupiter position: (672001355659, 391346143951);
position in orbit radii: 0.863200199947459, 0.50269254200488З

Saturn position: (1401858450222, 303208498675);
position in orbit radii: 0.977586088020672, 0.21144246769556

Uranus position: (2862967195300, 214392652386);
position in orbit radii: 0.997202088227086, 0.0746752533564329

Neptune position: (4491723244889, 171316357925);
position in orbit radii: 0.999271022222244, 0.0381126491490905


finished
```

Minute: 3 sec 8.968e-7% error, too much movement


\\LAPETUS\HOME\2021\21CI0860\Computer Science\2021-2022\VisualSt...

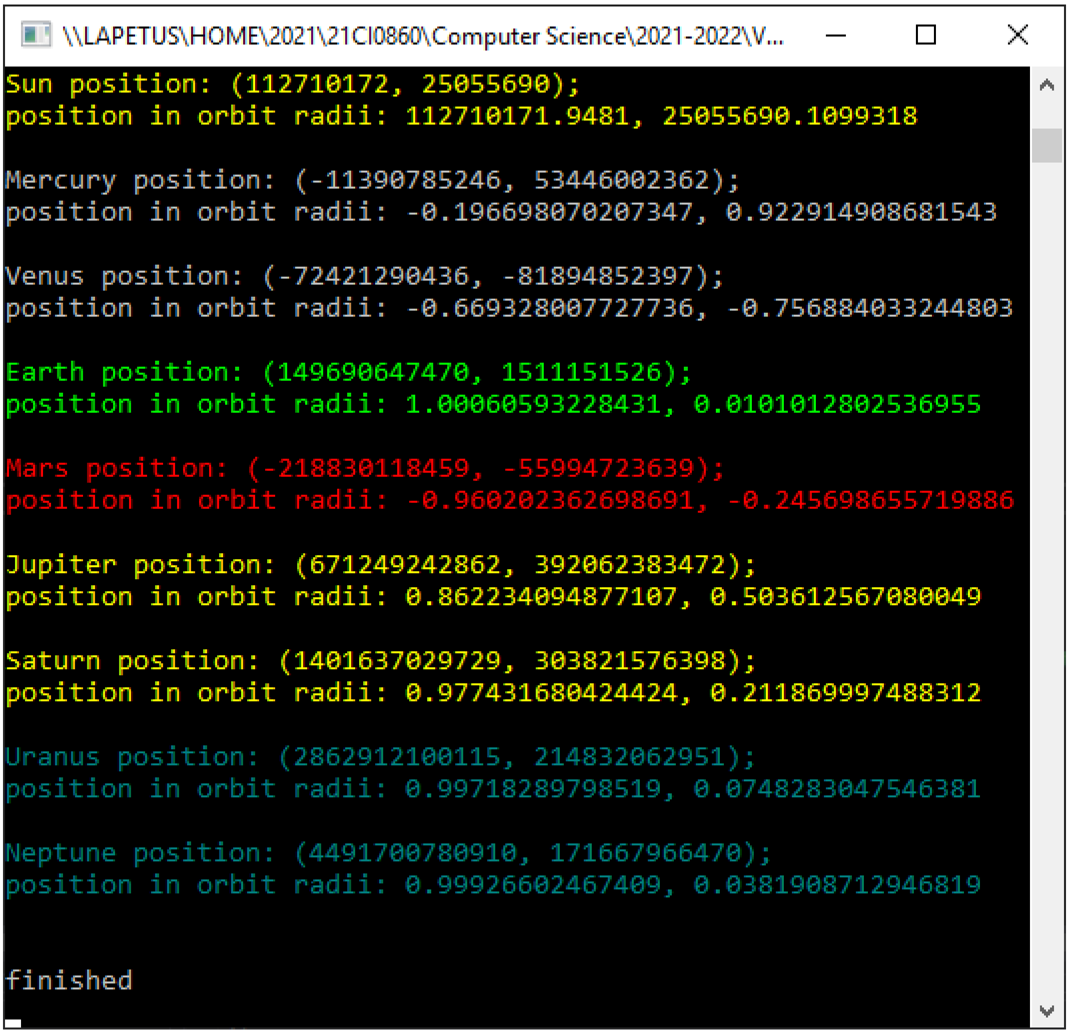
```
Sun position: (111935700, 24908539);
position in orbit radii: 111935699.713307, 24908538.9621133

Mercury position: (-8953890423, 55800070067);
position in orbit radii: -0.154617344558345, 0.963565361200551

Venus position: (-76486535934, -76445524078);
position in orbit radii: -0.706899592734253, -0.70652055525239

Earth position: (149710740214, -222256976);
position in orbit radii: 1.00074024207391, -0.00148567497246583

Mars position: (-219894374390, -50466319320);
position in orbit radii: -0.964872200042301, -0.221440628871621

Jupiter position: (672001134693, 391346136756);
position in orbit radii: 0.863199916111715, 0.502692532762532

Saturn position: (1401858389020, 303208498530);
position in orbit radii: 0.977586045341508, 0.211442467594066

Uranus position: (2862967180247, 214392652385);
position in orbit radii: 0.997202082984055, 0.0746752533558576

Neptune position: (4491723238759, 171316357925);
position in orbit radii: 0.999271020858519, 0.0381126491490686


finished
```

Hour: less than 1 sec 6.051e-5% error, too much movement


\\LAPETUS\HOME\2021\21CI0860\Computer Science\2021-2022\Vi...

```
Sun position: (111949189, 24909466);
position in orbit radii: 111949189.453549, 24909466.4082341

Mercury position: (-9125174689, 55686735563);
position in orbit radii: -0.157575111196191, 0.961608281169295

Venus position: (-76368768116, -76625403675);
position in orbit radii: -0.705811165579735, -0.708183028418491

Earth position: (149710829320, -222579621);
position in orbit radii: 1.00074083769817, -0.00148783169223504

Mars position: (-219866066616, -50632374644);
position in orbit radii: -0.964747988660827, -0.222169261271179

Jupiter position: (671987876689, 391345705251);
position in orbit radii: 0.86318288592039, 0.50269197848519

Saturn position: (1401854716899, 303208489829);
position in orbit radii: 0.977583484587834, 0.211442461526754

Uranus position: (2862966277084, 214392652288);
position in orbit radii: 0.997201768402764, 0.0746752533221621

Neptune position: (4491722870966, 171316357920);
position in orbit radii: 0.999270939035825, 0.038112649147899


finished
```

Day: less than 1 sec 0.0134% error, too little movement


\\LAPETUS\HOME\2021\21CI0860\Computer Science\2021-2022\V...

```
Sun position: (112710172, 25055690);
position in orbit radii: 112710171.9481, 25055690.1099318

Mercury position: (-11390785246, 53446002362);
position in orbit radii: -0.196698070207347, 0.922914908681543

Venus position: (-72421290436, -81894852397);
position in orbit radii: -0.669328007727736, -0.756884033244803

Earth position: (149690647470, 1511151526);
position in orbit radii: 1.00060593228431, 0.0101012802536955

Mars position: (-218830118459, -55994723639);
position in orbit radii: -0.960202362698691, -0.245698655719886

Jupiter position: (671249242862, 392062383472);
position in orbit radii: 0.862234094877107, 0.503612567080049

Saturn position: (1401637029729, 303821576398);
position in orbit radii: 0.977431680424424, 0.211869997488312

Uranus position: (2862912100115, 214832062951);
position in orbit radii: 0.99718289798519, 0.0748283047546381

Neptune position: (4491700780910, 171667966470);
position in orbit radii: 0.99926602467409, 0.0381908712946819


finished
```

The algorithm works, and as predicted the higher the time per tick the lower the accuracy.

Again I will assume that when simulating one second per tick shows the correct distance the earth travelled, if the sun was not orbiting the centre of mass at the start of the simulation and had to settle into place.
The error value I attributed to the different simulation times I found by doing that simulations' earth x position divided by the x position in the simulation with 1 second per tick, which would give me a percentage value difference which I will interpret as the error value.

Having each tick of the program simulate one hour at a time seems reasonable, as the program would need to simulate another 16526 years for the program to reach an error margin of 1% when simulating 9 bodies.
It is important for the time intervals of the program to be as small as possible because the number of calculations done scales with each object added scales exponentially, slowing the program down considerably. From the interview I know that 3 objects would be about as many as would be necessary, however if the program cannot handle the 9 objects of the solar system then it would be a bit problematic. Having the program automatically change tickspersecond to allow for the program to speed up the simulation but have a warning for the user regarding the loss of accuracy may be a good idea.

1.5.3 Second interview

1) Do you think that a 0.0006051% error in the simulation over the span of one year is an accurate enough representation for the purposes of learning how planets interact?

   Yes, it should be accurate enough as there isn't much that I could do with this program that would require it to be accurately simulated.

2) How important would a debug function be to you?

   It might be of some use to allow me to check if there was an error in the program but it depends on how well the debugger works.

3) Would a rewind function, where the simulation runs backwards to rewind be something useful?

   Yeah, it would allow me to run the program and then compare it to what would happen with different settings.

4) In the case that two objects would you prefer for their mass and velocities to just be added so that the program remains simple or would you prefer for some debris to be generated, taking into consideration that the number of calculations that have to be made grows exponentially with each object added?

   It might be nice to see that added as an optional setting, adding debris to a collision wouldn't change the outcome much more than that adding method.

5) The program currently uses Newtonian physics that does not take into account the expansion of the universe, the non-instant speed of gravity, relativity affecting objects moving extremely quickly and other phenomena that occur on extremely large scales. This means that it cannot accurately simulate the interaction of bodies on a galactic scale (100`000 light years). Would the implementation of more complicated equations that do take this into account be important?

   This might also be a cool setting to see.

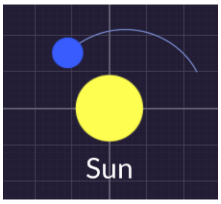
1.6 Objectives

Summary:
Have the program simulate the movement of planets to a reasonable degree of accuracy and display the planets.

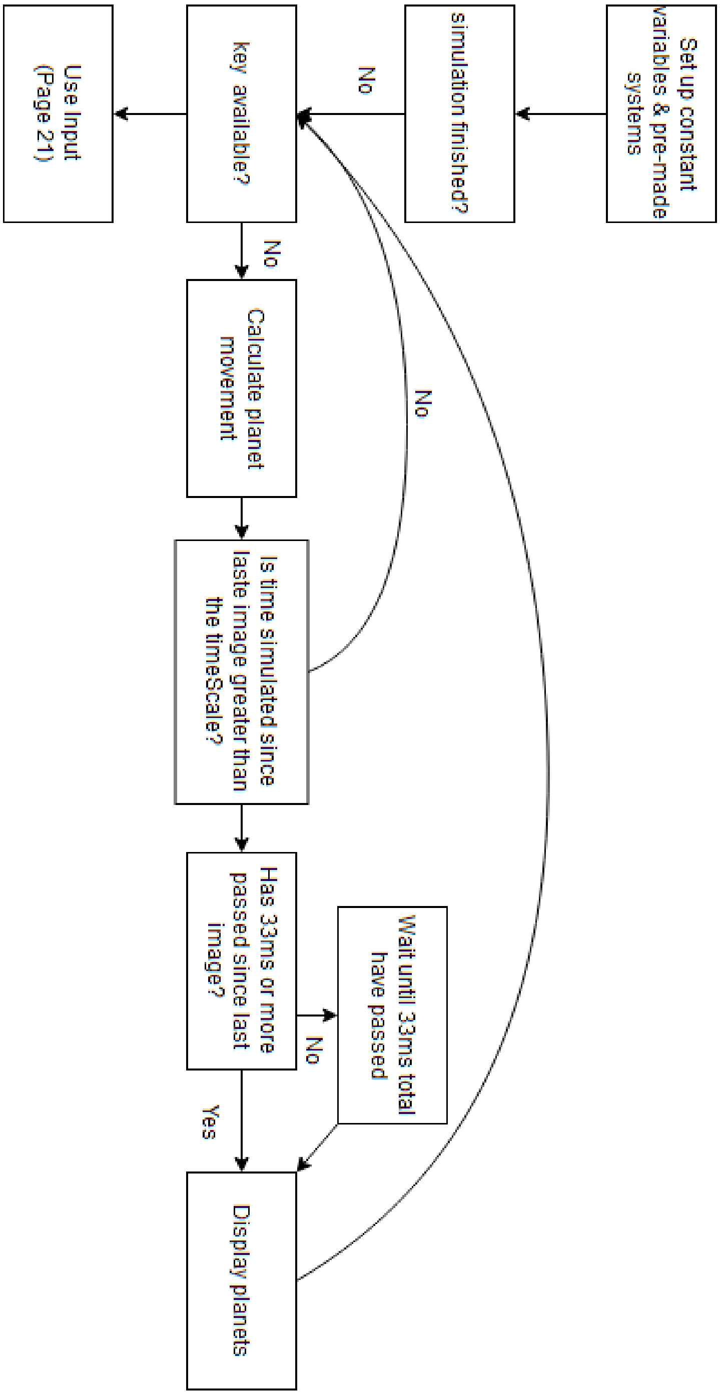
1. The Simulation Itself
   1.1. Be able to simulate the orbits of bodies with reasonable accuracy and speed
   1.2. Be able to simulate at different speeds (e.g. 1minute simulated every second; 1 week simulated every second etc.)
   1.3. When the simulation speed changes the accuracy of the simulation should lower to prevent the program from stutering
   1.4. When two objects overlap they should merge, with the resultant object having the mass and velocity as the sum of two of the two component objects'

2. The User Input
   2.1. Allow the user to add planets in desired orbits
   2.2. Allow the user to zoom in and out and move the point of view
   2.3. Allow for pre-set systems to be loaded
   2.4. Allow for users to save the systems they made

3. Display
   3.1. The user should be able to change the units used to or from kilograms, earth masses or solar masses
   3.2. Have the option to show the forces acting between objects
   3.3. When hovering the cursor over a setting or button in general the program should display a short description of what it does
   3.4. Have the option to draw a line to show the movement of the object, a trail that disappears over time so it does not clutter up the screen such as in this screenshot of one of the program I looked into:



2 Design


Set up constant variables & pre-made systems
simulation finished?
No
key available?
Use Input (Page 21)
No
Calculate planet movement
No
Is time simulated since laste image greater than the timeScale?
Has 33ms or more passed since last image?
No
Wait until 33ms total have passed
Yes
Display planets

Main:

The program will start with the inner planets of the solar system initially.

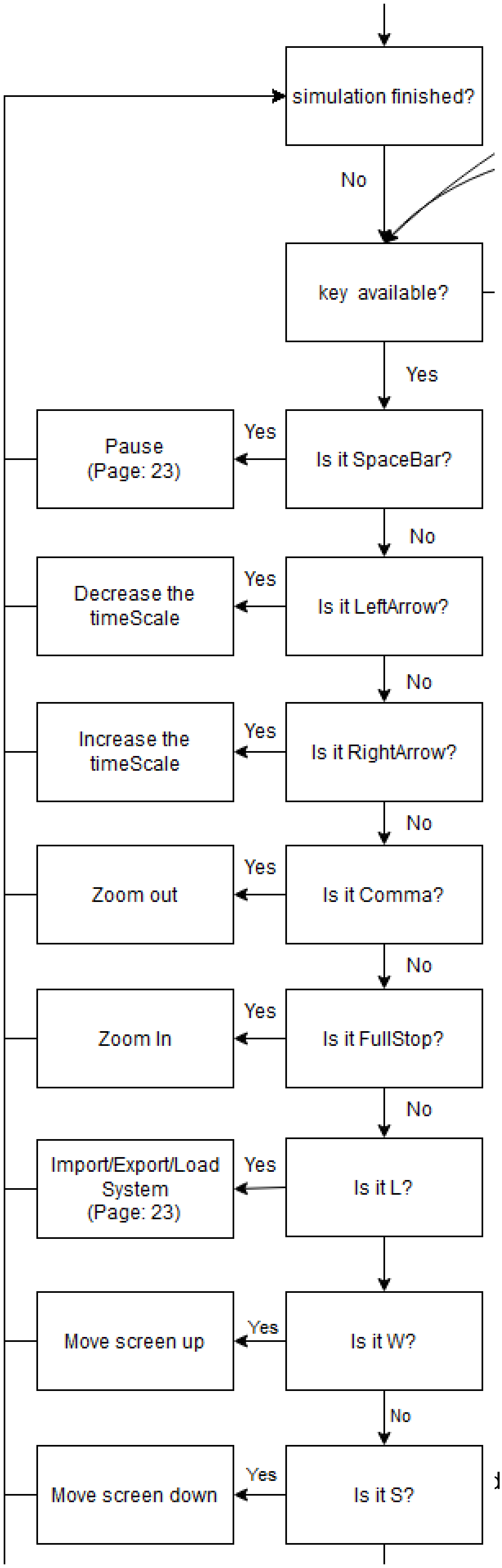
The program will work by having in the main two loops.
The outer loop happens indefinitely and holds the if/else statements for the user input as well as the inner loop.
The inner loop happens as long as no key is available. It is a continuous loop where each time around it calculates how much the planets move based on how much time is calculated per loop (**timePerTick**).
The inner loop uses the calculations described on Page 6 1.4.2; copied from the prototype, Page 7-13.

Everytime the inner loop goes around it also checks how much time has been simulated since an image has last been displayed. If it is greater or equal than how much time is supposed to pass each frame then it checks how much real time has passed since the last image. (**timeScale**).

If 33ms have passed since the last image then it displays a new image of the planets. If not then it waits for the total time spent since the last image to be 33ms. This is to keep the frames per second around 30.
It also prevents the program from suddenly showing many frames one after the other in the case that the program calculates the movement of the planets faster than usual.

Display planets on page 26

User input:

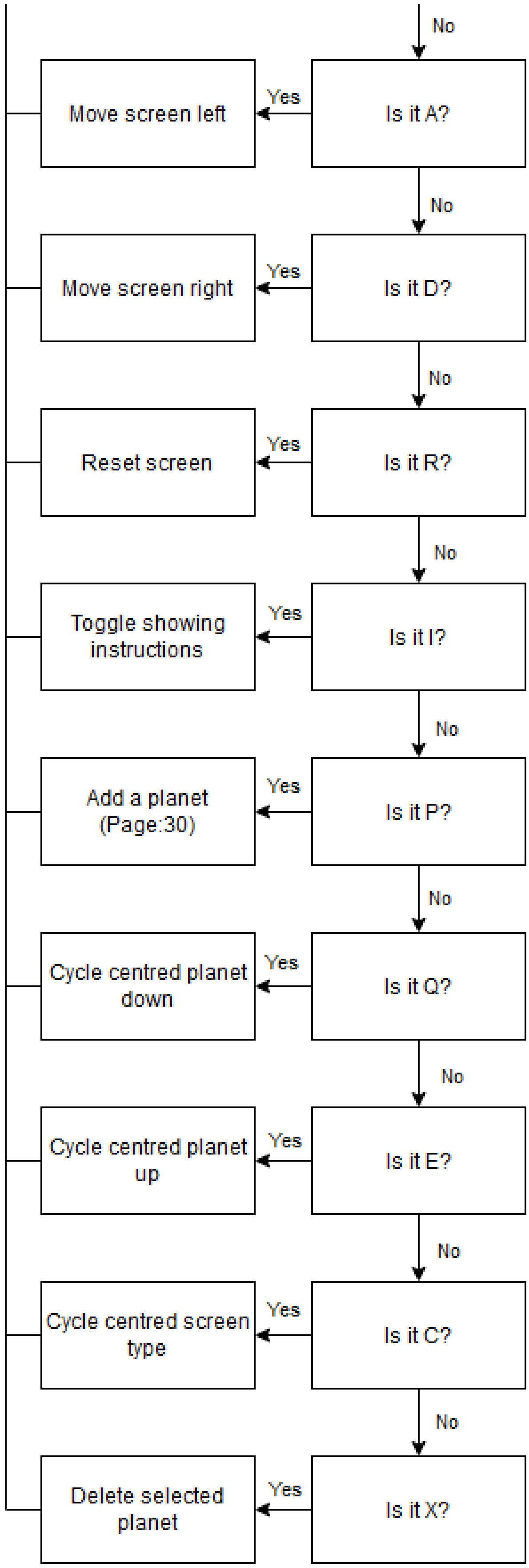


Decreasing/Increasing timeScale works by decreasing/increasing the time between frames.
If timeScale is larger than x300 the timePerTick, timePerTick increases by x5; up to 1 hour per loop. This is to avoid the accuracy of the simulation from decreasing too much.
If timeScale is less than x30 timePerTick, timePerTick decreases by x5.

Zooming In/Out is set to comma and full stop as those keys have the < > arrows. Zooming in and out works by multiplying/dividing respectively the largest distance and the **moveX**, **moveY** variables by 1.1
It also requires the screen to be cleared and trails queue to be cleaned.
TrailsQueue explain on page 24
TrailCleaner explained on 25

Largest distance represents the largest distance from the centre of the screen that the program needs to consider when showing an image. If the planet is farther away from the centre than that, then it does not need to be shown, no more checks needed.
It is by default the distance from the centre of the farthest away object plus 10% to give the planets some wiggle room.

Moving the screen works by increasing or decreasing two variables **moveX**, **moveY** one that stores how much to move the screen x/y
The screen needs to be cleared when the screen is moved



Reset Screen works by resetting the largest distance to the default, moveX and moveY to 0, and clearing trails.

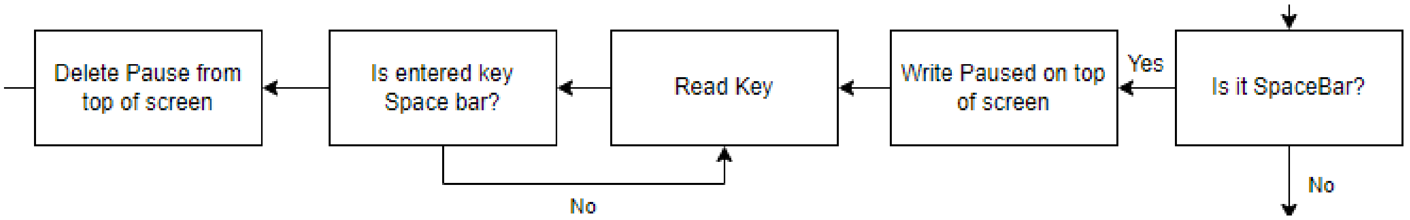
Cycle centred planet works by increasing a variable up or down by one, to select the index of an object in the list of objects. When it reaches the length of the list-1 it resets back to -1. -1 represents no object selected.
Trails need to be cleared

Toggle Showing Instructions would just change a boolean from true to false or vice versa to show or hide what buttons do what E.g. Use WASD to move the view

Cycle centred screen type works by cycling a value between 0 and 2.
0 means not centred,
1 means centred on the selected object,
2 means centrented on the selected object and displaying as if said object was stationary; showing the way other objects move relative to the selected one.
Centred screen explained on page 28

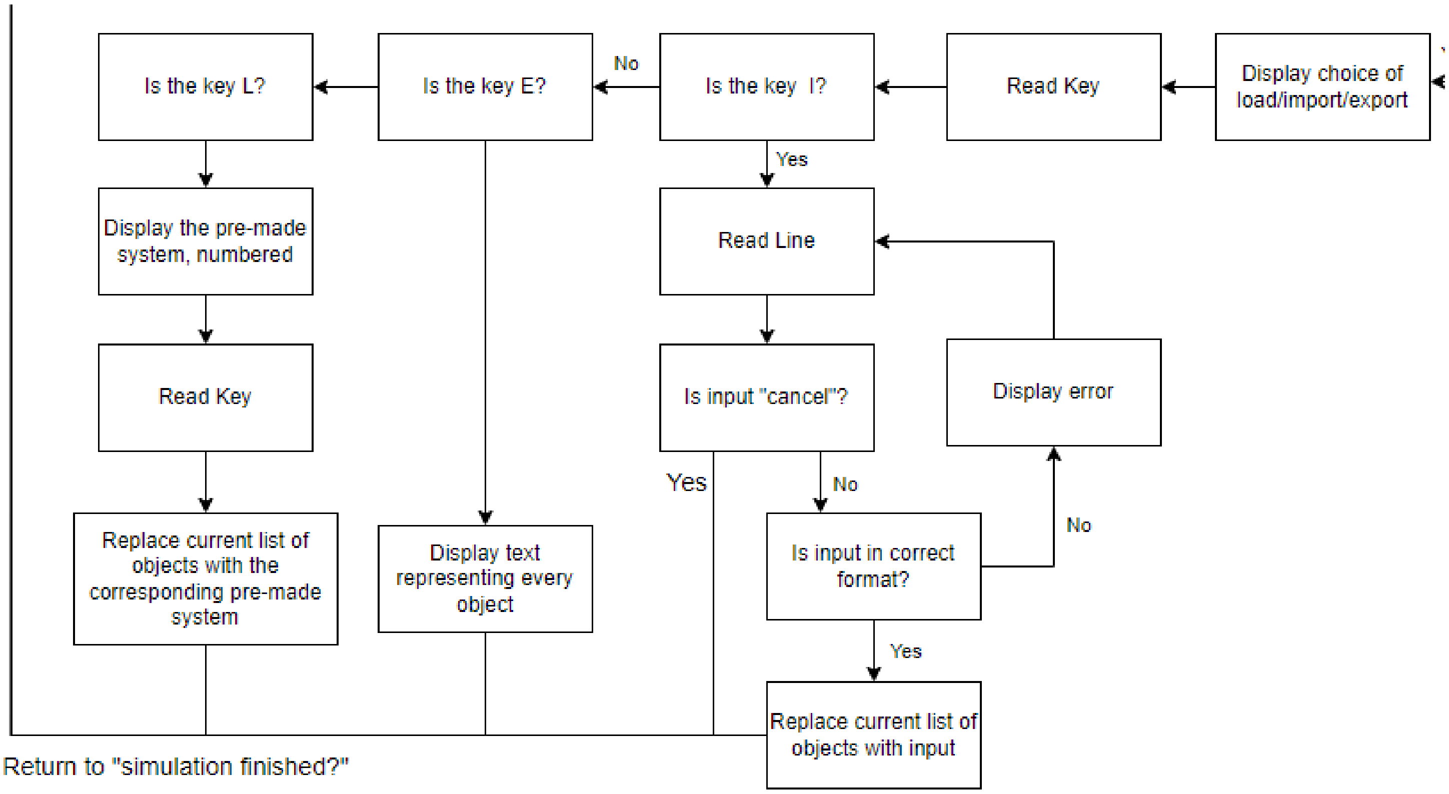
Delete selected planet works by making a copy of the object list, omitting the selected object, and replacing the current object list with this new one.

Pause:


Delete Pause from top of screen
Is entered key Space bar?
Read Key
Write Paused on top of screen
Yes
Is it SpaceBar?
No
No

Pause only stops the user from inputting anything and waits until the user enters space again to go back to the inner loop.

Import/Export/Load:


Is the key L?
Is the key E?
No
Is the key I?
Read Key
Display choice of load/import/export
Yes
Display the pre-made system, numbered
Read Line
Read Key
Is input "cancel"?
Display error
Yes
No
No
Replace current list of objects with the corresponding pre-made system
Display text representing every object
Is input in correct format?
Yes
Replace current list of objects with input
Return to "simulation finished?"

Loading will simply show a selection of systems, numbered and prompting the user to pick one

Exporting will show a large piece of text containing the info of all planets e.g.

PLANET NAME, XY COORDINATES, MASS, DIRECTION VECTOR;
PLANET2 NAME, XY COORDINATES, MASS, DIRECTION VECTOR;
…

It is expected that the user will save this text manually to a file on their computer.

Importing will prompt the user to enter the text from exporting to load.

Trails:

Trails will be handled as a list of 2d arrays.
Every planet will have a 2d array for it, so trails need to be a list as the number of planets can change. In the 2d array there will be the last few positions that the planet was in the last few times it was displayed. The arrays will be handled as a circular queue.

So, let's say in the array there are saved the last 3 positions; **trailLength** 3.
Then there will be 5 2d arrays.
In this there will be the planet name, xy coordinates where the planet was, and how many times that position had been displayed.
When that position has been displayed 3 times, that 2d array will be replaced by the next newest position and have its “displayed” counter set back to 0.

E.g. the 2d array for mars would look like this:
[[Mars ,-2, -4, 4] (position that has been that has already been displayed 3 times)
[Mars ,-1, -2, 3]
[Mars ,0, 0, 2]
[Mars, 1, 2, 1]
[Mars, 2, 4, 0]] (current position, not displayed as would overlap with the actual planet)

Next frame:
[[Mars ,3, 6, 0] (replaced with current position)
[Mars ,-1, -2, 4]
[Mars ,0, 0, 3]
[Mars, 1, 2, 2]
[Mars, 2, 4, 1]]

If a position has its displayed counter set to 4 or 0, then it will not be displayed. In the case of the displayed counter being 4 the array will immediately be replaced with the current position. This is important when first setting up trails.

Setting up trails/Cleaning trails:

The 2d array needs to have trailLength + 2 arrays in it, the one displayed, plus the one that needs to be replaced and the newest one.

As mentioned earlier, trails that have been displayed more than trailLength times will not be displayed and be immediately replaced.

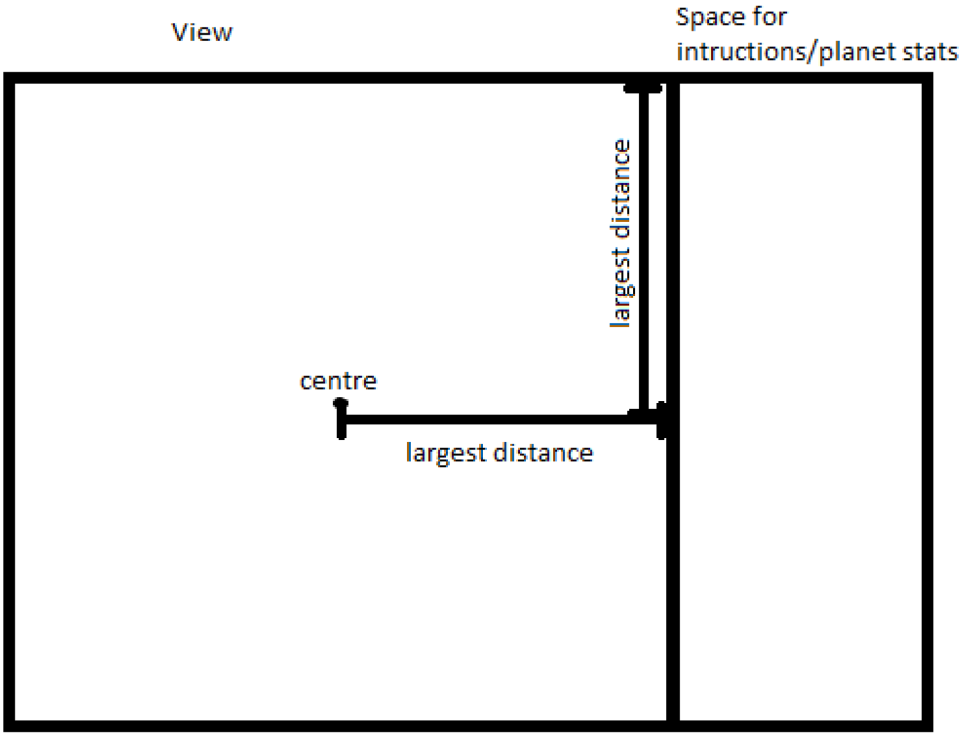
This means that when first setting up the trails 2d array list, every trail needs to have its displayed counter set to trailLength + 1 so that the program does not try to display the initial place-holder trails and immediately replaces them.

This also means that to “delete” the trails queue without making it again (**Cleaning trails**) all we need to do is set the displayed counter to trailLength+1 since the program will stop displaying all those trails and replace them.

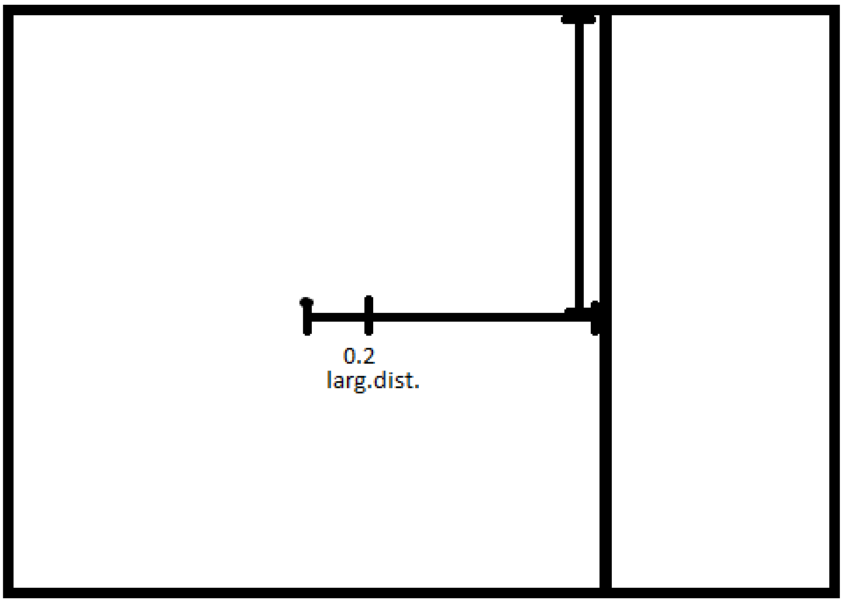
Display Planets:

Real Distance to On-Screen:

To turn the actual distance of the planets into coordinates on the screen I need to find the relative distance between them.
I can do this by dividing the x and the y coordinates of the planet by the **largestDistance** variable.

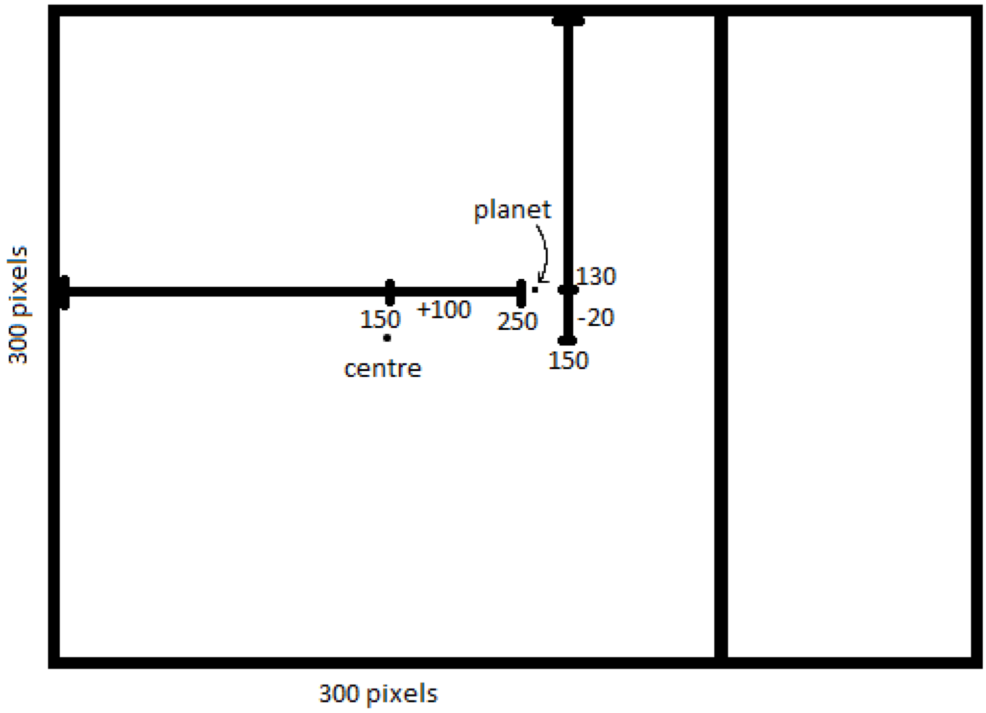


The resulting number can then be used to find the relative xy coordinates, e.g. if the x coordinate divided by the largest distance is 0.2 then its relative x coordinate is 0.2 along from the centre:



We then multiply the relative coordinates by half the view side length to find their actual coordinates on screen. So if the view was 300 pixels wide, the on-screen coordinate of the planet centre for the 0.2 example above would be 0.2*150 = 30

This coordinate system still needs to be processed to the coordinate equivalent of windows forms where things are placed relative to the top left corner of the window. This is done by doing half the screen length plus the x coordinate and minus the y coordinate.
E.g.
(view with side length 300)
a planet with the on-screen coordinates (100, 20)
has coordinates from the top left corner (250, 130)



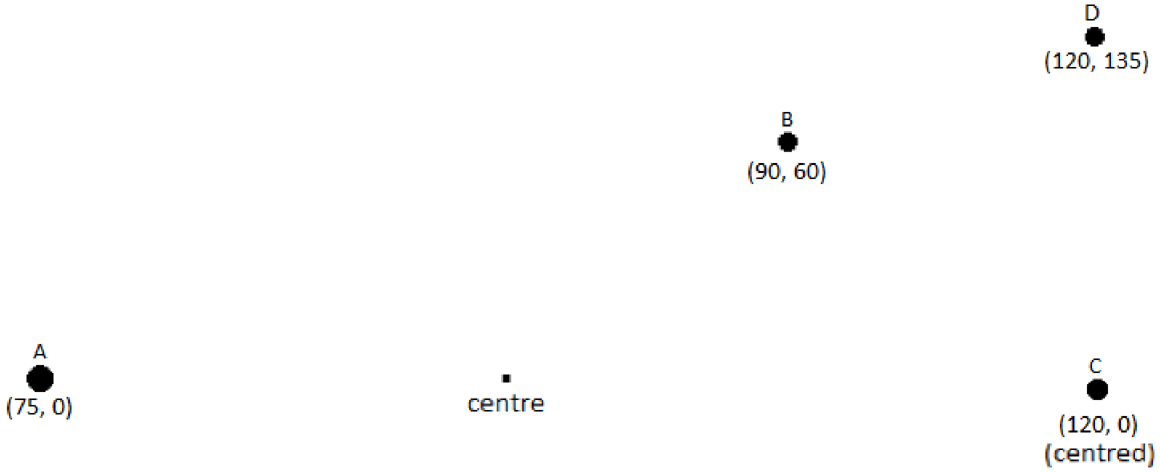
Finally, when actually using these coordinates to display the planets and trails, the program adds the moveX and moveY variables to the respective coordinate. By moving everything, it looks like the view is the thing moving.

Now, we can check if the planet is off-screen by checking if the coordinates are outside ±(half the view length).

Centred Screen:

To centre the screen on a planet we need to place it in the centre of the screen.
To do this we find the difference between the centred planet and the centre, and update the moveX and moveY variables that are used to move the camera based on that.

E.g.
Planets with on-screen xy coordinates (assuming view width is 300)



difference between centre and C:
(150 * -0.8, 150*0) = (-120, 0)
Therefore,
moveX = moveX + -120
moveY = moveY + 0

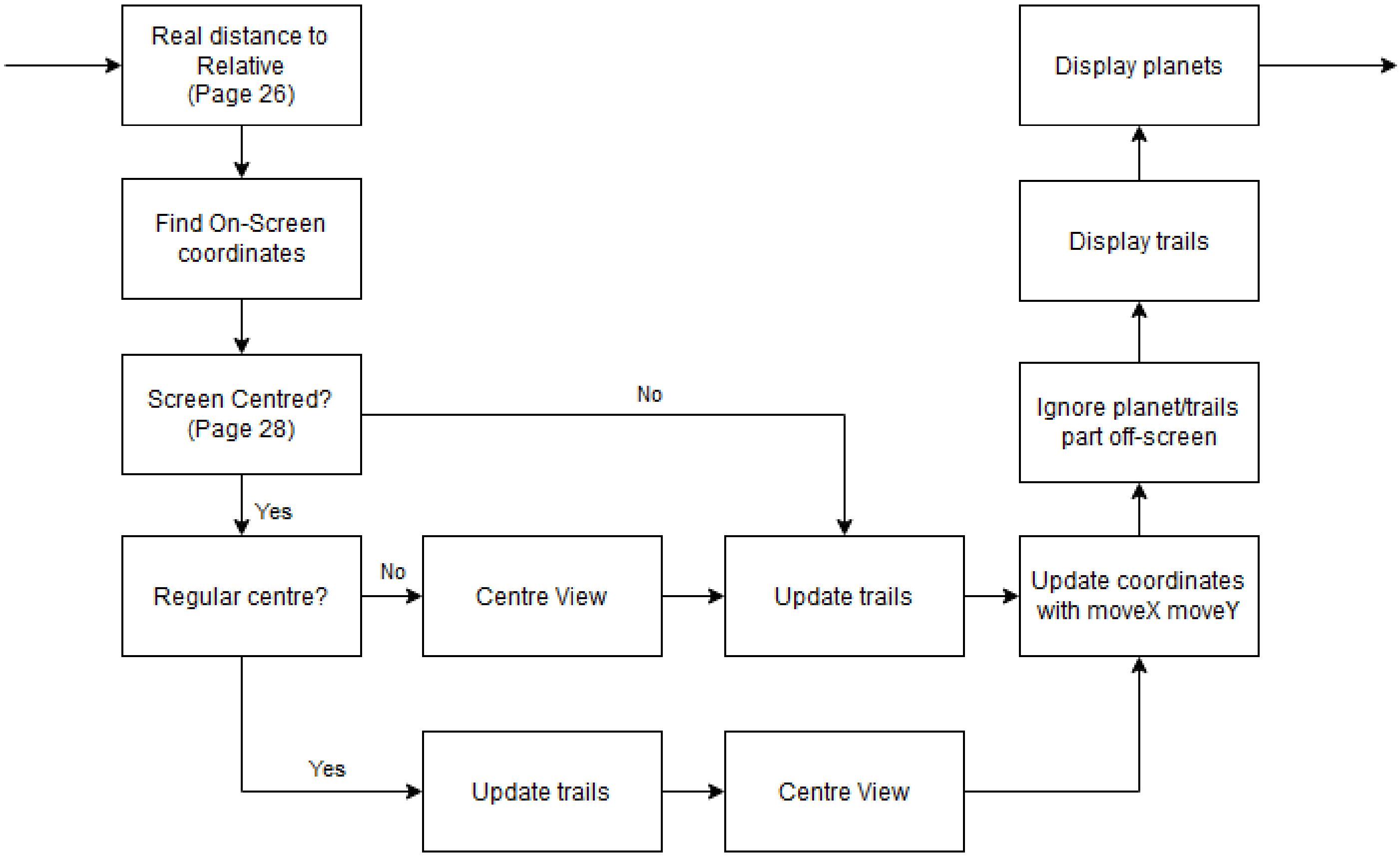
Now, when displaying the planets, everything will be moved such that C is in the centre. Since the movement necessary is added onto moveX and moveY, this does not interfere with the movement of the view.

We can also change whether or not the screen is just centred on a planet or centred on a planet as if it was stationary by changing when the trails are updated.

If we update the trails based on the pre-centred positions of the screen, then they will show the actual movement of the planets.

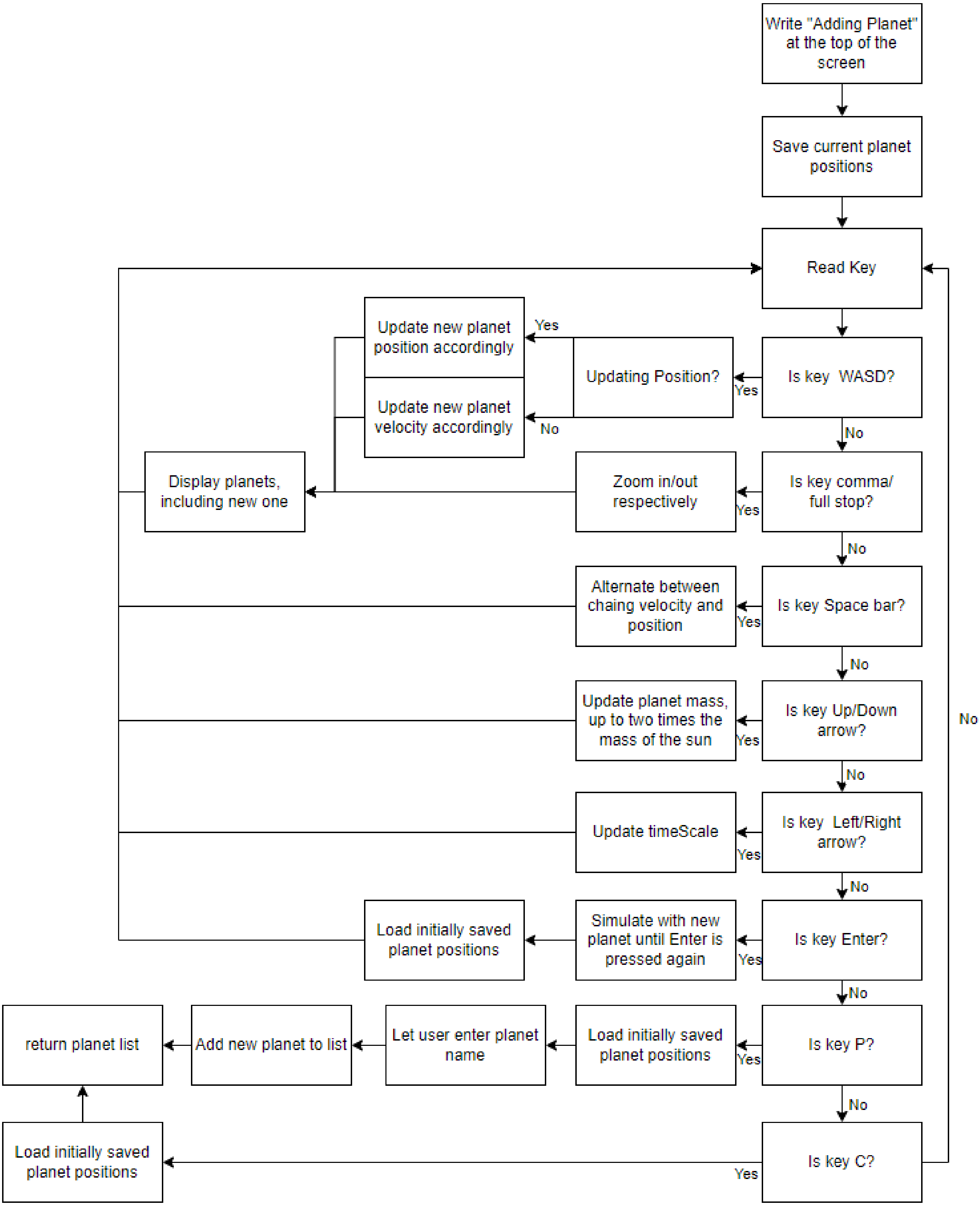
If we update the trails based on the post-centred position of the screen, then they will show the movement of all other planets around the centred planet.

Display Planets Flow-chart


Real distance to Relative (Page 26)
Find On-Screen coordinates
Screen Centred? (Page 28)
No
Yes
Regular centre?
No
Centre View
Update trails
Yes
Update trails
Centre View
Update coordinates with moveX moveY
Ignore planet/trails part off-screen
Display trails
Display planets

Trails are displayed first so that planets are then displayed over in the case that a planet is moving slowly enough that it is overlapping with its trail.
If the trails are updated pre-centred then the trails will represent the movements of the planets.
If the trails are updated post-centred then the trails will represent the movements of the planets relative to the centred planet.

Adding Planet:



Write "Adding Planet" at the top of the screen
Save current planet positions
Read Key
Is key WASD?
Yes
Updating Position?
Yes
Update new planet position accordingly
No
Update new planet velocity accordingly
No
Is key comma/ full stop?
Yes
Zoom in/out respectively
Display planets, including new one
No
Is key Space bar?
Yes
Alternate between chaing velocity and position
No
Is key Up/Down arrow?
Yes
Update planet mass, up to two times the mass of the sun
No
Is key Left/Right arrow?
Yes
Update timeScale
No
Is key Enter?
Yes
Simulate with new planet until Enter is pressed again
Load initially saved planet positions
No
Is key P?
Yes
Load initially saved planet positions
Let user enter planet name
Add new planet to list
return planet list
No
Is key C?
Yes
Load initially saved planet positions
No

Updating position works by moving the planet relative to the screen size, e.g. for screen width 300 pixels moves, when w/a/s/d is pressed, the planet should move one pixel. To find how much the planet needs to be moved by in order to move one pixel, we divide the largest distance by half the screen width. This is then used to add or subtract to the xy coordinates of the planet.

Updating velocity works by multiplying or dividing the xy velocity of the planet by 1.05 (5%) if the entered key is w/a/s/d and by 2 if the entered key is W/A/S/D (capital letter). So if the y velocity is 1 or greater and the entered key is W, the new velocity is 2. If it is -1, and the entered key is s, the new velocity is -1.05. To handle switching form positive to negative or vice versa, when the x/y velocity is between -1 and 1, if the entered key would cause the x/y velocity to further approach zero, it instead switches to 1 or -1.
E.g.
x velocity -1. Entered key is d. New velocity is 1.
y velocity is 0.9. Entered key is S. New velocity is -1.

Updating mass works by multiplying or dividing the mass by 2. The limit of the mass is two times the mass of the sun because I do not know if larger than that would break the simulation. Minimum mass is 1 Kg.

Updating timeScale is something that can be adjusted while not simulating for the case that the view has been zoomed in but the simulation happens too fast. The user can end the simulation, slow down the simulation speed and run again, now being able to see what is happening.

3 Technical Solution

Contents page for important code

Function	Description	Page
Body structure	The Body structure is used to store the name, xy coordinates, xy velocity, mass, name, velocity and colour of a planet.	40
addObject	To make creating a Body easier, addObject creates and returns a Body with the input parameters, named addObject because it is used to add onto Body lists.	40
distanceCalc, vectorCalc, gravPullForceCalc and accelerationsCalc	These implement newtonian physics to calculate the movement of the planets	40-42
objectsImporter	objectsImporter reads an input, checks if it is the correct format and then converts the input into a Body. The Body is then added onto a list which then replaces the objects list.	69
objectsExporter	objectsExporter reads the objects list and outputs on the screen text to represent the planets. Due to the limitations of the console, the text is in the form: XYposition,XYvelocity,mass,radius,name,colour,"newline" This means that when the user enters the text into objectsImporter, ReadLine will automatically stop at "newline" and continue afterwards, making it split the planets.	70
trailQueueAdd	This replaces the current trail queue with a new one that contains the old one's trails, plus space for any new objects in the objects list and removes trails for objects that are no longer in the objects list.	44-45
XYPosToArray	This calculates the onscreen position of the planets. It uses the view height (arrayHeight) as the largest distance equivalent to account for differing aspect ratios, as most aspect ratios have a smaller height than width, meaning that we should be accounting for	87

	height. It also needs to move the planets a bit to the right in the case that the height and width is not the same to keep the view centred. The positions are then returned as an array with two integers as the xy coordinates.	
arrayHeight / arrayWidth	While not functions, it is important to address that these two variables, measuring the height and width of the view are set to the largest console height-2 and width/2-4. The height is set to max-2 to leave space for two lines so that the program can show the time per tick and timeScale. The width is set to (max/2)-4 because the console is divided into 16 pixels high by 8 pixels wide sections for letters. This means that we need sections to be grouped as two width-wise to keep the resolution height and width-wise the same, so the actual width is max/2. The reason why 4 is taken away is to leave some space at the right of the screen, as when trying to full screen the console with the right-most section having a letter assigned to it will sometimes cause the console to crash.	72
trailPrinter	Trail printer goes through the trailQueue and prints the trails, if they are not outside the view, taking into account the movement of the camera and that the console has double the resolution width-wise than height-wise plus 1 to adjust for the outline of the view. Whenever a trail piece is printed it's time to exist decreases by 1. Null is used to represent that that trail piece should not be printed, and will instead have a location be set to it immediately. The program will then go to the next planet’s trails so as not to set up all null trail pieces at once. In the case that the trail has time to exist set to 0 or lower, it is replaced with the current planet’s xy position and the time to exist is set back to however long the trail should be (13).	88-89
centreView	This updates the moveX, moveY variables to move the camera such that the centred planet is in the centre of the view, taking into account that the console has double width resolution compared to height resolution. This does not intervene with the trails as trailPrinter uses xyPos.	89

altcentreView	This updates the on-screen xy coordinates of the planet to move the centred planet into the centre of the screen. This means that when trails are updated they will be based on the post-centred positions as trailPrinter uses xyPos to update trails.	90-91

4 Testing

Test No.	Reason for test	input	Expected result	Pass/ Fail	Test time on video
1	Test accuracy at 60 hours/sec, more importantly 1 minute per tick, while simulating 5 bodies for a prolonged period of time	Load inner solar system, Press: right arrow (x9) Let it run for 30 min	All planets still within circular orbits	Pass	Results at 32:00
2	Test accuracy max speed, 1 hour per tick, while simulating 5 bodies for a prolonged period of time	Load inner solar system, Press: right arrow (x17) Let it run for 30 min	All planets still within circular orbits	Pass	Results at 32:20
3	Test accuracy at 60 hours/sec, 1 minute per tick, while simulating 9 bodies of varying size and large distances for a prolonged period of time	Load whole solar system, Press: right arrow (x9) Let it run for 30 min	All planets still within circular orbits	Pass	Results at 32:40
4	Test max speed, 1 hour per tick, while simulating 9 bodies of varying size and large distances for a prolonged period of time	Load whole solar system, Press: right arrow (x17) Let it run for 30 min	All planets still within circular orbits	Pass	Results at 33:20
5	Test if forces are calculated correctly	Load binary star system, Add planet at (0,0), the barycentre of the binary stars	New planet remains in place	Pass	34:05
6	Test if adding multiple planets causes errors	Load inner solar system, Add 3 planets, With orbits between sun and mercury, Between venus and earth, farther than mars..	Adding another planet does not affect another previously added planet	Pass	35:00
7	Test if every new planet has its name correctly saved	Load inner solar system,	When selected, planets show	Pass	35:00

		Add 3 planets, With orbits between sun and mercury, Between venus and earth, farther than mars. Named , New 1, New 2 and New 3	correct name		
8	Test if new planets have colours saved correctly	Load inner solar system, Add 3 planets, Colour them Blue, Yellow and Red	Planets show correct colours	Pass	35:00
9	Test if collisions occur when planets are close enough	Load inner solar system, Add a planet near venus orbit with no velocity	Planets falls into the sun, collides, and the two objects become one	Pass	39:55
10	Test if collisions result in planets with correct mass	Load inner solar system, Add a planet near venus orbit, at least 1e28 kg mass, with no velocity	Planets falls into the sun, collides, the resulting object has 1.999e30 kg mass (sun has 1.989e30 kg)	Pass	40:40
11	Test if loading into systems multiple times causes errors	Load inner solar system, Load whole solar system, Load binary star system	Inner solar system loads, whole solar system loads, binary star system loads	Pass	44:00
12	Test if loading the same system multiple times causes errors	Load inner solar system, Load inner solar system, Load whole solar system, Load whole solar system, Load binary star system, Load binary star system	Inner solar system loads both times, whole solar system loads both times, binary star system loads both times	Pass	42:45
13	Test if exporting works	Load inner solar system, Add planet orbiting	A large wall of text, that has the planet names	Pass	45:20

		above Mars, named New Mars Export	and colour at the end of each line		
14	Test if importing works	Import data from last test	The inner solar system loads with another planet above mars loads, with all the planets in the same position as before exporting	Pass	46:30
15	Test if loading, exporting and importing within the same “session” doesn’t cause errors	Load whole solar system, Add planet between Mars and Jupiter named New Mars, Export system, Load inner solar system, Import previous system	The whole solar system loads, the planet New Mars is added orbiting between mars and jupiter, The inner solar system loads, The system with New Mars loads	Pass	47:20
16	Test if moving the view works	Load inner solar system, Press: W (x4) A (x4) S (x4) D (x4)	All the planets move along with their trails	Pass	49:20
17	Test if zooming the view in and out works	Load inner solar system Press > (x5) < (x5)	The view zooms in on (0,0) Then it zooms out to what it was before	Pass	49:50
18	Test if zooming in/out after the view has moved works	Load inner solar system, Press: W(x2) D(x6) > (x5) < (x5)	The view moves up, to the right, zooms in and out.	Pass	50:15
19	Test if centering the view on planets works	Load inner solar system, Press: E (x2) C	On the top right of the screen the program shows the Sun and then Mercury being selected.	Pass	50:40

			The view then centres on Mercury.		
20	Test if the alternative centering (showing the system as if everything was moving around the centred object) works	Load inner solar system, Press: E (x4) C (x2)	The view centres on the Earth. The view switches to “alternative centering”. Mars should be seen switching directions occasionally if the function works as it should.	Pass	51:00
21	Test if moving and zooming after the view is centred works	Load inner solar system, Press: E (x2) C W (x2) S (x4) D (x3) > (x4) < (x4)	The view centres on Mercury. It then moves, zooms in and back out to the initial zoom while still moving with Mercury	Pass	51:40
22	Test if moving and zooming after the view is centred with alternative centring works	Load inner solar system, Press: E (x2) C (x2) W (x2) S (x4) D (x3) > (x4) < (x4)	The view centres on Mercury with alternative centring. It then moves, zooms in and back out to the initial zoom while keeping Mercury appearing stationary	Pass	52:10
23	Test if removing planets works	Load inner solar system Press: E X Y	When X is pressed a box appears to confirm the deletion. After confirming the planet is deleted	Pass	52:50
24	Test if opening/closing the instructions menu causes	Load inner solar system	The instructions menu should	Pass	53:20

	errors in the way the planets are displayed	Press: I (x2)	close and then open		
25	Test if opening/closing the instructions menu causes errors while adding a planet	Load inner solar system, Press: P I (x2)	The program enters planet adder mode. The instruction menu closes and then opens.	Fail	54:05
26	Test if the cancel function when adding a planet works	Load inner solar system Press: P C	The program goes to add a planet and then returns to simulating the inner solar system	Pass	55:25

Notes:

Test 4:
The entire solar system moved up while remaining in their circular orbits. The inner solar system in the same time span of 30*60*2 = 3600 years did not. This is probably because for the inner solar system I made the Sun orbit around the solar system's barycentre by making it move downwards, but forgot to make it move downwards faster for the whole system to account for Jupiter, Saturn, Uranus and Neptune moving up which ended up slowly dragging the whole solar system up.

Test 25 actual result:
The instructions menu did not close. No other errors occurred during test 25.

5 Evaluation

Overall

The program I've made creates a simple to understand play-ground to see how the world works on a solar level, as intended. The trails are a very useful visual queue for the relative planet's velocities and are necessary for when the screen is centred on a planet to show . The way that planets are added allows the user to try to place them from multiple locations, velocities and how they would react with differing mass. To use it however, it does take some time to read the instructions, and to memorise the controls of the planet adding part of it. Moreso, the program using the console limits its visual clarity, as volumetric rendering is not an option which could have been nice to show just how large the distance between planets are, even if impractical (on a 1920x1080 screen, if the sun was from top to bottom of the screen, 1080 pixels in diameter, the Earth would be 10 pixels in diameter, and 0.12 pixels if the screen had the sun's centre on the left and the earth on the right, along the 1920 pixels).

User Feedback

Q1) What do you think of how the program runs?
A1) The program runs very efficiently, it displays the solar system clearly and with no problems, there is no stuttering in any of the visual portions displaying the planets or statistics. Everything moves smoothly in my opinion.

Q2) What do you think of how the program looks?
A2) I find the interface style very minimalistic, it's not overloaded with information or menus, it is straight to the point with all the information it has displayed and makes it all easy to understand.

Q3) Do you think it has helped you understand how objects move in space?
A3) Yes, the program mostly helped me understand how objects move in space a lot better, however there are some parts of the program like the trails having gaps due to the high velocity of the objects making it a bit confusing and the fact that the console limits the representation of the objects collisions if they are in the same pixel range and making tough to understand if a collision has occurred or not

Q4) What could be improved/changed?
A4) From what I can see now, there are not many things that would require changes as the program is already very good at doing what it was meant to do. Maybe just the gap error left by the trail is one thing that could be looked into, but I understand that for the object collision problem it is just the console's limitations and therefore see no need for any major improvements.

Objectives Review

1.1 The program can simulate the inner solar system such that at its fastest and least accurate it goes 6*(10^-5) % out of sync for every year simulated, so this objective has been achieved.
1.2 The program can simulate between 30 seconds per second up to 2 years per second, so this objective has been achieved
1.3 The program can switch between simulating 1 second per second at a time up to 1 hour at a time to lower the time spent calculating, so this objective has been achieved
1.4 This objective has been achieved

2.1 The add planet function allows the user to trial and error their way into adding a new planet into their desired orbit, so this objective has been achieved.
2.2 This objective has been achieved, however while the program can zoom in and out, trails are reset. The program could be improved by having the trails' position updated when zooming in/out as opposed to having them reset.
2.3 The program has a few pre-sets that can be loaded, so this objective has been achieved.
2.4 The program allows the user to save and load the system that they made as a long piece of text, making these saves easier to share. This objective has been achieved, but it would be nice to have the program be able to save the systems locally as well as in text form.

3.1 This objective has not been achieved. It could be achieved by parsing the numbers for length/mass through a function to solar masses or earth masses before displaying them.
3.2 This objective has not been achieved. It could be achieved by adding the two planets that exert the most force on the selected planet as one of the planet’s stats. This would also require the program to save the two planets with the highest force on each planet while calculating forces in the accelerationsCalc function.
3.3 This objective has been achieved in the form of the instructions tab which can show what each button does.
3.4 This objective has been achieved, however since the trails made are based on the visual position of the planets, if a planet moves very fast, the trail will have a gap in it which is a problem. To be clear, it is not a problem for the planet as jumping multiple spaces implies high speed, but it is a problem for the trails because they are meant to show the trajectory the planet took.
To fix this, the trails queue would have to be changed significantly, from a list of a 2d array, to a list of a list of an array.
The actual length of the trails would not be determined by the number of trail pieces, and instead be adaptive in length, hence why the 2d array is changed to a list of an array.
Whether or not a trail piece should be deleted would then be determined by the amount of simulated time for which that trail piece has existed (e.g. half a year trail length would mean that the earth’s trail would be halfway around the sun).
Whether or not a new trail piece should be created would then be determined by if the planet has moved a space, leaving a trail piece at its previous location.

6 Code

```
class Program
{
    struct Body
    {
        public double posX; //m
        public double posY;
        public double xVelocity; // m/s
        public double yVelocity;
        public double mass; //kg
        public string name;
        public double radius; //m
        public string colour;
    }

    static Body addObject(double X, double Y, double xVel, double yVel, double mas,
double rad, string na, string color)
    {
        Body temp;
        temp.posX = X;
        temp.posY = Y;
        temp.xVelocity = xVel;
        temp.yVelocity = yVel;
        temp.mass = mas;
        temp.name = na;
        temp.radius = rad;
        temp.colour = color;
        return temp;
    }

    static double distanceCalc(double x1, double y1, double x2, double y2)
    {
        double distance = Math.Sqrt(Math.Pow((x1 - x2), 2) + Math.Pow((y1 - y2), 2));
        //distance = square root of ((difference in x axis) ^ 2 + (difference in y axis ^ 2))
        return distance;
    }

    static double[] vectorCalc(double x1, double y1, double x2, double y2)
    {
        //function made just in case it is needed.
        double[] vector = new double[2];
        vector[0] = x2 - x1;
        vector[1] = y2 - y1;
        return vector;
    }
    static double gravPullForceCalc(double mass1, double mass2, double distance)
    {
        double gravConstant = 6.6743e-11;

        double gravitationalPull = (mass1 * mass2) / Math.Pow(distance, 2);
        //gravitational pull = gravitational constant * (mass1*mass2) / distance^2

        gravitationalPull = gravConstant * gravitationalPull;
        //adding the gravitational constant

        return gravitationalPull;
    }

    static double[,] accelerationCalc(List<Body> Objects)
    {
        double[,] accelerations = new double[Objects.Count, 2];
        double distance;
        double gravPullForce;
        double[] unitVector = new double[2];

        for (int body1 = 0; body1 < Objects.Count; body1++)
        {
          for (int body2 = 0; body2 < Objects.Count; body2++)
          {
            if (body1 != body2)
            {

              distance = distanceCalc(Objects[body1].posX, Objects[body1].posY,
Objects[body2].posX, Objects[body2].posY);
              unitVector = vectorCalc(Objects[body1].posX, Objects[body1].posY,
Objects[body2].posX, Objects[body2].posY);

              unitVector[0] = unitVector[0] / distance;
              unitVector[1] = unitVector[1] / distance;

              gravPullForce = gravPullForceCalc(Objects[body1].mass,
Objects[body2].mass, distance);

              for (int axis = 0; axis < unitVector.Length; axis++)
              {
                accelerations[body1, axis] += (unitVector[axis] * gravPullForce) /
Objects[body1].mass;
              }

            }
          }
```

```
            }
            return accelerations;
        }

        static double largestDistanceFromCenter(List<Body> Objects)
        {
            double largestDistance = 0;

            for (int i = 0; i < Objects.Count; i++)
            {
                if (Math.Sqrt(Objects[i].posX * Objects[i].posX + Objects[i].posY * Objects[i].posY)
+ Objects[i].radius > largestDistance)
                {
                    largestDistance = Math.Sqrt(Objects[i].posX * Objects[i].posX + Objects[i].posY
* Objects[i].posY);
                    largestDistance += Objects[i].radius;
                }
            }
            for (int i = 0; i < Objects.Count; i++)
            {
                if (largestDistance < Objects[i].radius * 2)
                {
                    largestDistance = Objects[i].radius * 2;
                }
            }
            return largestDistance;
        }

        static List<Body> velocityUpdate(List<Body> Objects, double[,] accelerations, double
timePerTick)
        {
            Body holder;
            for (int i = 0; i < Objects.Count; i++)
            {
                holder = Objects[i];

                holder.xVelocity = holder.xVelocity + accelerations[i, 0] * timePerTick;
                //x axis velocity += x axis acceleration
                holder.yVelocity = holder.yVelocity + accelerations[i, 1] * timePerTick;
                //y axis velocity += y axis acceleration

                Objects[i] = holder;
            }
            return Objects;
        }

        static List<Body> positionUpdate(List<Body> Objects, double timePerTick)
```

```
{
   Body holder;
   for (int i = 0; i < Objects.Count; i++)
   {
      holder = Objects[i];

      holder.posX = holder.posX + Objects[i].xVelocity * timePerTick;
      //x axis position += x axis velocity
      holder.posY = holder.posY + Objects[i].yVelocity * timePerTick;
      //y axis position += y axis velocity

      Objects[i] = holder;
   }
   return Objects;
}

static void colourChanger(string colour)
{
   if (colour == "yellow")
   {
      Console.ForegroundColor = ConsoleColor.Yellow;
   }
   else if (colour == "dark yellow")
   {
      Console.ForegroundColor = ConsoleColor.DarkYellow;
   }
   else if (colour == "green")
   {
      Console.ForegroundColor = ConsoleColor.Green;
   }
   else if (colour == "dark green")
   {
      Console.ForegroundColor = ConsoleColor.DarkGreen;
   }
   else if (colour == "red")
   {
      Console.ForegroundColor = ConsoleColor.Red;
   }
   else if (colour == "dark red")
   {
      Console.ForegroundColor = ConsoleColor.DarkRed;
   }
   else if (colour == "magenta")
   {
      Console.ForegroundColor = ConsoleColor.Magenta;
   }
   else if (colour == "dark magenta")
```

```
        {
            Console.ForegroundColor = ConsoleColor.DarkMagenta;
        }
        else if (colour == "blue")
        {
            Console.ForegroundColor = ConsoleColor.Blue;
        }
        else if (colour == "dark blue")
        {
            Console.ForegroundColor = ConsoleColor.DarkBlue;
        }
        else if (colour == "cyan")
        {
            Console.ForegroundColor = ConsoleColor.Cyan;
        }
        else if (colour == "dark cyan")
        {
            Console.ForegroundColor = ConsoleColor.DarkCyan;
        }
        else if (colour == "white")
        {
            Console.ForegroundColor = ConsoleColor.White;
        }
    }

    static void scroll(int arrayWidth, int row)
    {
        Console.CursorLeft = arrayWidth * 2 + 2;
        Console.CursorTop = row;
    }

    static List<string[,]> trailQueueAdd(List<string[,]> trailsQueue, int trailLenght,
List<Body> Objects)
    {
        List<string[,]> trailsQueueReplacement = new List<string[,]> { };

        for (int i = 0; i < Objects.Count; i++)
        {
            trailsQueueReplacement.Add(new string[trailLenght, 4]);
            for (int j = 0; j < trailLenght; j++)
            {
                trailsQueueReplacement[i][j, 0] = Objects[i].name;
            }

            //initialising the list to be full of empty 2d arrays that will be recycled as the trail
fades
            //instead of just keep adding more to the list using .removeat() and .add()
            //the way the list will be used is:
            //the list contains one arary for each planet
            //each 2d aryay contains five arrays with
            //the name of the body that will be unchanging,
            //the x and y position where that piece of the trail needs to be placed, and
            //and how long ago the that piece of the trail was first placed (based on how many
times it has been drawn)
            //when the trail has been drawn a certain number of times it gets removed and in
the same 2d array the location is changed and timer is reset
        }

        if (trailsQueue.Count <= trailsQueueReplacement.Count)
        {
            for (int i = 0; i < trailsQueue.Count; i++)
            {
                for (int j = 0; j < trailLenght; j++)
                {
                    trailsQueueReplacement[i][j, 1] = trailsQueue[i][j, 1];
                    trailsQueueReplacement[i][j, 2] = trailsQueue[i][j, 2];
                }
            }
        }

        return trailsQueueReplacement;
    }

    static void cleanTrails(ref List<string[,]> trails, int trailLenght, int planetCount)
    {
        for (int i = 0; i < planetCount; i++)
        {
            for (int j = 0; j < trailLenght; j++)
            {
                trails[i][j, 3] = null;
            }
        }
    }

    static List<Body> planetRemover(List<Body> Objects, int id)
    {
        List<Body> newObjects = new List<Body>();
        for (int i = 0; i < Objects.Count; i++)
        {
            if (i != id)
            {
                newObjects.Add(Objects[i]);
            }
        }
        return newObjects;
    }

    static List<Body> bodyCopy(List<Body> original)
    {
        List<Body> Copy = new List<Body>();
        for (int i = 0; i < original.Count; i++)
        {
            Copy.Add(original[i]);
        }
        return Copy;
    }

    static List<Body> addPlanet(List<Body> Objects, int arrayHeight, int arrayWidth, double
largestDistance, List<string[,]> trailsQueue, int trailLenght, double moveX, double moveY)
    {
        Body newPlanet = new Body();
        List<Body> newObjects = bodyCopy(Objects);
        List<string[,]> newtrailsQueue = trailQueueAdd(trailsQueue, trailLenght,
newObjects);
        //List<Body> backUpObjects = Objects;
        ConsoleKeyInfo enteredKey;

        Console.CursorVisible = true;
        arrayWidth = (Console.WindowWidth - 50) / 2 - 2;

        double cursorX = arrayWidth;
        double cursorY = arrayHeight / 2 + 1;
        double jumpOne = 0;

        double posX = 0;
        double posY = 0;
        double Xvelocity = 0;
        double Yvelocity = 0;
        double mass = 1000000;
        double radius = 200;
        string name = "????";
        string colour = "red";

        int Adjust = (arrayWidth - 1) / 2 - (arrayHeight - 1) / 2;

        bool accept = false;
        bool adjustingVelocity = false;
        bool simulating = false;
        bool doneSimulating = false;
        bool showingInstructions = true;
        bool cancel = false;

        double solarMass = 1.989e30;
        double time;
        double timeLastImage;
        int timePerTickInSec;
        int timeScale = 60 * 60;
        timePerTickInSec = 60;
        int planetCount;
        double largestDistanceBackup;
        bool collision;

        double posXScroll = largestDistance / (arrayWidth / 2) * 1.6;
        double posYScroll = largestDistance / (arrayHeight / 2);

        int actualTimeMilliseconds = int.Parse(DateTime.Now.Millisecond.ToString());

        double[,] acceleration;

        Console.CursorTop = (int)Math.Round(cursorY);
        Console.CursorLeft = (int)Math.Round(cursorX);

        Console.Clear();

        boxPrinter(arrayHeight, arrayWidth);

        printInstructions(showingInstructions, arrayWidth, true, adjustingVelocity);

        displayPlanetStats(newObjects, newObjects.Count - 1, arrayWidth);

        visualPosition(newObjects, ref largestDistance, ref trailsQueue, trailLenght, 0, 0,
(int)moveX, (int)moveY, false, arrayHeight, arrayWidth);

        newObjects = bodyCopy(Objects);

        newObjects.Add(addObject(posX, posY, Xvelocity, Yvelocity, mass, radius, name,
colour));

        while (!accept)
        {
          Console.CursorLeft = arrayWidth - 8;
          Console.CursorTop = 1;
          Console.Write(">ADDING  PLANET<");

          planetCount = newObjects.Count;

          time = 0;
```

```
            timeLastImage = 1;

            actualTimeMilliseconds = int.Parse(DateTime.Now.Millisecond.ToString());

            visualPosition(Objects, ref largestDistance, ref trailsQueue, trailLenght, -1, 0,
(int)moveX, (int)moveY, false, arrayHeight, arrayWidth);

            Console.Write("Current Time Scale: " + secondsSimplifier(timeScale * 30) + " per
second    ");
            Console.Write("\nTime per tick: " + secondsSimplifier(timePerTickInSec) + "     ");

            Console.CursorTop = (int)Math.Round(cursorY);
            Console.CursorLeft = (int)Math.Round(cursorX);
            Console.Write(name.Substring(2));
            Console.CursorTop = (int)Math.Round(cursorY);
            Console.CursorLeft = (int)Math.Round(cursorX);

            enteredKey = Console.ReadKey(true);


            if (enteredKey.Key == ConsoleKey.OemComma)
            {
               largestDistance = largestDistance * 1.1;
               moveX = moveX / 1.1;
               posX = posX * 1.1;
               moveY = moveY / 1.1;
               posY = posY * 1.1;
               //zoom out
               posXScroll = largestDistance / (arrayWidth / 2) * 1.85;
               posYScroll = largestDistance / (arrayHeight / 2);

               boxCleaner(arrayHeight, arrayWidth);
               visualPosition(Objects, ref largestDistance, ref trailsQueue, trailLenght, -1, 0,
(int)moveX, (int)moveY, false, arrayHeight, arrayWidth);

               Console.CursorLeft = arrayWidth - 8;
               Console.CursorTop = 1;
               Console.Write(">ADDING  PLANET<");
            }
            else if (enteredKey.Key == ConsoleKey.OemPeriod && largestDistance > 10)
            {
               largestDistance = largestDistance / 1.1;
               moveX = moveX * 1.1;
               posX = posX / 1.1;
               moveY = moveY * 1.1;
               posY = posY / 1.1;
               //zoom in
```

```
                posXScroll = largestDistance / (arrayWidth / 2) * 1.85;
                posYScroll = largestDistance / (arrayHeight / 2);

                boxCleaner(arrayHeight, arrayWidth);
                visualPosition(Objects, ref largestDistance, ref trailsQueue, trailLenght, -1, 0,
(int)moveX, (int)moveY, false, arrayHeight, arrayWidth);

                Console.CursorLeft = arrayWidth - 8;
                Console.CursorTop = 1;
                Console.Write(">ADDING  PLANET<");
            }
            else if (enteredKey.Key == ConsoleKey.Enter)
            {
                simulating = true;

                boxCleaner(arrayHeight, arrayWidth);

                newObjects = bodyCopy(Objects);

                newObjects.Add(addObject(posX, posY, Xvelocity, Yvelocity, mass, radius,
name, colour));

                newtrailsQueue = trailQueueAdd(trailsQueue, trailLenght, newObjects);
            }
            else if (enteredKey.Key == ConsoleKey.C)
            {
                cancel = true;
                accept = true;
            }
            else if (enteredKey.Key == ConsoleKey.Spacebar)
            {
                adjustingVelocity = !adjustingVelocity;
                printInstructions(showingInstructions, arrayWidth, true, adjustingVelocity);
            }
            else if (enteredKey.Key == ConsoleKey.UpArrow)
            {
                if (mass < solarMass * 100)
                {
                    mass *= 2;
                    radius = Math.Round(radius * Math.Pow(2, 1.0 / 3.0));
                    //need to raise 2 to the power of 1.0/3.0 to cube root it and not 1/3 because
1/3 return the closest integer; 0
                    //1.0/3.0 correctly returns a float, although 1/3 cannot be represent as a
floating point number, it gives a floating point error
                    //so the answer needs then to be rounded
                }
                newObjects = bodyCopy(Objects);
                newObjects.Add(addObject(posX, posY, Xvelocity, Yvelocity, mass, radius,
name, colour));

                clearCentredStats(arrayWidth);

                displayPlanetStats(newObjects, newObjects.Count - 1, arrayWidth);

                newObjects = bodyCopy(Objects);
            }
            else if (enteredKey.Key == ConsoleKey.DownArrow)
            {
                if (mass > 1)
                {
                    mass /= 2;
                    radius = Math.Round(radius / Math.Pow(2, 1.0 / 3.0));
                    //need to raise 2 to the power of 1.0/3.0 to cube root it and not 1/3 because
1/3 return the closest integer; 0
                    //1.0/3.0 correctly returns a float, although 1/3 cannot be represent as a
floating point number, it gives a floating point error
                    //so the answer needs then to be rounded
                }
                newObjects = bodyCopy(Objects);

                newObjects.Add(addObject(posX, posY, Xvelocity, Yvelocity, mass, radius,
name, colour));

                clearCentredStats(arrayWidth);

                displayPlanetStats(newObjects, newObjects.Count - 1, arrayWidth);

                newObjects = bodyCopy(Objects);
            }
            else if (enteredKey.Key == ConsoleKey.RightArrow)
            {
                timeChanger(ref timeScale, ref timePerTickInSec, true);
                //speed up
            }
            else if (enteredKey.Key == ConsoleKey.LeftArrow)
            {
                timeChanger(ref timeScale, ref timePerTickInSec, false);
                //slow down
            }
            else if (enteredKey.Key == ConsoleKey.P)
            {
                accept = true;
            }
```

```
else
{
    if (!adjustingVelocity)
    {
        if (enteredKey.KeyChar == 'A')
        {
            Console.Write("  ");
            if (cursorX - 1 > 1)
            {
                posX -= posXScroll;
                cursorX -= 2;
            }
            newObjects = bodyCopy(Objects);

            newObjects.Add(addObject(posX, posY, Xvelocity, Yvelocity, mass, radius,
name, colour));

            clearCentredStats(arrayWidth);

            displayPlanetStats(newObjects, newObjects.Count - 1, arrayWidth);

            newObjects = bodyCopy(Objects);
        }
        else if (enteredKey.KeyChar == 'a')
        {
            Console.Write("  ");
            if (cursorX - 0.2 > 1)
            {
                posX -= posXScroll / 10;
                jumpOne -= 0.2;
            }
            if (jumpOne <= -1)
            {
                cursorX -= 2;
                jumpOne = 1;
            }
            newObjects = bodyCopy(Objects);

            newObjects.Add(addObject(posX, posY, Xvelocity, Yvelocity, mass, radius,
name, colour));

            clearCentredStats(arrayWidth);

            displayPlanetStats(newObjects, newObjects.Count - 1, arrayWidth);

            newObjects = bodyCopy(Objects);
        }
```

```
            else if (enteredKey.KeyChar == 'D')
            {
                Console.Write("  ");
                if (cursorX + 2 < arrayWidth * 2 + 1)
                {
                    posX += posXScroll;
                    cursorX += 2;
                }
                newObjects = bodyCopy(Objects);

                newObjects.Add(addObject(posX, posY, Xvelocity, Yvelocity, mass, radius,
name, colour));

                clearCentredStats(arrayWidth);

                displayPlanetStats(newObjects, newObjects.Count - 1, arrayWidth);

                newObjects = bodyCopy(Objects);
            }
            else if (enteredKey.KeyChar == 'd')
            {
                Console.Write("  ");
                if (cursorX + 0.2 < arrayWidth * 2 + 1)
                {
                    posX += posXScroll / 10;
                    jumpOne += 0.2;
                }
                if (jumpOne >= 1)
                {
                    cursorX += 2;
                    jumpOne = -1;
                }
                newObjects = bodyCopy(Objects);

                newObjects.Add(addObject(posX, posY, Xvelocity, Yvelocity, mass, radius,
name, colour));

                clearCentredStats(arrayWidth);

                displayPlanetStats(newObjects, newObjects.Count - 1, arrayWidth);

                newObjects = bodyCopy(Objects);
            }
            else if (enteredKey.KeyChar == 'W')
            {
                Console.Write("  ");
                if (cursorY > 1)
```

```
                            {
                                posY += posYScroll;
                                cursorY--;
                            }
                            newObjects = bodyCopy(Objects);

                            newObjects.Add(addObject(posX, posY, Xvelocity, Yvelocity, mass, radius,
name, colour));

                            clearCentredStats(arrayWidth);

                            displayPlanetStats(newObjects, newObjects.Count - 1, arrayWidth);

                            newObjects = bodyCopy(Objects);
                        }
                        else if (enteredKey.KeyChar == 'w')
                        {
                            Console.Write("  ");
                            if (cursorY - 0.1 > 1)
                            {
                                posY += posYScroll / 10;
                                cursorY -= 0.1;
                            }
                            newObjects = bodyCopy(Objects);

                            newObjects.Add(addObject(posX, posY, Xvelocity, Yvelocity, mass, radius,
name, colour));

                            clearCentredStats(arrayWidth);

                            displayPlanetStats(newObjects, newObjects.Count - 1, arrayWidth);

                            newObjects = bodyCopy(Objects);
                        }
                        else if (enteredKey.KeyChar == 'S')
                        {
                            Console.Write("  ");
                            if (cursorY < arrayHeight)
                            {
                                posY -= posYScroll;
                                cursorY++;
                            }
                            newObjects = bodyCopy(Objects);

                            newObjects.Add(addObject(posX, posY, Xvelocity, Yvelocity, mass, radius,
name, colour));
```

```
                    clearCentredStats(arrayWidth);

                    displayPlanetStats(newObjects, newObjects.Count - 1, arrayWidth);

                    newObjects = bodyCopy(Objects);
                }
                else if (enteredKey.KeyChar == 's')
                {
                    Console.Write("  ");
                    if (cursorY < arrayHeight)
                    {
                        posY -= posYScroll / 10;
                        cursorY += 0.1;
                    }
                    newObjects = bodyCopy(Objects);

                    newObjects.Add(addObject(posX, posY, Xvelocity, Yvelocity, mass, radius,
name, colour));

                    clearCentredStats(arrayWidth);

                    displayPlanetStats(newObjects, newObjects.Count - 1, arrayWidth);

                    newObjects = bodyCopy(Objects);
                }

                newObjects = bodyCopy(Objects);

                clearCentredStats(arrayWidth);

                newObjects.Add(addObject(posX, posY, Xvelocity, Yvelocity, mass, radius,
name, colour));

                displayPlanetStats(newObjects, newObjects.Count - 1, arrayWidth);

                newObjects = bodyCopy(Objects);

                Console.CursorTop = (int)Math.Round(cursorY);
                Console.CursorLeft = (int)Math.Round(cursorX);
            }
            else if (adjustingVelocity)
            {
                if (enteredKey.KeyChar == 'w')
                {
                    if (Yvelocity == 0)
                    {
                        Yvelocity = 1;
```

```
    }
    else if (Yvelocity > -2 && Yvelocity < 0)
    {
        Yvelocity = 0;
    }
    else if (Yvelocity < 0)
    {
        Yvelocity = Yvelocity / 1.05;
    }
    else if (Yvelocity > 0)
    {
        Yvelocity = Yvelocity * 1.05;
    }
}
else if (enteredKey.KeyChar == 'W')
{
    if (Yvelocity == 0)
    {
        Yvelocity = 1;
    }
    else if (Yvelocity > -2 && Yvelocity < 0)
    {
        Yvelocity = 0;
    }
    else if (Yvelocity < 0)
    {
        Yvelocity = Yvelocity / 2;
    }
    else if (Yvelocity > 0)
    {
        Yvelocity = Yvelocity * 2;
    }
}
else if (enteredKey.KeyChar == 's')
{
    if (Yvelocity == 0)
    {
        Yvelocity = -1;
    }
    else if (Yvelocity < 2 && Yvelocity > 0)
    {
        Yvelocity = 0;
    }
    else if (Yvelocity < 0)
    {
        Yvelocity = Yvelocity * 1.05;
    }
    else if (Yvelocity > 0)
    {
        Yvelocity = Yvelocity / 1.05;
    }
}
else if (enteredKey.KeyChar == 'S')
{
    if (Yvelocity == 0)
    {
        Yvelocity = -1;
    }
    else if (Yvelocity < 2 && Yvelocity > 0)
    {
        Yvelocity = 0;
    }
    else if (Yvelocity < 0)
    {
        Yvelocity = Yvelocity * 2;
    }
    else if (Yvelocity > 0)
    {
        Yvelocity = Yvelocity / 2;
    }
}
else if (enteredKey.KeyChar == 'a')
{
    if (Xvelocity == 0)
    {
        Xvelocity = -1;
    }
    else if (Xvelocity < 2 && Xvelocity > 0)
    {
        Xvelocity = 0;
    }
    else if (Xvelocity < 0)
    {
        Xvelocity = Xvelocity * 1.05;
    }
    else if (Xvelocity > 0)
    {
        Xvelocity = Xvelocity / 1.05;
    }
}
else if (enteredKey.KeyChar == 'A')
{
    if (Xvelocity == 0)
    {
        Xvelocity = -1;
    }
    else if (Xvelocity < 2 && Xvelocity > 0)
    {
        Xvelocity = 0;
    }
    else if (Xvelocity < 0)
    {
        Xvelocity = Xvelocity * 2;
    }
    else if (Xvelocity > 0)
    {
        Xvelocity = Xvelocity / 2;
    }
}
else if (enteredKey.KeyChar == 'd')
{
    if (Xvelocity == 0)
    {
        Xvelocity = 1;
    }
    else if (Xvelocity > -2 && Xvelocity < 0)
    {
        Xvelocity = 0;
    }
    else if (Xvelocity < 0)
    {
        Xvelocity = Xvelocity / 1.05;
    }
    else if (Xvelocity > 0)
    {
        Xvelocity = Xvelocity * 1.05;
    }
}
else if (enteredKey.KeyChar == 'D')
{
    if (Xvelocity == 0)
    {
        Xvelocity = 1;
    }
    else if (Xvelocity > -2 && Xvelocity < 0)
    {
        Xvelocity = 0;
    }
    else if (Xvelocity < 0)
    {
        Xvelocity = Xvelocity / 2;
```

```
                            }
                            else if (Xvelocity > 0)
                            {
                                Xvelocity = Xvelocity * 2;
                            }
                        }


                        newObjects = bodyCopy(Objects);

                        clearCentredStats(arrayWidth);

                        newObjects.Add(addObject(posX, posY, Xvelocity, Yvelocity, mass, radius,
name, colour));

                        displayPlanetStats(newObjects, newObjects.Count - 1, arrayWidth);

                        newObjects = bodyCopy(Objects);
                    }
                }

                largestDistanceBackup = largestDistance;

                while (simulating)
                {
                    collision = false;
                    Console.CursorVisible = false;
                    Console.CursorLeft = arrayWidth - 6;
                    Console.CursorTop = 1;
                    Console.Write(">SIMULATING<");
                    planetCount = newObjects.Count;

                    while (!Console.KeyAvailable)
                    {

                        newObjects = collisionChecker(newObjects, ref newtrailsQueue, trailLenght,
arrayHeight, arrayWidth, ref collision);

                        acceleration = accelerationCalc(newObjects);

                        newObjects = velocityUpdate(newObjects, acceleration, timePerTickInSec);

                        newObjects = positionUpdate(newObjects, timePerTickInSec);

                        if (timeLastImage + timeScale <= time)
                        {
```

```
                        if (int.Parse(DateTime.Now.Millisecond.ToString()) <
actualTimeMilliseconds + 30 && int.Parse(DateTime.Now.Millisecond.ToString()) >
actualTimeMilliseconds - 100)
                        {
                            System.Threading.Thread.Sleep(actualTimeMilliseconds + 30 -
int.Parse(DateTime.Now.Millisecond.ToString()));
                        }

                        visualPosition(newObjects, ref largestDistance, ref newtrailsQueue,
trailLenght, -1, 0, (int)moveX, (int)moveY, true, arrayHeight, arrayWidth);

                        Console.Write("Current Time Scale: " + secondsSimplifier(timeScale * 30)
+ " per second     ");
                        Console.Write("\nTime per tick: " + secondsSimplifier(timePerTickInSec) +
"     ");

                        if (!collision)
                        {
                            clearCentredStats(arrayWidth);
                            displayPlanetStats(newObjects, planetCount - 1, arrayWidth);
                        }

                        timeLastImage = time;
                        actualTimeMilliseconds = int.Parse(DateTime.Now.Millisecond.ToString());
                    }

                    time += timePerTickInSec;
                }

                enteredKey = Console.ReadKey(true);

                if (enteredKey.Key == ConsoleKey.OemComma)
                {
                    largestDistance = largestDistance * 1.1;
                    moveX = moveX / 1.1;
                    moveY = moveY / 1.1;
                    //zoom out
                    cleanTrails(ref newtrailsQueue, trailLenght, planetCount);
                    boxCleaner(arrayHeight, arrayWidth);
                }
                else if (enteredKey.Key == ConsoleKey.OemPeriod && largestDistance >
100000)
                {
                    largestDistance = largestDistance / 1.1;
                    moveX = moveX * 1.1;
                    moveY = moveY * 1.1;
                    //zoom in
```

```
                        cleanTrails(ref newtrailsQueue, trailLenght, planetCount);
                        boxCleaner(arrayHeight, arrayWidth);
                    }
                    else if (enteredKey.Key == ConsoleKey.RightArrow)
                    {
                        timeChanger(ref timeScale, ref timePerTickInSec, true);
                        //speed up
                    }
                    else if (enteredKey.Key == ConsoleKey.LeftArrow)
                    {
                        timeChanger(ref timeScale, ref timePerTickInSec, false);
                        //slow down
                    }
                    else if (enteredKey.Key == ConsoleKey.Enter)
                    {
                        simulating = false;
                        doneSimulating = true;
                    }
                    else if (enteredKey.Key == ConsoleKey.P)
                    {
                        accept = true;
                        simulating = false;
                        doneSimulating = true;
                    }

                }

                if (doneSimulating)
                {
                    Console.CursorVisible = true;

                    newObjects = bodyCopy(Objects);

                    newObjects.Add(addObject(posX, posY, Xvelocity, Yvelocity, mass, radius,
name, colour));

                    clearCentredStats(arrayWidth);
                    displayPlanetStats(newObjects, newObjects.Count - 1, arrayWidth);

                    planetCount = newObjects.Count;

                    boxCleaner(arrayHeight, arrayWidth);

                    Console.CursorLeft = arrayWidth - 8;
                    Console.CursorTop = 1;
                    Console.Write(">ADDING  PLANET<");
```

```
          largestDistance = largestDistanceBackup;

          doneSimulating = false;
        }

      }

      if (cancel)
      {
        boxCleaner(arrayHeight, arrayWidth);
        clearCentredStats(arrayWidth);
        return Objects;
      }

      visualPosition(newObjects, ref largestDistance, ref newtrailsQueue, trailLenght, -1, 0,
(int)moveX, (int)moveY, false, arrayHeight, arrayWidth);

      boxPrinter(5, 20, true, arrayWidth - 21, arrayHeight / 2 - 3);

      Console.CursorLeft = arrayWidth - 9;
      Console.CursorTop = arrayHeight / 2 - 1;
      Console.Write("Enter Planet Name:");
      Console.CursorVisible = false;

      bool takingInput = true;
      name = "";
      string[] colours = new string[] { "red", "dark red", "magenta", "dark magenta", "yellow",
"dark yellow", "green", "dark green", "blue", "dark blue", "cyan", "dark cyan", "white" };
      string gap = "";

      for (int i = 0; i < 20; i++)
      {
        gap += " ";
      }
      while (takingInput)
      {
        enteredKey = Console.ReadKey(true);
        if (enteredKey.Key == ConsoleKey.Enter && name.Length >= 2)
        {
          takingInput = false;
        }
        else if (enteredKey.Key != ConsoleKey.Backspace && enteredKey.Key !=
ConsoleKey.Enter && enteredKey.KeyChar != '\0' && enteredKey.KeyChar != '\t' &&
name.Length < 18)
        {
          name += enteredKey.KeyChar;
          gap = gap.Substring(gap.Length - 1);
```

```
    }
    else if (enteredKey.Key == ConsoleKey.Backspace)
    {
        if (name.Length > 0)
        {
            name = name.Substring(0, name.Length - 1);
            gap += " ";
        }
    }

    Console.CursorLeft = arrayWidth - 9;
    Console.CursorTop = arrayHeight / 2 + 1;
    Console.Write(name);
    Console.Write(gap);
}

Console.CursorLeft = arrayWidth - 9;
Console.CursorTop = arrayHeight / 2 - 1;
Console.Write("Select Planet Colour:");
Console.CursorVisible = false;

int colourID = 0;
takingInput = true;

while (takingInput)
{
    Console.CursorLeft = arrayWidth - 9;
    Console.CursorTop = arrayHeight / 2 + 1;

    Console.Write("← ");

    colourChanger(colours[colourID]);

    Console.Write(name);

    Console.ForegroundColor = ConsoleColor.Gray;

    Console.Write(" →");

    enteredKey = Console.ReadKey(true);

    if (enteredKey.Key == ConsoleKey.Enter)
    {
        colour = colours[colourID];

        takingInput = false;
    }
```

```
                else if (enteredKey.Key == ConsoleKey.LeftArrow)
                {
                    if (colourID - 1 > 0)
                    {
                        colourID--;
                    }
                    else
                    {
                        colourID = colours.Length - 1;
                    }
                }
                else if (enteredKey.Key == ConsoleKey.RightArrow)
                {
                    if (colourID + 1 < colours.Length)
                    {
                        colourID++;
                    }
                    else
                    {
                        colourID = 0;
                    }
                }

            }

            newObjects = bodyCopy(Objects);
            newPlanet = addObject(posX, posY, Xvelocity, Yvelocity, mass, radius, name,
colour);
            newObjects.Add(newPlanet);
            Console.Clear();

            return newObjects;
        }

        static List<Body> collision(List<Body> Objects, int object1, int object2)
        {
            List<Body> newObjects = new List<Body>();
            Body resultantObject;

            double newX = (Objects[object1].posX + Objects[object2].posX) / 2;

            double newY = (Objects[object1].posY + Objects[object2].posY) / 2;

            double newMass = Objects[object1].mass + Objects[object2].mass;

            double newXVelocity = Objects[object1].xVelocity * Objects[object1].mass +
Objects[object2].xVelocity * Objects[object2].mass;
```

```
        //energy = mass * velocity
        double newYVelocity = Objects[object1].yVelocity * Objects[object1].mass + 
Objects[object2].yVelocity * Objects[object2].mass;

        newXVelocity = newXVelocity / newMass;
        newYVelocity = newYVelocity / newMass;

        //energy / mass = velocity

        double newRadius;
        newRadius = Objects[object1].radius * Objects[object1].radius * 3.141 + 
Objects[object2].radius * Objects[object2].radius * 3.141; //total volume
        newRadius = Math.Sqrt(newRadius / 3.141);

        string newName;
        string newColour;
        int newID;
        int removedID;
        if (Objects[object1].mass > Objects[object2].mass)
        {
          newName = Objects[object1].name;
          newColour = Objects[object1].colour;
          newID = object1;
          removedID = object2;
        }
        else
        {
          newName = Objects[object2].name;
          newColour = Objects[object2].colour;
          newID = object2;
          removedID = object1;
        }

        resultantObject = addObject(newX, newY, newXVelocity, newYVelocity, newMass, 
newRadius, newName, newColour);

        for (int i = 0; i < Objects.Count; i++)
        {
          if (i != removedID)
          {
            if (i == newID)
            {
              newObjects.Add(resultantObject);
            }
            else
            {
              newObjects.Add(Objects[i]);
```

```
                    }
                }
            }

            return newObjects;
        }

        static List<Body> collisionChecker(List<Body> Objects, ref List<string[,]> trailsQueue,
int trailLenght, int arrayHeight, int arrayWidth, ref bool anyCollision)
        {
            List<double[]> line1 = new List<double[]> { };
            List<double[]> line2 = new List<double[]> { };
            double[] holerPoint = new double[] { 0, 0 };
            line1.Add(holerPoint);
            line1.Add(holerPoint);
            line2.Add(holerPoint);
            line2.Add(holerPoint);

            bool collisionOccured = false;
            double distance;
            for (int i = 0; i < Objects.Count - 1; i++)
            {
                for (int j = i + 1; j < Objects.Count; j++)
                {
                    distance = distanceCalc(Objects[i].posX, Objects[i].posY, Objects[j].posX,
Objects[j].posY);

                    if (Objects[i].radius + Objects[j].radius > distance)
                    {
                        collisionOccured = true;
                        anyCollision = true;
                    }
                    else
                    {
                        line1[0][0] = Objects[i].posX;
                        line1[0][1] = Objects[i].posY;
                        line1[1][0] = Objects[i].posX - Objects[i].xVelocity;
                        line1[1][1] = Objects[i].posY - Objects[i].yVelocity;

                        line2[0][0] = Objects[j].posX;
                        line2[0][1] = Objects[j].posY;
                        line2[1][0] = Objects[j].posX - Objects[j].xVelocity;
                        line2[1][1] = Objects[j].posY - Objects[j].yVelocity;

                        if (lineOverlapCheck(line1, line2))
                        {
                            collisionOccured = true;
                            anyCollision = true;
                        }
                    }
                    if (collisionOccured)
                    {
                        Objects = collision(Objects, i, j);
                        trailsQueue = trailQueueAdd(trailsQueue, trailLenght, Objects);
                        boxCleaner(arrayHeight, arrayWidth);
                        collisionOccured = false;
                    }
                }
            }
            return Objects;
        }

        static bool lineOverlapCheck(List<double[]> line1, List<double[]> line2)
        {
            bool overlap = false;
            if (arePointAntiClockwise(line1[0], line2[0], line2[1]) != arePointAntiClockwise(line1[1],
line2[0], line2[1]) && arePointAntiClockwise(line1[0], line1[1], line2[0]) !=
arePointAntiClockwise(line1[0], line1[1], line2[1]))
            {
                overlap = true;
            }
            else
            {
                overlap = false;
            }
            return overlap;
        }
        //using algorithm from
https://bryceboe.com/2006/10/23/line-segment-intersection-algorithm/
        static bool arePointAntiClockwise(double[] point1, double[] point2, double[] point3)
        {
            bool isAntiClockwise;
            if ((point3[1] - point1[1]) * (point2[0] - point1[0]) > (point2[1] - point1[1]) * (point3[0] -
point1[0]))
            {
                isAntiClockwise = true;
            }
            else
            {
                isAntiClockwise = false;
            }
            return isAntiClockwise;
        }
```

```
static string secondsSimplifier(int seconds)
{
    int minutes;
    int hours;
    int days;
    int weeks;
    if (seconds % 60 == 0)
    {
        minutes = seconds / 60;
        if (minutes % 60 == 0)
        {
            hours = minutes / 60;
            if (hours % 24 == 0)
            {
                days = hours / 24;
                if (Math.Round(days / 365.25) > 1)
                {
                    return Math.Round(days / 365.25) + " years";
                }
                if (days % 7 == 0)
                {
                    weeks = days / 7;
                    if (weeks > 1)
                    {
                        return weeks + " weeks";
                    }
                    else
                    {
                        return weeks + " week";
                    }
                }
                if (days > 1)
                {
                    return days + " days";
                }
                else
                {
                    return days + " day";
                }
            }
            if (hours > 1)
            {
                return hours + " hours"; ;
            }
            else
            {
                return hours + " hour";
```

```
                    }
                }
                if (minutes > 1)
                {
                    return minutes + " minutes";
                }
                else
                {
                    return minutes + " minute";
                }
            }
            if (seconds > 1)
            {
                return seconds + " seconds";
            }
            else
            {
                return seconds + " seconds";
            }


        }

        static void timeChanger(ref int timeScale, ref int timePerTickInSec, bool increase)
        {
            int[] timeScales = new int[] { 1, 5, 10, 30, 60, 120, 600, 1800, 3600, 7200, 14400,
43200, 86400, 172800, 345600, 604800, 1209600, 2419200 };
            //timeScales: 1s, 5s, 10s, 30s, 1min, 2mins, 10mins, 30mins, 1hour, 2hours, 4hours,
1day, 2days, 4days, 1week, 2weeks, 4weeks
            int scalePosition;
            int tickPosition;
            for (scalePosition = 0; scalePosition < timeScales.Length; scalePosition++)
            {
                if (timeScale == timeScales[scalePosition])
                {
                    break;
                }
            }
            for (tickPosition = 0; tickPosition < timeScales.Length; tickPosition++)
            {
                if (timePerTickInSec == timeScales[tickPosition])
                {
                    break;
                }
            }
            if (increase == true && scalePosition < timeScales.Count() - 1)
            {
```

```
            timeScale = timeScales[scalePosition + 1];
            if (tickPosition < scalePosition - 4 && timeScales[tickPosition + 1] <= 60 * 60)
            {
                timePerTickInSec = timeScales[tickPosition + 1];
            }
        }

        else if (increase == false && scalePosition > 0)
        {
            timeScale = timeScales[scalePosition - 1];

            if (tickPosition > scalePosition - 6 && tickPosition > 0)
            {
                timePerTickInSec = timeScales[tickPosition - 1];
            }
        }
    }

    static List<Body> objectsImporter(List<string> input)
    {
        List<Body> output;
        try
        {
            output = new List<Body> { };
            string[] splitInput2 = new string[8];

            for (int i = 0; i < input.Count(); i++)
            {
                splitInput2 = input[i].Split(',');
                output.Add(addObject(Convert.ToDouble(splitInput2[0]),
Convert.ToDouble(splitInput2[1]), Convert.ToDouble(splitInput2[2]),
Convert.ToDouble(splitInput2[3]), Convert.ToDouble(splitInput2[4]),
Convert.ToDouble(splitInput2[5]), splitInput2[6], splitInput2[7]));
            }
            return output;
        }
        catch (Exception e)
        {
            Console.Clear();
            Console.WriteLine("Input error, check that you have copied the text correctly");
            Console.WriteLine("Enter any key to return");
            Console.ReadKey(true);
            return null;
        }
    }
```

```
        static string objectsExporter(List<Body> input)
        {
            string output = "";
            for (int i = 0; i < input.Count(); i++)
            {
                output += input[i].posX.ToString() + ",";
                output += input[i].posY.ToString() + ",";
                output += input[i].xVelocity.ToString() + ",";
                output += input[i].yVelocity.ToString() + ",";
                output += input[i].mass.ToString() + ",";
                output += input[i].radius.ToString() + ",";
                output += input[i].name + ",";
                output += input[i].colour;
                output += "\n";
            }
            return output;
        }

        static void Main(string[] args)
        {
            List<Body> solarSystem = new List<Body>();
            List<Body> extraObjects = new List<Body>();
            List<Body> Objects = new List<Body>();

            solarSystem.Add(addObject(0, 0, 0, -0.19, 1.989e30, 696340000, "Sun", "yellow"));
            solarSystem.Add(addObject(5.791e10, 0, 0, 47.36e3, 3.285e23, 2439700, "Mercury",
    "gray"));
            solarSystem.Add(addObject(1.082e11, 0, 0, 35.02e3, 4.867e24, 6051800, "Venus",
    "gray"));
            solarSystem.Add(addObject(1.496e11, 0, 0, 29.78e3, 5.972e24, 6371000, "Earth",
    "green"));
            solarSystem.Add(addObject(2.279e11, 0, 0, 24.07e3, 6.39e23, 3396200, "Mars",
    "red"));

            solarSystem.Add(addObject(7.785e11, 0, 0, 13e3, 1.898e27, 69911000, "Jupiter",
    "dark yellow"));
            solarSystem.Add(addObject(1.434e12, 0, 0, 9.68e3, 5.683e26, 58232000, "Saturn",
    "dark yellow"));
            solarSystem.Add(addObject(2.871e12, 0, 0, 6.80e3, 8.681e25, 25362000, "Uranus",
    "dark cyan"));
            solarSystem.Add(addObject(4.495e12, 0, 0, 5.43e3, 1.024e26, 24622000,
    "Neptune", "dark blue"));

            extraObjects.Add(addObject(-2e9, 0, 0, 9.108e4, 9.945e29, 74085000, "1Sun",
    "yellow"));
            extraObjects.Add(addObject(2e9, 0, 0, -9.108e4, 9.945e29, 74085000, "2Sun",
    "yellow"));
```

```
int planetCount = Objects.Count;
ConsoleKey enteredKey = ConsoleKey.L;
string input;
List<string> newObjects = new List<string>();

double[,] acceleration;
bool simulationFinished = false;
bool paused = false;
bool alreadyPaused = false;
bool showingInstructions = true;
bool collision = false;
bool justOpened = true;
int centredType = 0;

int arrayHeight = Console.LargestWindowHeight - 4;
int arrayWidth = (Console.LargestWindowWidth - 50) / 2 - 2;

Console.WindowHeight = Console.LargestWindowHeight;
Console.WindowWidth = Console.LargestWindowWidth;

int centredPlanet = -1;
int minCentred = -1;

double moveX = 0;
double moveY = 0;

double largestDistance = largestDistanceFromCenter(Objects) * 1.1;
//largest dinstance that needs to be considered

double time = 1;
double timeLastImage = 1;

int timeScale = 1;
int timePerTickInSec = 1;

int actualTimeMilliseconds = int.Parse(DateTime.Now.Millisecond.ToString());

List<string[,]> trailsQueue = new List<string[,]> { };
int trailLenght = 13;

trailsQueue = trailQueueAdd(trailsQueue, trailLenght, Objects);

Console.WriteLine("Starting");
System.Threading.Thread.Sleep(500);

Console.Clear();
```

```
      Console.CursorVisible = false;
      Console.Title = "NEA";

      while (!simulationFinished)
      {
        if (!justOpened)
        {
          while (!Console.KeyAvailable)
          {

            Objects = collisionChecker(Objects, ref trailsQueue, trailLenght, arrayHeight,
arrayWidth, ref collision);
            //Checks if there are two bodies that overlap

            acceleration = accelerationCalc(Objects);
            //accelerations stored as (object, 0 and 1), where object referes to the
number that body is in the things array and 0 being x axis acceleration or 1 being y axis
acceleration

            Objects = velocityUpdate(Objects, acceleration, timePerTickInSec);
            //update the velocity of objects using the acceleration found

            Objects = positionUpdate(Objects, timePerTickInSec);
            //update the velocity of objects using the velocities

            if (timeLastImage + timeScale <= time)
            {
              if (int.Parse(DateTime.Now.Millisecond.ToString()) <
actualTimeMilliseconds + 30 && int.Parse(DateTime.Now.Millisecond.ToString()) >
actualTimeMilliseconds - 100)
              {
                System.Threading.Thread.Sleep(actualTimeMilliseconds + 30 -
int.Parse(DateTime.Now.Millisecond.ToString()));
              }

              visualPosition(Objects, ref largestDistance, ref trailsQueue, trailLenght,
centredPlanet, centredType, (int)moveX, (int)moveY, true, arrayHeight, arrayWidth);
              Console.Write("Current Time Scale: " + secondsSimplifier(timeScale * 30)
+ " per second     ");
              Console.Write("\nTime per tick: " + secondsSimplifier(timePerTickInSec) +
"     ");

              if (centredPlanet != -1)
              {
                clearCentredStats(arrayWidth);
                displayPlanetStats(Objects, centredPlanet, arrayWidth);
```

```
                        }

                        timeLastImage = time;
                        actualTimeMilliseconds = int.Parse(DateTime.Now.Millisecond.ToString());
                    }

                    time += timePerTickInSec;
                }

                enteredKey = Console.ReadKey(true).Key;
            }

            planetCount = Objects.Count;

            if (enteredKey == ConsoleKey.Spacebar)
            {
                paused = true;
            }
            else if (enteredKey == ConsoleKey.LeftArrow)
            {
                timeChanger(ref timeScale, ref timePerTickInSec, false);
                //speed down
            }
            else if (enteredKey == ConsoleKey.RightArrow)
            {
                if (largestDistanceFromCenter(Objects) > 1e9)
                {
                    timeChanger(ref timeScale, ref timePerTickInSec, true);
                    //speed up
                }
                else if (largestDistanceFromCenter(Objects) < 1e9 && timePerTickInSec < 60)
                {
                    timeChanger(ref timeScale, ref timePerTickInSec, true);
                }
            }
            else if (enteredKey == ConsoleKey.OemComma)
            {
                largestDistance = largestDistance * 1.1;
                moveX = moveX / 1.1;
                moveY = moveY / 1.1;
                //zoom out
                boxCleaner(arrayHeight, arrayWidth);
                cleanTrails(ref trailsQueue, trailLenght, planetCount);
                visualPosition(Objects, ref largestDistance, ref trailsQueue, trailLenght,
centredPlanet, centredType, (int)moveX, (int)moveY, true, arrayHeight, arrayWidth);
            }
            else if (enteredKey == ConsoleKey.OemPeriod && largestDistance > 10)
            {
                largestDistance = largestDistance / 1.1;
                moveX = moveX * 1.1;
                moveY = moveY * 1.1;
                //zoom in
                boxCleaner(arrayHeight, arrayWidth);
                cleanTrails(ref trailsQueue, trailLenght, planetCount);
                visualPosition(Objects, ref largestDistance, ref trailsQueue, trailLenght, centredPlanet, centredType, (int)moveX, (int)moveY, true, arrayHeight, arrayWidth);
            }
            else if (enteredKey == ConsoleKey.L)
            {
                Console.CursorVisible = true;
                Console.Clear();
                if (!justOpened)
                {
                    Console.WriteLine("(E)xporting current system, (I)mporting something new or (L)oading a preset? (C) to cancel");
                    enteredKey = Console.ReadKey(true).Key;
                    while (enteredKey != ConsoleKey.E && enteredKey != ConsoleKey.I && enteredKey != ConsoleKey.L && enteredKey != ConsoleKey.C)
                    {
                        enteredKey = Console.ReadKey(true).Key;
                    }
                }
                else
                {
                    Console.WriteLine("(I)mporting something or (L)oading a preset?");
                    enteredKey = Console.ReadKey(true).Key;
                    while (enteredKey != ConsoleKey.I && enteredKey != ConsoleKey.L)
                    {
                        enteredKey = Console.ReadKey(true).Key;
                    }
                }
                if (enteredKey == ConsoleKey.E)
                {
                    Console.Clear();
                    Console.Write(objectsExporter(Objects));
                    Console.Write("\n\nCopy text then enter R to return to simulation.");
                    enteredKey = Console.ReadKey(true).Key;
                    while (enteredKey != ConsoleKey.R)
                    {
                        enteredKey = Console.ReadKey(true).Key;
                    }
                }
                else if (enteredKey == ConsoleKey.I)
                {
```

```
                Console.Write("Enter import text. (C) to finish:\n");
                input = "";
                newObjects = new List<string> { };
                while (true)
                {
                    input = Console.ReadLine();
                    if (input.ToLower() != "c")
                    {
                        newObjects.Add(input);
                    }
                    else
                    {
                        break;
                    }
                }
                if (objectsImporter(newObjects) != null)
                {
                    if (newObjects.Count != 0)
                    {
                        Objects = objectsImporter(newObjects);
                        timeScale = 1;
                        timePerTickInSec = 1;
                        trailsQueue = new List<string[,]>();
                        trailsQueue = trailQueueAdd(trailsQueue, trailLenght, Objects);
                        largestDistance = largestDistanceFromCenter(Objects) * 1.1;


                        justOpened = false;
                    }
                    else
                    {
                        Console.Clear();
                        Console.WriteLine("Input error, check that you have copied the text
correctly");

                        Console.WriteLine("Enter any key to return");
                        Console.ReadKey(true);
                        enteredKey = ConsoleKey.L;
                    }
                }
                else
                {
                    enteredKey = ConsoleKey.L;
                }
            }
            else if (enteredKey == ConsoleKey.L)
            {
                Console.Clear();
                Console.Write("Presets:");
                Console.Write("\n1) inner solar system\n2) outer solar system\n3) whole solar
system\n4) binary stars\n5) solar system with binary stars");
                enteredKey = Console.ReadKey(true).Key;
                while (enteredKey != ConsoleKey.D1 && enteredKey != ConsoleKey.D2 &&
enteredKey != ConsoleKey.D3 && enteredKey != ConsoleKey.D4 && enteredKey !=
ConsoleKey.D5 && enteredKey != ConsoleKey.C)
                {
                    enteredKey = Console.ReadKey(true).Key;
                }
                if (enteredKey == ConsoleKey.D1)
                {
                    Objects = new List<Body> { };
                    for (int i = 0; i < 5; i++)
                    {
                        Objects.Add(solarSystem[i]);
                    }
                }
                else if (enteredKey == ConsoleKey.D2)
                {
                    Objects = new List<Body> { };
                    Objects.Add(solarSystem[0]);
                    for (int i = 5; i < 9; i++)
                    {
                        Objects.Add(solarSystem[i]);
                    }
                }
                else if (enteredKey == ConsoleKey.D3)
                {
                    Objects = new List<Body> { };
                    for (int i = 0; i < solarSystem.Count; i++)
                    {
                        Objects.Add(solarSystem[i]);
                    }
                }
                else if (enteredKey == ConsoleKey.D4)
                {
                    Objects = new List<Body> { };
                    Objects.Add(extraObjects[0]);
                    Objects.Add(extraObjects[1]);
                }
                else if (enteredKey == ConsoleKey.D5)
                {
                    Objects = new List<Body> { };
                    Objects.Add(extraObjects[0]);
                    Objects.Add(extraObjects[1]);
                    for (int i = 1; i < solarSystem.Count; i++)
```

```
                            {
                                Objects.Add(solarSystem[i]);
                            }
                        }
                        justOpened = false;
                        timeScale = 1;
                        timePerTickInSec = 1;
                        trailsQueue = new List<string[,]>();
                        trailsQueue = trailQueueAdd(trailsQueue, trailLenght, Objects);
                        largestDistance = largestDistanceFromCenter(Objects) * 1.1;
                    }
                    Console.CursorVisible = false;
                    Console.Clear();
                    boxPrinter(arrayHeight, arrayWidth);
                    printInstructions(showingInstructions, arrayWidth);
                }
                else if (enteredKey == ConsoleKey.W)
                {
                    moveY++;
                    boxCleaner(arrayHeight, arrayWidth);
                    visualPosition(Objects, ref largestDistance, ref trailsQueue, trailLenght,
centredPlanet, centredType, (int)moveX, (int)moveY, true, arrayHeight, arrayWidth);
                }
                else if (enteredKey == ConsoleKey.S)
                {
                    moveY--;
                    boxCleaner(arrayHeight, arrayWidth);
                    visualPosition(Objects, ref largestDistance, ref trailsQueue, trailLenght,
centredPlanet, centredType, (int)moveX, (int)moveY, true, arrayHeight, arrayWidth);
                }
                else if (enteredKey == ConsoleKey.A)
                {
                    moveX += 2;
                    boxCleaner(arrayHeight, arrayWidth);
                    visualPosition(Objects, ref largestDistance, ref trailsQueue, trailLenght,
centredPlanet, centredType, (int)moveX, (int)moveY, true, arrayHeight, arrayWidth);
                }
                else if (enteredKey == ConsoleKey.D)
                {
                    moveX -= 2;
                    boxCleaner(arrayHeight, arrayWidth);
                    visualPosition(Objects, ref largestDistance, ref trailsQueue, trailLenght,
centredPlanet, centredType, (int)moveX, (int)moveY, true, arrayHeight, arrayWidth);
                }
                else if (enteredKey == ConsoleKey.R)
                {
                    moveX = 0;
                    moveY = 0;
                    largestDistance = largestDistanceFromCenter(Objects) * 1.1;
                    boxCleaner(arrayHeight, arrayWidth);
                    cleanTrails(ref trailsQueue, trailLenght, planetCount);
                    visualPosition(Objects, ref largestDistance, ref trailsQueue, trailLenght,
centredPlanet, centredType, (int)moveX, (int)moveY, true, arrayHeight, arrayWidth);
                }
                else if (enteredKey == ConsoleKey.I)
                {
                    showingInstructions = !showingInstructions;
                    if (showingInstructions)
                    {
                        Console.Clear();
                        arrayWidth = (Console.WindowWidth - 50) / 2 - 2;
                    }
                    else
                    {
                        arrayWidth = (Console.WindowWidth - 30) / 2 - 2;
                    }
                    printInstructions(showingInstructions, arrayWidth);
                    cleanTrails(ref trailsQueue, trailLenght, planetCount);
                    boxCleaner(arrayHeight, arrayWidth);
                    boxPrinter(arrayHeight, arrayWidth);
                }
                else if (enteredKey == ConsoleKey.P)
                {
                    Objects = addPlanet(Objects, arrayHeight, arrayWidth, largestDistance,
trailsQueue, trailLenght, moveX, moveY);
                    //Objects = new List<Body>();
                    //Objects = holder;
                    //holder = new List<Body>();
                    trailsQueue = trailQueueAdd(trailsQueue, trailLenght, Objects);
                    printInstructions(showingInstructions, arrayWidth);
                    boxCleaner(arrayHeight, arrayWidth);
                    boxPrinter(arrayHeight, arrayWidth);
                }
                else if (enteredKey == ConsoleKey.Q)
                {
                    centredPlanet--;
                    if (centredPlanet < minCentred)
                    {
                        centredPlanet = planetCount - 1;
                    }
                    //cycle planets backwards

                    if (centredType != 0)
                    {
```

```
        cleanTrails(ref trailsQueue, trailLenght, planetCount);
        boxCleaner(arrayHeight, arrayWidth);
        moveX = 0;
        moveY = 0;
    }
    else
    {
        clearCentredStats(arrayWidth);
    }
}
else if (enteredKey == ConsoleKey.E)
{
    centredPlanet++;
    if (centredPlanet >= planetCount)
    {
        centredPlanet = minCentred;
    }
    //cycle planets forwards

    if (centredType != 0)
    {
        cleanTrails(ref trailsQueue, trailLenght, planetCount);
        boxCleaner(arrayHeight, arrayWidth);
        moveX = 0;
        moveY = 0;
    }
    else
    {
        clearCentredStats(arrayWidth);
    }
}
else if (enteredKey == ConsoleKey.C)
{
    //cycle centre view type
    if (centredPlanet > -1)
    {
        if (centredType < 2)
        {
            centredType++;
            minCentred = 0;
            if (centredType == 2)
            {
                boxCleaner(arrayHeight, arrayWidth);
                cleanTrails(ref trailsQueue, trailLenght, planetCount);
            }
        }
        else
        {
            centredType = 0;
            moveX = 0;
            moveY = 0;
            minCentred = -1;
            boxCleaner(arrayHeight, arrayWidth);
            cleanTrails(ref trailsQueue, trailLenght, planetCount);
        }
    }
    printInstructions(showingInstructions, arrayWidth);
}
else if (enteredKey == ConsoleKey.X)
{
    boxPrinter(5, 20, true, arrayWidth - 21, arrayHeight / 2 - 3);

    Console.CursorLeft = arrayWidth - 9;
    Console.CursorTop = arrayHeight / 2 - 2;
    Console.Write("Removing:");

    Console.CursorLeft = arrayWidth - 9;
    Console.CursorTop = arrayHeight / 2 - 1;
    Console.Write(Objects[centredPlanet].name);

    Console.CursorLeft = arrayWidth - 9;
    Console.CursorTop = arrayHeight / 2;
    Console.Write("Are you sure?(Y/N)");

    if (Console.ReadKey().Key == ConsoleKey.Y)
    {
        Objects = planetRemover(Objects, centredPlanet);
        centredPlanet = -1;
        trailsQueue = trailQueueAdd(trailsQueue, trailLenght, Objects);
        boxCleaner(arrayHeight, arrayWidth);
        clearCentredStats(arrayWidth);
    }

}
while (paused)
{
    if (alreadyPaused == false)
    {
        Console.CursorLeft = arrayWidth - 4;
        Console.CursorTop = 1;
        Console.Write(">PAUSED<");
        Console.CursorLeft = 0;
        Console.CursorTop = arrayHeight + 4;
        alreadyPaused = true;
```

```
                    }

                    enteredKey = Console.ReadKey(true).Key;

                    if (enteredKey == ConsoleKey.Spacebar)
                    {
                        paused = false;
                        alreadyPaused = false;
                        Console.CursorLeft = arrayWidth - 4;
                        Console.CursorTop = 1;
                        Console.Write("        ");
                    }
                    System.Threading.Thread.Sleep(50);
                }
                System.Threading.Thread.Sleep(30);
            }

            //visualPosition(things, ref largestDistance, ref trails, trailLenght, -1);
            //Console.Write("\nfinished");
            //Console.ReadKey();
        }

        static void displayPlanetStats(List<Body> things, int centredPlanet, int arrayWidth)
        {
            int row = 0;
            scroll(arrayWidth, row);
            Console.Write("Centred Planet:");
            row++;
            scroll(arrayWidth, row);
            Console.Write("Name: " + things[centredPlanet].name);
            row++;
            scroll(arrayWidth, row);
            Console.Write("X: " + Math.Round(things[centredPlanet].posX) + "m");
            row++;
            scroll(arrayWidth, row);
            Console.Write("Y: " + Math.Round(things[centredPlanet].posY) + "m");
            row++;
            scroll(arrayWidth, row);
            Console.Write("X Velocity: " + Math.Round(things[centredPlanet].xVelocity, 3) +
"m/s");
            row++;
            scroll(arrayWidth, row);
            Console.Write("Y Velocity: " + Math.Round(things[centredPlanet].yVelocity, 3) +
"m/s");
            row++;
            scroll(arrayWidth, row);
            Console.Write("Mass: " + things[centredPlanet].mass + "kg");
```

```
    row++;
    scroll(arrayWidth, row);
    Console.Write("Radius: " + things[centredPlanet].radius + "m");
}

static void clearCentredStats(int arrayWidth)
{
    string clear = "";
    for (int i = 0; i < Console.WindowWidth - (arrayWidth * 2 + 2); i++)
    {
        clear += " ";
    }
    for (int row = 0; row < 8; row++)
    {
        scroll(arrayWidth, row);
        Console.Write(clear);
    }
}

static void printInstructions(bool open, int arrayWidth,
    bool forPlanetAdder = false, bool editingVelocity = false)
{
    int row = 8;
    string spacer = "";
    string whatIsEdited;
    for (int i = 0; i < (Console.WindowWidth - (arrayWidth * 2 + 2) - 12) / 2 - 2; i++)
    {
        spacer += "-";
    }
    scroll(arrayWidth, row);
    Console.Write(spacer + "INSTRUCTIONS" + spacer);
    row++;
    if (open)
    {
        if (forPlanetAdder)
        {
            if (editingVelocity)
            {
                whatIsEdited = "velocity";
            }
            else
            {
                whatIsEdited = "position";
            }
            for (int i = 0; i < Console.WindowWidth - (arrayWidth * 2 + 2); i++)
            {
                spacer += "-";
            }
            scroll(arrayWidth, row);
            Console.Write("WASD to edit the planets' " + whatIsEdited + " quickly");
            row++;
            scroll(arrayWidth, row);
            Console.Write("wasd to edit the planets' " + whatIsEdited + " accurately");
            row++;
            scroll(arrayWidth, row);
            Console.Write("SPACE to switch editing velocity and position");
            row++;
            scroll(arrayWidth, row);
            Console.Write("UP/DOWN ARROW to edit mass");
            row++;
            scroll(arrayWidth, row);
            Console.Write("LEFT/RIGHT ARROW to edit radius");
            row++;
            scroll(arrayWidth, row);
            Console.Write("C to cancel");
            row++;
            scroll(arrayWidth, row);
            Console.Write("ENTER to start / reset simulation");
            row++;
            scroll(arrayWidth, row);
            Console.Write("< and > to zoom in/out");
            row++;
            scroll(arrayWidth, row);
            Console.Write("LEFT/RIGHT ARROW to change the simulation speed");
        }
        else
        {
            scroll(arrayWidth, row);
            Console.Write("WASD to move camera");
            row++;
            scroll(arrayWidth, row);
            Console.Write("< > to zoome in/out");
            row++;
            scroll(arrayWidth, row);
            Console.Write("LEFT/RIGHT ARROW to change simulation speed");
            row++;
            scroll(arrayWidth, row);
            Console.Write("P to add planet");
            row++;
            scroll(arrayWidth, row);
            Console.Write("Q and E to select planets");
            row++;
            scroll(arrayWidth, row);
            Console.Write("C to centre on selected planet");
```

```
            row++;
            scroll(arrayWidth, row);
            Console.Write("R to reset camera");
            row++;
            scroll(arrayWidth, row);
            Console.Write("SPACE to pause");
            row++;
            scroll(arrayWidth, row);
            Console.Write("X to remove selected planet");
            row++;
            scroll(arrayWidth, row);
            Console.Write("L to load/import/export systems");
            row++;
            scroll(arrayWidth, row);
            Console.Write("I to hide this menu");
        }
    }
    else
    {
        string clear = "";
        for (int i = 0; i < Console.WindowWidth - (arrayWidth * 2 + 2); i++)
        {
            clear += " ";
        }
        for (row = 9; row < 20; row++)
        {
            scroll(arrayWidth, row);
            Console.Write(clear);
        }
    }
}

static void boxPrinter(int arrayHeight, int arrayWidth,
    bool empty = false, int x = 0, int y = 0)
{
    string characterCounter = "";
    for (int i = 0; i < (arrayWidth + 1) * 2; i++)
    {
        characterCounter += "-";
    }

    Console.CursorLeft = x;
    Console.CursorTop = y;
    Console.Write(characterCounter);
    Console.CursorTop = y + 1;
    for (int i = 0; i < arrayHeight + 1; i++)
    {
```

```
                Console.CursorLeft = x;
                Console.Write("|");
                if (empty)
                {
                    for (int j = 0; j < arrayWidth * 2; j++)
                    {
                        Console.Write(" ");
                    }
                }
                Console.CursorLeft = x + arrayWidth * 2 + 1;
                Console.Write("|");
                Console.CursorTop = y + i + 1;
            }
            Console.CursorLeft = x;
            Console.WriteLine(characterCounter);
        }

        static void boxCleaner(int arrayHeight, int arrayWidth)
        {
            string clear = "";
            for (int i = 0; i < arrayWidth * 2; i++)
            {
                clear += " ";
            }
            for (int i = 1; i < arrayHeight + 1; i++)
            {
                Console.CursorLeft = 1;
                Console.CursorTop = i;
                Console.Write(clear);
            }
        }

        static void cursorArrayPrinter(List<Body> things, int[,] xyPos, int moveX, int moveY, int
arrayHeight, int arrayWidth)
        {
            //boxPrinter(arrayHeight, arrayWidth);

            for (int i = 0; i < things.Count; i++)
            {
                if (xyPos[i, 0] != -1)
                {
                    if (xyPos[i, 1] * 2 + 1 + moveX < arrayWidth * 2 && xyPos[i, 2] + moveY <
arrayHeight)
                    {
                        if (xyPos[i, 1] * 2 + 1 + moveX > 0 && xyPos[i, 2] + 1 + moveY > 0)
                        {
                            Console.CursorLeft = xyPos[i, 1] * 2 + 1 + moveX;
                            Console.CursorTop = xyPos[i, 2] + 1 + moveY;
                            colourChanger(things[xyPos[i, 0]].colour);
                            Console.Write(things[xyPos[i, 0]].name.Substring(0, 2));
                            Console.ForegroundColor = ConsoleColor.Gray;
                        }
                    }
                }
            }
            Console.CursorTop = arrayHeight + 2;
            Console.CursorLeft = 0;
        }

        static void cycleTrailQueue(int posX, int posY, ref List<string[,]> trailsQueue, string
BodyName, int trailLenght, int moveX, int moveY, int arrayHeight, int arrayWidth)
        {
            for (int i = 0; i < trailsQueue.Count; i++)
            {
                if (BodyName == trailsQueue[i][0, 0])
                {
                    for (int j = 0; j < trailLenght; j++)
                    {
                        if (trailsQueue[i][j, 3] == null)
                        {
                            if (posX * 2 + 1 + moveX < arrayWidth * 2 && posY + moveY <
arrayHeight)
                            {
                                if (posX + 1 + moveX > 0 && posY + 1 + moveY > 0)
                                {
                                    trailsQueue[i][j, 1] = posX.ToString();
                                    trailsQueue[i][j, 2] = posY.ToString();
                                    trailsQueue[i][j, 3] = trailLenght.ToString();
                                }
                            }
                            break;
                        }
                        else
                        {
                            if (int.Parse(trailsQueue[i][j, 3]) <= 0)
                            {
                                if (int.Parse(trailsQueue[i][j, 1]) * 2 + 1 + moveX > 0 &&
int.Parse(trailsQueue[i][j, 2]) + 1 + moveY > 0)
                                {
                                    if (int.Parse(trailsQueue[i][j, 1]) * 2 + 1 + moveX < arrayWidth * 2 &&
int.Parse(trailsQueue[i][j, 2]) + moveY < arrayHeight)
                                    {
                                        Console.CursorLeft = int.Parse(trailsQueue[i][j, 1]) * 2 + 1 + moveX;
                                        Console.CursorTop = int.Parse(trailsQueue[i][j, 2]) + 1 + moveY;
                                        Console.Write("  ");
                                    }
                                }
                                trailsQueue[i][j, 1] = posX.ToString();
                                trailsQueue[i][j, 2] = posY.ToString();
                                trailsQueue[i][j, 3] = trailLenght.ToString();

                                break;
                            }
                        }
                    }
                    break;
                }
            }
        }

        static int[,] XYPosToArray(List<Body> things, ref double distanceScale, int arrayHeight,
int arrayWidth, int[,] xyPos)
        {
            double tablePosX;
            double tablePosY;

            int Adjust;
            Adjust = (arrayWidth - 1) / 2 - (arrayHeight - 1) / 2;

            for (int i = 0; i < things.Count; i++)
            {
                tablePosX = (arrayHeight - 1) / 2 + (things[i].posX / distanceScale) * (arrayHeight -
1) / 2 + Adjust;

                tablePosY = (arrayHeight - 1) / 2 - (things[i].posY / distanceScale) * (arrayHeight -
1) / 2;

                xyPos[i, 0] = (int)Math.Round(tablePosX);
                xyPos[i, 1] = (int)Math.Round(tablePosY);


            }
            return xyPos;
        }

        static void trailPrinter(List<string[,]> trailsQueue, int trailLenght, int moveX, int moveY,
int arrayHeight, int arrayWidth)
        {
            Console.ForegroundColor = ConsoleColor.DarkGray;
            for (int i = 0; i < trailsQueue.Count; i++)
            {
                for (int j = 0; j < trailLenght; j++)
                {
                    if (trailsQueue[i][j, 3] != null)
                    {
                        if (int.Parse(trailsQueue[i][j, 1]) * 2 + 1 + moveX < arrayWidth * 2 &&
int.Parse(trailsQueue[i][j, 2]) + moveY < arrayHeight)
                        {
                            if (int.Parse(trailsQueue[i][j, 1]) * 2 + 1 + moveX > 0 &&
int.Parse(trailsQueue[i][j, 2]) + 1 + moveY > 0)
                            {
                                Console.CursorLeft = int.Parse(trailsQueue[i][j, 1]) * 2 + 1 + moveX;
                                Console.CursorTop = int.Parse(trailsQueue[i][j, 2]) + 1 + moveY;
                                Console.Write(trailsQueue[i][j, 0].Substring(0, 2));
                            }
                        }
                        trailsQueue[i][j, 3] = (int.Parse(trailsQueue[i][j, 3]) - 1).ToString();
                    }
                }
            }
            Console.ForegroundColor = ConsoleColor.Gray;
            Console.CursorTop = arrayHeight + 2;
            Console.CursorLeft = 0;
        }

        static void centreView(List<Body> things, int[,] xyPos, int centredPlanet, ref int moveX,
ref int moveY, int arrayHeight, int arrayWidth)
        {
            int centreX = (int)Math.Round((double)(arrayWidth) / 2);
            int centreY = (int)Math.Round((double)(arrayHeight) / 2);

            int xCentreDiff = centreX - xyPos[centredPlanet, 0];
            int yCentreDiff = centreY - xyPos[centredPlanet, 1];

            moveX += xCentreDiff * 2;
            moveY += yCentreDiff;
        }

        static int[,] altcentreView(List<Body> things, int[,] xyPos, int centredPlanet, int moveX,
int moveY, int arrayHeight, int arrayWidth, ref List<string[,]> trailsQueue, int trailLenght)
        {
            int centreX = (int)Math.Round((double)(arrayWidth) / 2);
            int centreY = (int)Math.Round((double)(arrayHeight) / 2);

            int xCentreDiff = centreX - xyPos[centredPlanet, 0];
            int yCentreDiff = centreY - xyPos[centredPlanet, 1];

            for (int i = 0; i < things.Count; i++)
            {
                xyPos[i, 0] = xyPos[i, 0] + xCentreDiff;
                xyPos[i, 1] = xyPos[i, 1] + yCentreDiff;
                cycleTrailQueue(xyPos[i, 0], xyPos[i, 1], ref trailsQueue,
things[i].name.Substring(0, 2), trailLenght, moveX, moveY, arrayHeight, arrayWidth);
            }
            return xyPos;
        }

        static int[,] removeOverlap(List<Body> things, int[,] xyPos)
        {
            int[,] newXY = new int[things.Count, 3];
            for (int i = 0; i < things.Count; i++)
            {
                for (int j = 0; j < 2; j++)
                {
                    newXY[i, j + 1] = xyPos[i, j];
                }
                newXY[i, 0] = i;
            }

            for (int i = 0; i < things.Count - 1; i++)
            {
                for (int j = i + 1; j < things.Count; j++)
                {
                    if (newXY[i, 0] != -1 && newXY[j, 0] != -1)
                    {
                        if (newXY[i, 1] == newXY[j, 1] && newXY[i, 2] == newXY[j, 2])
                        {
                            if (things[newXY[i, 0]].mass > things[newXY[j, 0]].mass)
                            {
                                newXY[j, 0] = -1;
                            }
                            else
                            {
                                newXY[i, 0] = -1;
                            }
                        }
                    }
                }
            }

            return newXY;
        }
        static void visualPosition(List<Body> things, ref double largestDistance, ref List<string[,]> trailsQueue, int trailLenght, int centredPlanet, int centredType, int moveX, int moveY, bool useTrails, int arrayHeight, int arrayWidth)
        {

            int[,] xyPos = new int[things.Count, 2];
            string[,] table = new string[arrayHeight, arrayWidth];

            XYPosToArray(things, ref largestDistance, arrayHeight, arrayWidth, xyPos);

            if (things.Count == 2)
            {
                boxCleaner(arrayHeight, arrayWidth);
            }

            if (centredType == 0)
            {
                for (int i = 0; i < things.Count; i++)
                {
                    cycleTrailQueue(xyPos[i, 0], xyPos[i, 1], ref trailsQueue, things[i].name, trailLenght, moveX, moveY, arrayHeight, arrayWidth);
                }
            }

            if (centredPlanet > -1)
            {
                if (centredType == 1)
                {
                    centreView(things, xyPos, centredPlanet, ref moveX, ref moveY, arrayHeight, arrayWidth);
                    for (int i = 0; i < things.Count; i++)
                    {
                        cycleTrailQueue(xyPos[i, 0], xyPos[i, 1], ref trailsQueue, things[i].name.Substring(0, 2), trailLenght, moveX, moveY, arrayHeight, arrayWidth);
                    }
                    boxCleaner(arrayHeight, arrayWidth);
                }
                else if (centredType == 2)
                {
                    xyPos = altcentreView(things, xyPos, centredPlanet, moveX, moveY, arrayHeight, arrayWidth, ref trailsQueue, trailLenght);
                }
            }

            if (useTrails)
            {
                trailPrinter(trailsQueue, trailLenght, moveX, moveY, arrayHeight, arrayWidth);
            }

            xyPos = removeOverlap(things, xyPos);

            cursorArrayPrinter(things, xyPos, moveX, moveY, arrayHeight, arrayWidth);
        }
    }
```