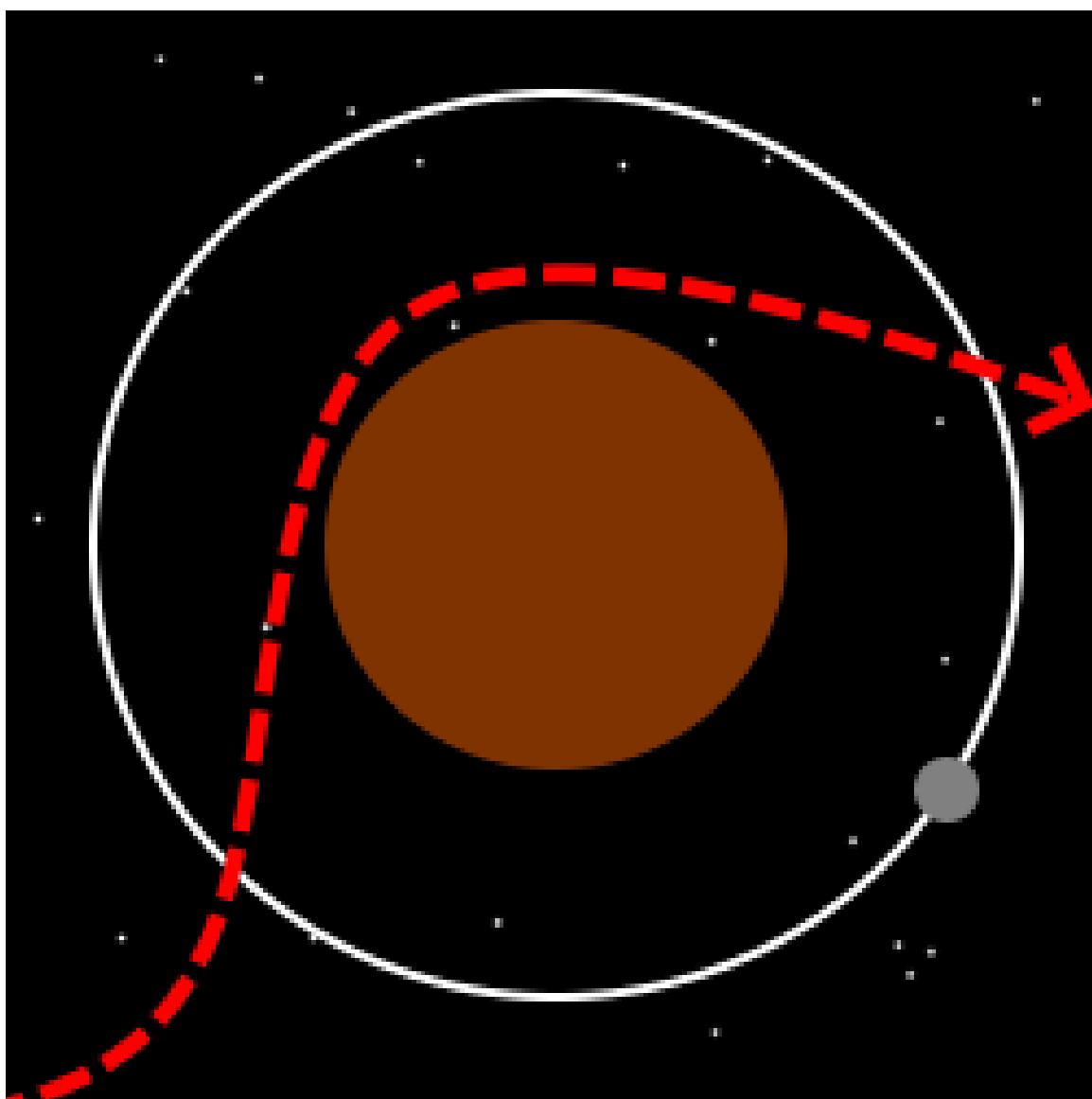


Orbital Mechanics Simulator

Sam Prince | Candidate number 6103 | Centre number 64859
St George's College, Weybridge



Contents

Analysis.....	3
Problem identification	3
Thinking Abstractly	3
Thinking Ahead.....	4
Thinking Procedurally	4
Thinking Logically	5
Thinking Concurrently.....	5
Features of the proposed solution	5
Stakeholders.....	6
Stakeholder interviews	6
Research.....	12
Saving/loading of scenarios	16
Success Criteria.....	16
Hardware/software requirements	20
Limitations.....	20
Design	21
Hierarchy chart.....	22
Class Diagrams.....	23
Algorithms/logic	28
Data structures.....	33
GUI.....	36
Usability	39
Post-dev test plans	39
Development	42
Stage 1 – Main menu, creating and drawing planets/stars to the screen	42
Success criteria	42
Testing plan	42
Development.....	43
Testing + Bug fixing	57
Stakeholder Feedback.....	59
Stage 2 – Physics engine and camera movement	59
Success criteria	59
Testing plan	60
Development.....	60

Testing + Bug fixing	74
Stakeholder Feedback.....	77
Stage 3 – Orbit/trajectory and equipotential lines.....	77
Success criteria	77
Testing plan	78
Development.....	78
Testing + Bug fixing	97
Stakeholder Feedback.....	99
Stage 4 – Object manipulation	99
Success criteria	99
Testing plan	100
Development.....	101
Testing + Bug fixing	116
Stakeholder Feedback.....	123
Stage 5 – Build menu and spacecraft	123
Success criteria	123
Testing plan	123
Development.....	124
Testing + Bug fixing	133
Stakeholder Feedback.....	134
Stage 6 – Accessing the program, saving and loading scenarios.....	134
Success criteria	134
Testing plan	135
Development.....	135
Testing + Bug fixing	144
Stakeholder Feedback.....	145
Stage 7 – Misc features, changes and fixes.....	145
Success criteria	145
Testing plan	146
Development.....	146
Testing + Bug fixing	152
Stakeholder Feedback.....	154
Evaluation.....	155
Post-dev alpha testing	155
Stakeholder beta testing/final feedback	165
Success criteria comparison	169

Success of usability features	173
Maintainability	173
Approaching limitations.....	174
Final Code	175
main.py	175
engine.py.....	189
References	191

Analysis

Problem identification

Orbital mechanics can be difficult to explain to students, and there is a limited number of simulations online that would allow students to better understand this concept. As a physics student myself, I am eager to be able to apply concepts I have learnt in class to applications such as this not only to consolidate my own knowledge, but also to teach others. This project's purpose is to provide a visual, physics-based sandbox in which students and teachers can manipulate objects around planets and stars to simulate the effects of gravity.

The end user will be physics students or teachers that will be able to use my software to enhance learning. Currently, teachers at my school use a variety of different PHET simulations in order to demonstrate individual concepts. The issue with these is that they are usually not customisable enough to demonstrate how different concepts can be applied in multiple situations. As many of them focus on the relationship between only two or three variables, I intend for my software to demonstrate how many variables work together to produce the effects we can observe in reality.

I predict that this project will develop in either of two directions:

1. A game with various stages/levels based on launching spacecraft to separate locations using gravitational fields.
2. A simulation of the solar system, with the ability to change speeds and masses of planets, possibly to add new planets as well. The user would be able to create a spacecraft object which they would be able to control using the keyboard.

Most of the complexity of this project will come with the creation of a physics engine from scratch. I have decided to program this in Python, not only due to my familiarity with the language but also due to its versatility and number of useful libraries available.

Orbits can be difficult to understand without a visual aid; consequently, a visual solution is required to illustrate this concept. Furthermore, orbits can be difficult and time-consuming to calculate and predict by hand, and so accurate calculations completed by a computer are the best way to correctly convey this concept. Therefore, a computational solution is ideal to solve this problem.

Thinking Abstractly

Abstraction is important for this solution as in reality, there are many different factors that affect the movement of celestial bodies that for this purpose are completely out of the scope of the project.

Therefore, I will only be considering the major factors in an object's orbit such as its mass and radius, as well as its velocity.

Many graphical aspects of the solution can be ignored, as this is an educational simulation and as such is not intended to have looks as a high priority. I will also be programming this solution to be 2D, as I believe that coding in 3D will not be worth the added educational value compared to the time it would take to implement.

Another aspect that can be abstracted is the way that objects are displayed; I intend for the objects to be simple shapes, as they do not need to look fancy in order to work and demonstrate physics concepts. What is important, however, is that users can tell what an object is meant to be at a glance.

In reality, planets and stars will not be the same as the ideal models that I will use in my solution. As this is intended to be used by GCSE/A level physics students, the level of understanding they should have is only limited to these ideal bodies that are used as such in the equations they learn; creating anything that is overcomplex may be unhelpful or even misleading.

Thinking Ahead

The inputs of the solution will be the text input for changing variables of objects, the mouse click for selecting buttons and possibly arrow keys/WASD for any camera controls, if required. The outputs will be the effects of the physics engine on the behaviour of the objects, and how these objects are displayed to the user.

Caching frequently used variables is beneficial for performance as while the added variable may take up one more space in memory, it saves the process of having to call or process a value from another method multiple times in the same procedure. However, using many different aliases for variables may clutter the code and as such make it less readable/maintainable.

Reusable components are also ideal for these same reasons, and also make changes that affect multiple parts of the program easy to change or keep track of. The number of points of failure in the code is also reduced when re-using a component as it will be consistent no matter where in the program it has been called.

Thinking Procedurally

The core components of this problem are:

- Creating accurate and scale simulations of objects in motion
- Illustrating these motions in an easy to interpret form
- Making the processes behind the physics of the simulation predictable and easy to understand by students
- Providing an accessible platform for all students and teachers to be able to easily use

Therefore, the components of the solution to this problem should include:

- The calculations that need to be done each tick for the physics
- The display of the simulation to the user
- How the user interacts with the program
- The overall performance of the system
- The accessibility/usability of the system both by students and by teachers

In terms of the calculations needed, I will use the actual calculations also used in the physics curriculum in order to process all logic for the simulation. This ensures that it can be understood by students and consequently be useful in lessons.

To display the simulation to the user, I can use the functionality provided by the Pygame library to create a window independent of the console where I can use the library's sprite objects to draw what is needed to the screen.

The controls should be simple and familiar to students and teachers alike in order to not disrupt learning due to any difficulty with correctly manipulating the simulation.

The system can be optimized through continual testing of both individual modules and the system as a whole, in order to easily track and fix areas that negatively impact performance.

The system should also be accessible by all students in a classroom to a reasonable extent; it is unacceptable to exclude groups of students from being able to benefit from this tool.

Thinking Logically

Decisions will need to be made by the program when the user enters certain data that will affect other variables as part of that object; for example, if surface gravity of an object is changed, the program must decide whether to change that object's radius or mass. This should be determined by what the user wants to stay constant in the calculation; in the physics curriculum, there are questions in which a variable is changed, and the student is asked by how much another variable changes if all else remains constant. This will allow for this type of question to be visualised by students, with realistic effects that are reflected in the behaviour of the objects.

Other than this, the program will just process a set of predetermined calculations each tick. The main logic of the software will run in a loop, in which the user's input and its effects will be processed and the result of the physics calculations on each object's movement and position will need to be calculated before the objects are drawn on the screen.

Thinking Concurrently

Many calculations will have to be made in the same loop cycle for the physics effects to be calculated according to reality. For this reason, processing all the logic of calculating movement for each object would be ideal as nothing should change in terms of input data between calculations for each celestial object. It may be more challenging to implement, however, as I have no previous experience with concurrent programming.

Features of the proposed solution

The program will feature customisable stars and planets that can be placed and edited to create any scenario, which will then be fully simulated. This allows for teachers to show the application of certain concepts in multiple situations, and so build understanding in students. A spacecraft object will be able to be used to navigate through the scenario to allow for the movement of real-life spacecraft to be shown. The paths of objects' orbits will be predicted and drawn on the screen by the system to aid in visualising the movement of objects.

There will also be a save and load system for users to be able to save scenarios to load at a later date, which will also allow for preset scenarios to be created to save time when a specific scenario is needed. This can also act as a 'reset' button if a 'reset' button is needed to set the scenario back up again if changes are made to it.

The program will be written in Python with the Pygame¹ library due to my familiarity with it and its use in displaying information to the screen.

Stakeholders

Stakeholder	Role	Justification
Mr Douglas-Scott	Teacher of Physics	Knowledge of physics and judge of accuracy of my simulation. Would be able to give feedback on how this simulation could be used in a classroom and use it as an aid to instruct students.
Kai Loach	Physics and computer science student	Would be able to give feedback on how useful the simulation is in aiding learning. As a computer science student, would likely be able to find and exploit any bugs in the program.
Oli Parks	Computer science student	As above, would likely be able to find bugs and provide feedback to a smaller extent of how well they understood the simulation.
ChatGPT (May 24 2023 Version) ChatGPT (openai.com)	AI language model – used for initial stakeholder interview	I thought it would be interesting to ask ChatGPT to respond to questions from my survey. However, its feedback will be of secondary significance to the other stakeholders as it may produce inaccurate information and will be unable to provide feedback on the program itself. GPT is purely being used to generate suggestions or ideas that may be useful to implement in this project and <u>will NOT be used to write code or documentation</u> .

Stakeholder interviews

To find out what my stakeholders are looking for in such a program, I have devised a set of ten questions to ask them. These questions have been designed that I will be able to gain a significant insight into the stakeholders' requirements for the project, and details of any other features that they might want.

The following is the responses to these questions from each of my stakeholders:

1. What do students need to learn from this simulation?

Mr Douglas-Scott – The effects that changing radius, mass and velocity have on the orbits of various celestial bodies and planets. It should simply show the relationship between orbital period, velocity and mass and radius.

Kai – How distance and other factors affects period of orbits – how the equations apply to real life. Students need to learn about the proportion between the velocity of the object and the mass and radius of the planet whilst also showing the time period for each one.

Oli – How different types of objects would orbit. Like, how objects of different mass or speed would orbit.

¹ (Pygame Documentation, n.d.)

ChatGPT – Students should be taught several key concepts and principles related to orbital mechanics. Here are some of the learning objectives that students can achieve through the simulation:

- Learning how objects move in space under the influence of gravity.
- A clear understanding of Kepler's Three Laws of Planetary Motion
- Learning about various orbital parameters such as eccentricity, periapsis and apoapsis.
- Relationship between mass of a celestial body and how it affects the shape of the orbit.
- Learning about diverse types of orbits
- The mathematical equations required to calculate orbital parameters.
- Real-world applications of these concepts

2. What specific features are important in a simulation of this kind?

Mr Douglas-Scott – Simple user interface for educational professional. Clear scales for the variables involved. Simulation should be based on real data and be scientifically accurate.

Kai – User friendly UI and a good visual representation of the orbit.

Oli – Usability needs to be considered; tooltips to explain concepts, it must be accurate to real life: it can't be wrong. Add customisation to add some fun to the simulation, for example a different colour of rocket. Make it engaging to the students.

ChatGPT – Here are some key features that you should consider using:

- Accurate physics simulation: accurately models gravitational forces and their impact on the motion of objects in space.
- Interactive controls to manipulate parameters such as mass, velocity, and eccentricity. Users should be able to manipulate these values in real-time to see their results.
- Use visually appealing graphics to represent celestial bodies, orbits, and other relevant elements.
- Display relevant data and information about the selected celestial bodies and their orbits.
- Time control, allowing users to observe the orbit in real-time, slow motion or fast-forward mode. This allows students to visualise the changes in position, speed, and trajectory of objects over time.
- Educational resources: informative content, such as text descriptions, tooltips, or pop-up explanations, which provide clear explanations of the underlying concepts, Kepler's laws, and other principles related to orbital mechanics.
- Guided tutorials: provide tutorials to walk students through the simulation's features and functionalities.
- Assessment tools: create interactive challenges to evaluate students' comprehension and reinforce their learning.
- Platform compatibility: ensure that the simulation can be accessed on various platforms.

3. Are there any existing solutions that you currently know of or use? How could they be improved upon?

Mr Douglas-Scott – PHET simulations that show the orbit of the earth around the sun. These are useful for showing the orbits of the earth and the moon but they don't show any other planets. User interface is very clear and easy to use. The only variables are mass and radius however.

Kai – We used one in physics class, but that one didn't explain what it was doing so show the calculations or formulas it uses.

Oli – I don't really have anything that I use.

ChatGPT – Yes, there are existing simulations. These include:

- Universe Sandbox
- Kerbal Space Program
- NASA's Eyes

While these existing solutions provide valuable resources for learning about orbital mechanics, there are several areas where improvements can be made:

1. Enhanced Accuracy: Improving the accuracy of physics simulations and incorporating more precise calculations can help create a more realistic representation of orbital mechanics. This will provide students with a deeper understanding of the underlying principles.
2. Expanded Parameters and Customization: Offering a wider range of adjustable parameters and customization options would allow students to explore a broader range of orbital scenarios. This could include factors such as gravitational assists, atmospheric drag, and complex multi-body interactions.
3. Improved Visualization and User Interface: Enhancing the visual representation of celestial bodies, orbits, and trajectory paths can improve the overall user experience. Additionally, a user-friendly interface with intuitive controls and clear visual aids can make the simulation more accessible and engaging for students.
4. Integration of Educational Resources: Incorporating comprehensive educational resources directly within the simulation, such as detailed explanations, interactive guides, and links to additional learning materials, can enhance the educational value and provide students with a more comprehensive learning experience.
5. Mobile and Web Accessibility: Expanding the accessibility of the simulation by making it available as a mobile app or web-based tool would enable students to access and explore orbital mechanics from various devices, both inside and outside the classroom.
6. Collaboration and Multiplayer Features: Introducing collaborative and multiplayer features could enable students to work together, share their findings, and collaborate on orbital missions or challenges. This promotes teamwork and provides an interactive learning environment.

4. What level of realism/accuracy is expected?

Mr Douglas-Scott – A certain level of simplification is expected, given how many variables are involved in reality. However, clear relationships, like radius and period, velocity and radius etc. should be accurate as this can be solved from A level physics.

Kai – As accurate as it would be without other unaccounted for variables.

Oli – Pretty real. As a simulation for education, it won't be any good if the information it teaches is wrong.

ChatGPT - The level of realism and accuracy expected in a simulation of orbital mechanics depends on the target audience and the intended educational purpose. Here are a few considerations:

1. **Basic Understanding:** For younger students or those who are new to the subject, a simplified representation of orbital mechanics may be sufficient. The focus should be on conveying the basic concepts and principles of orbits rather than precise calculations. The simulation can provide a visual approximation of orbits and their characteristics without requiring complex physics simulations.
2. **Intermediate Education:** For middle and high school students, a higher level of realism and accuracy is desirable. The simulation should aim to accurately depict the motion of objects in space, considering factors such as gravitational forces, orbital parameters, and basic physics principles. While some simplifications may still be present, the simulation should strive to provide a reasonably realistic representation of orbital behaviour.
3. **Advanced Learning:** For college-level or advanced learners, a higher level of accuracy and realism is expected. The simulation should incorporate more sophisticated physics simulations and calculations, including more complex orbital scenarios and the effects of additional factors like atmospheric drag, gravitational perturbations from other bodies, and more precise numerical integration methods. This level of detail allows students to explore advanced concepts and conduct in-depth analyses of orbital mechanics.

It's important to note that achieving absolute accuracy in a simulation can be challenging due to the complexity of orbital mechanics and the trade-offs between computational resources and real-time performance. Striking a balance between accuracy and usability is crucial to ensure that the simulation remains accessible and engaging for the intended audience.

5. Should the simulation allow users to customise parameters such as mass, radius, and velocity of objects?

Mr Douglas-Scott – Yes.

Kai – Yes and maybe others like time period.

Oli – Yeah, with a little orbit indicator that you could use to make it easier to see what can work.

ChatGPT - Allowing users to customize parameters such as mass, radius, and velocity in the simulation of orbital mechanics is highly beneficial. It promotes exploration, experimentation, and hands-on learning, enabling students to observe the effects of parameter changes on orbits. Customization reinforces key concepts, encourages critical thinking and problem-solving skills, and makes the simulation adaptable to different levels of learners. However, it is important to provide appropriate limitations and guidance to maintain realistic and stable orbits.

6. How interactive should the simulation be?

Mr Douglas-Scott – A clear user interface is important as teachers are more likely to make sue of it if it is clear. Interactive would be good as predictions can be made and calculations performed. Ideally usable by teachers students and non specialist teachers if necessary. If not interactive it will be of less use but then data should be displayed very clearly.

Kai – The simulation should have variables that can change and be showing the orbit.

Oli – Yes, what if I want to see what happens when smaller objects interact? Students should be able to change even the smallest aspects of the simulation.

ChatGPT - A high level of interactivity and user control is desired in the simulation. This allows for hands-on exploration, customization, concept reinforcement, active problem solving, and

personalized learning experiences. By providing interactive features, the simulation becomes a powerful tool for engaging students and facilitating a deeper understanding of orbital mechanics.

7. Should the simulation provide other visual cues, such as annotations?

Mr Douglas-Scott – This works best if it is optional. Too much information can overload the visuals. PHET have the option to hide or show data, orbital paths etc. which works very well I think.

Kai – I think it would be helpful to have annotations for example the mass of the planet being to the planet and not the orbiting object.

Oli – Yes.

ChatGPT - Providing visual cues in the form of annotations enhances understanding in the simulation. They clarify explanations, highlight key features, reinforce learning objectives, promote interactive exploration, and contextualize real-world applications. These cues act as valuable educational tools that support students in comprehending and engaging with the principles of orbital mechanics.

8. Is there a need for tutorials/resources that teachers could use in addition to this simulation?

Mr Douglas-Scott – Ideally no - if the user interface is clear it should be self explanatory. Ideally some accompanying worksheets with practice problems would be a great teacher resource.

Kai – No.

Oli – Yes.

ChatGPT - Tutorials and additional resources are valuable for teachers to complement the simulation and support the learning process. They provide guided learning, conceptual explanations, reinforcement of learning objectives, differentiated instruction, assessment tools, and integration with lesson plans. By utilizing these resources, teachers can enhance their teaching strategies, address diverse student needs, and promote a deeper understanding of orbital mechanics concepts.

9. Where should this simulation be available?

Mr Douglas-Scott – Most of these simulations are from big providers so somewhere easy to access.

Kai – The simulation should be available online like a website.

Oli – Web app, downloadable exe, on mobile devices. This is so that it can be accessible to anyone.

ChatGPT - The simulation should be available in various platforms and settings to maximize accessibility and reach a wider audience. Here's a summary of the potential availability options:

1. Classroom Setting: The simulation should be accessible within classrooms, either through desktop computers, interactive whiteboards, or projectors. This allows teachers to incorporate it directly into their lessons, providing students with hands-on learning experiences.

2. Online/Web-Based: Making the simulation available as a web-based application allows students to access it from any device with an internet connection. This flexibility enables learning both inside and outside the classroom, accommodating remote or self-paced learning scenarios.
3. Mobile Devices: Developing a mobile application for iOS and Android devices expands the accessibility of the simulation. Students can then access it on their smartphones or tablets, providing a convenient and portable learning tool.
4. Learning Management Systems (LMS): Integrating the simulation with popular Learning Management Systems used in educational institutions, such as Moodle or Canvas, enables seamless integration into existing online course modules. Students can access the simulation and associated resources directly through their LMS accounts.
5. Open Educational Resources (OER) Platforms: Making the simulation available on open educational resources platforms, such as Khan Academy or OpenStax, increases its visibility and reach. These platforms are widely used by educators and students to access free educational materials.
6. Educational Software Repositories: Uploading the simulation to educational software repositories, such as GitHub or SourceForge, allows educators and developers to discover, download, and customize the simulation for their specific needs.
7. Science Centres and Museums: Partnering with science centres and museums to install the simulation on interactive exhibits or educational kiosks provides a hands-on learning experience for visitors of all ages.

10. Are there any accessibility needs that should be considered?

Mr Douglas-Scott – Think about colours, some colour schemes ie. yellow on red/ vice versa can be hard to read. Consider colourblind perhaps? Consider the level of maths/ physics knowledge this will require and who your intended user is.

Kai – Should be able to run on a computer/ whiteboard for schools.

Oli – Maybe include alternate ways to control the simulation.

ChatGPT - Yes, there are several accessibility needs that should be considered to ensure inclusivity and equal access to the simulation. Here's a summary of the key considerations:

1. Visual Accessibility: The simulation should provide options for adjustable text size, high contrast modes, and compatibility with screen readers. These features benefit students with visual impairments or those who have difficulty reading small text or distinguishing colours.
2. Auditory Accessibility: Providing captions or transcripts for audio content within the simulation helps students with hearing impairments to access and understand the information. Additionally, providing alternative visual cues or descriptions for auditory information ensures that students who are deaf or hard of hearing can fully engage with the simulation.
3. Motor Accessibility: The simulation should have controls that are adaptable to different input methods, such as keyboard navigation or alternative input devices. This accommodates students with motor disabilities who may have difficulty using a standard mouse or touch screen.
4. Multilingual Support: Offering multilingual options or translations within the simulation allows students whose first language may not be English to access the content and understand the instructions and explanations effectively.

5. User Interface Design: The user interface should be intuitive, with clear navigation and easily identifiable controls. This benefits students with cognitive or learning disabilities who may require a simpler and more streamlined interface.
6. Compatibility with Assistive Technologies: Ensuring compatibility with commonly used assistive technologies, such as screen readers or speech recognition software, is essential for students who rely on these tools for accessibility.
7. Alternative Formats: Providing alternative formats, such as downloadable PDFs or accessible HTML versions, allows students to access the simulation through different means and platforms that are compatible with their assistive technologies.

At this point, the usefulness of ChatGPT in providing feedback is diminished due to the difficulty of it correctly understanding my code. Therefore, I will not be consulting it for feedback on my software.

Research

²³One major existing solution is the video game '**Kerbal Space Program**' – gameplay involves building a rocket and flying it in a fully physics-based 3D solar system.



Constructing a rocket in-game.

The 'map view', where orbits are represented by different colours of lines.

There are two main reasons why this game is not suitable for the purpose of demonstrating orbital mechanics to students:

- The software is proprietary and may be financially restrictive.
- The game focuses on constructing rockets to fly – while it is a physics-based world, the focus on gameplay means that it may take time for gameplay to progress until an orbit can be achieved.

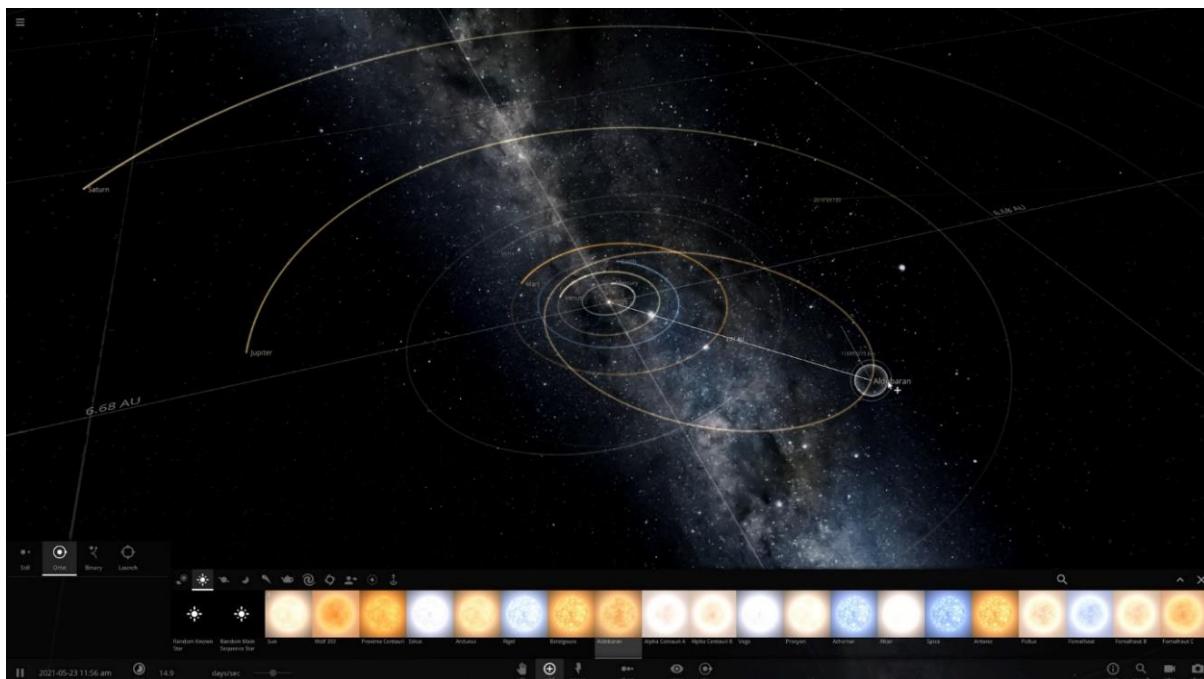
Features to carry forward	Justification
Orbit lines	Allows for a more complete visualisation of the path an object will take, easier to tell whether an object is in orbit
Ability to focus the camera on specific objects	Allows users to more easily observe an object of interest

² Image 1: (Larkin, 2018)

³ Image 2: (Given, 2017)

Tracker for change in velocity required	Spacecraft use many different manoeuvres in order to save fuel; to demonstrate how much fuel would be used on a particular manoeuvre to compare to others, a counter would be useful.
Features not to carry forward	
Spacecraft construction	Unnecessary to demonstrate the concept
Limited amount of fuel	Unnecessary, and would make it more a game instead of a simulation. Calculations to do with mass of spacecraft with loss of fuel would be required and may impact performance.

⁴Another solution is '**Universe Sandbox 2**', an accurate physical simulation of the entire universe based on real-life observations and equations. In this software, users can place objects and change values such as masses, surface temperatures and velocities. The user can also speed up time to observe how the universe changes.

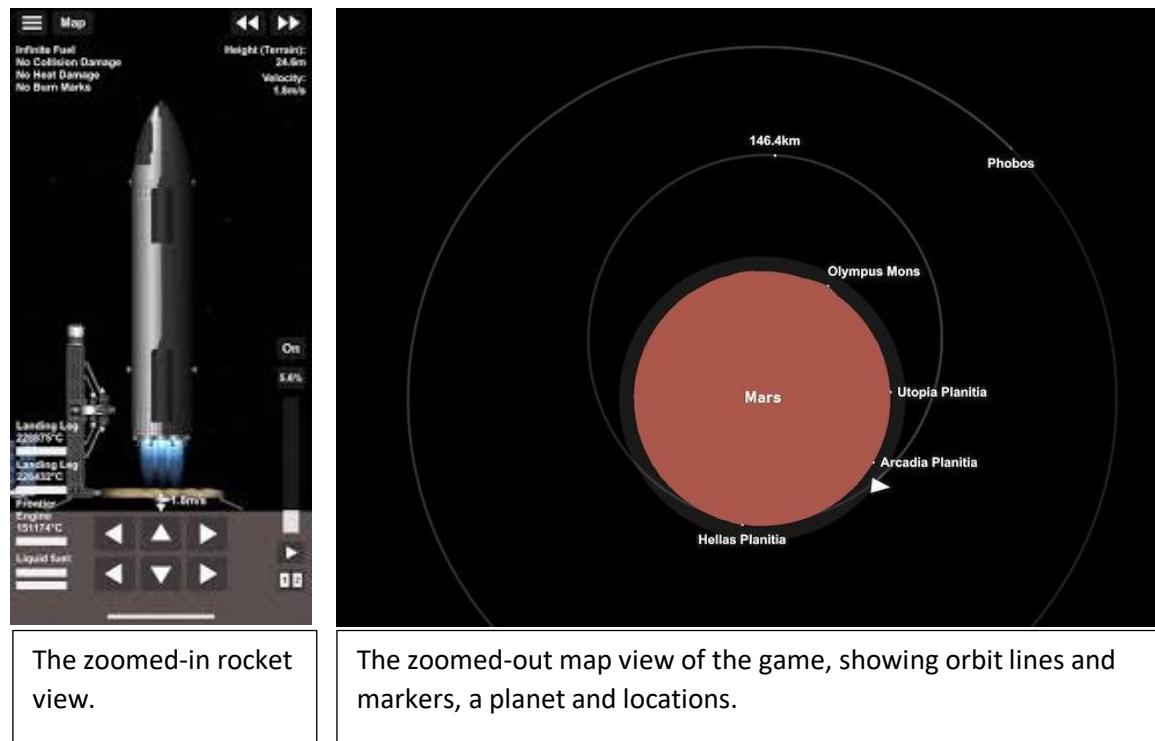


This software, however, is not suitable for use in the classroom again for financial reasons as it is proprietary. It may also have a negative performance impact on older devices but has low system requirements.

⁴ Image 3: (Cesare, 2018)

Features to carry forward	Justification
Realistic physics calculations	In an educational simulation, having inaccurate and unrealistic results is unacceptable as it may harm the understanding of students.
Ability to place/delete celestial bodies	Allows for a multitude of situations to be created to demonstrate these concepts
Ability to change the parameters of celestial bodies, i.e., mass	Again, allows for multiple situations to be created to demonstrate these concepts
Basic system requirements	Allows for the program to be run on almost all devices
Features not to carry forward	
3D view	Too complex to implement in the limited time I have, and unnecessary
Ability to visit any star system in the universe via procedurally generated systems	Outside the scope of the project

⁵⁶A final example of an existing solution is '**Spacecraft Simulator**' – this is a mobile game in which players build rockets and launch them into a simulated solar system, like Kerbal Space Program, but instead in a 2D plane.



Features to carry forward	Justification
Orbit lines and apoapsis/periapsis markers	Allows students to see the orbits of objects easily

⁵ Image 4: (spaceflightsim1157, 2022)

⁶ Image 5: (Morojna, 2022)

Top-down 2D view	While it might not be able to demonstrate the concept in a more detailed way, this approach allows for the use of simple controls, such as an interactive whiteboard.
Touch controls for spacecraft	Provides an accessible method to control the motion of a spacecraft object without the use of menus or key bindings.
Features not to carry forward	
Spacecraft construction	Unnecessary to demonstrate the concept
Spacecraft destruction	Unnecessary to demonstrate the concept

As part of this solution, I would like to make it so that while the colour of planets can be determined by the user, the colour of stars correlates to their surface temperature as happens in reality. I researched solutions involving using the graphs of star colour against surface temperature, trying to find an algorithm that could convert a given temperature to an RGB value corresponding to the colour of light that would be given off. The difficulty with this is that the scale of temperature against each RGB value is not linear and is different for all three colours.

I came across a solution for a procedure on Stack Overflow that utilised the Numpy library with three predefined curves to look up a value for each when a temperature was passed through. However, as I am not currently using the Numpy library as part of my program, I am reluctant to add it to the list of software requirements. Another solution I considered was just manually programming an RGB value for certain ranges of temperatures, which has the advantage of not relying on another library. The issue with this solution is that the transition between different colours would not be smooth; it would jump in between colours when a threshold is passed.

In the end, due to both time and complexity limitations, I decided to use the solution from stack overflow with Numpy. This was because it would be both easier to implement and also a more realistic solution that would provide a better user experience. I assumed that should the user have the technical knowledge required to install the Pygame library to use this (in the event that I am not able to implement this into a website for easier access), it would not be difficult for them to install the commonly used Numpy library as well. The code I have used will be commented as such, and below links to the answer on stack overflow. The code I will use is part of the second answer to this⁷ query:

<https://stackoverflow.com/questions/21977786/star-b-v-color-index-to-apparent-rgb-color>

This page was accessed 04/01/2024 at 22:33.

Another thing that I would like to achieve is to have this program accessible from an online source. I researched three main methods to achieve this:

- Django – web framework that can run Python code, but may not be suitable for pygame

⁷ (DocLeonard, 2019)

- PythonAnywhere – made by Anaconda, this site allows users to host and run Python code in the cloud easily and would be the perfect solution. However, it does not support the use of Pygame.
- Pygbag – A library for running Pygame code in a web browser, which can then be uploaded to itch.io.

Depending on the nature of my code, in a later development stage I will have the final decision on which solution to use (if I do decide to put this online).

Saving/loading of scenarios

Unlike the software outlined in my research, my program will need to be much less intensive on computer resources to be run on as many devices as possible. Consequently, to be able to demonstrate any scenario in this area, multiple different scenarios will have to be created. To make this quick and easy to do, I have decided to use a system of saving and loading different scenarios.

My initial idea for saving and loading scenarios from the program was to store the data of each object in the simulated scene to a line in a .txt file. Each line would contain the properties of each object, such as mass/speed/direction of travel etc., and to load the scenario the program would simply have to read the selected file one line at a time. I was unsure of the method I was going to use to select the file to load from, but after some online research I narrowed it down to these methods:

- User would input the name of a file they saved, which they would have to remember from memory (not ideal but simplest implementation)
- Same as above, but a list of the names of the available files is displayed on-screen (second easiest to implement but would require the ability to read the name of each text file in the program's root folder)
- Opening a file dialogue for the user to select their file (The most challenging to implement and would require the use of the Tkinter library from what I've seen, may only work on devices with supported operating systems)

Of course, these methods would only be available to me if the program was being run on the user's device; if it was being run via a website, it would be difficult to download files when saving scenarios and to get the user to upload files. After doing research on this, I found that in this case it would be outside the scope of what is possible in my limited timeframe.

Success Criteria

Requirement # and description	Desired feature or extra (time limitation)?	Justification	Success Criteria
1	Options menu	Desired	A menu is needed to be able to choose basic options and manipulate the simulation. An easy to understand and navigable UI.

1A	Options to manipulate mass of objects	Desired	Mass is an important factor in equations for gravity. Being able to change it can demonstrate this fact.	Ability to change the mass of objects in-game, and the effects of the change reflecting on the physics.
1B	Options to manipulate speed of objects	Desired	Speed is another important factor in determining the orbit of an object. To change the orbit, the user must be able to change the object's speed.	Ability to change the speed of objects in-game, and the effects of the change reflecting on the physics.
1C	Options to manipulate direction of travel of objects	Desired	Again, allows for scenarios to be changed to demonstrate different concepts.	Ability to change the direction of travel of objects in-game, and the effects of the change reflecting on the physics.
1D	Options to manipulate time period of objects	Extra	While slightly more difficult to implement as it affects/is affected by multiple other factors such as orbital radius and speed, should still be able to be changed.	Ability to change the time period of objects in-game, and the effects of the change reflecting on the physics.
1E	Option to control simulation speed	Extra	Can be used to show how an orbit with a long time period would play out, in a short amount of time.	Speed of every object in the simulation can be sped up or stopped while keeping physics consistent.
1F	Options to show labels above objects to display information about them	Desired	The program must be able to give information about objects so that the effects of changing certain factors can be more easily seen.	Labels on objects that give the user the basic information of it when selected.
2	Controllable spacecraft/object	Desired	The user needs to be able to manipulate an object to be able to travel throughout the simulation and experience the effects of gravitational force.	Object that the player can directly control via a separate menu.
3	Orbit lines	Desired	Allows user to see where an object is going to go, and whether it is in orbit.	Accurate lines that predict a reasonable distance ahead what the trajectory of an object will be.
3A	Equipotential lines	Extra	Allows the user to see visually what the effects of gravity will be across the simulation.	Accurate equipotential lines between objects that can be toggled on or off.

3B	Apoapsis and periapsis markers	Extra	May become difficult to calculate but will provide a good visual indication of altitude.	Accurate apoapsis/periapsis markers displayed at correct points on an orbit line.
4	Realistic physics	Desired	The entire point of the simulation. Without realistic physics, this cannot be used as an educational tool and so this must be done correctly.	Objects in the game behave as in real life, calculations used to determine forces must be accurate.
4A	Atmospheric height and effects	Extra	Would only affect the spacecraft but could be useful in demonstrating concepts such as aerobraking.	An atmosphere around planets is present, that slows down the spacecraft when passing through.
4B	Planets exert gravitational force on each other	Desired	Results in a more accurate program.	Planets' gravitational force on each other is considered when determining movement.
4C	Collision physics	Extra	Obviously if two planets collide then they should produce a destructive effect, but this is not the most important feature of the simulation.	When two objects collide, they react instead of passing through one another.
4D	Keep selected parameters from changing as the simulation runs	Desired	To better see the effects of changing one variable on another without other variables getting in the way, certain parameters should be able to be 'locked' in order that they do not affect other factors.	Ability to 'lock' variables of choice so that they do not change as the simulation runs.
5	Multiple planets/stars	Desired	Allows the user to experience gravitational forces from multiple bodies.	Multiple planets and stars can be represented in the software at the same time.
5A	Preset scenarios	Desired	Can be used to quickly show a concept without having to set it up manually.	At least 3 preset scenarios are able to be loaded at some point in the program.
5B	Create + destroy planets	Desired	Allows the user to customise the simulation by creating multiple planets to demonstrate any scenario.	Users can place at least two celestial bodies at variable locations. Users can destroy objects.

5C	Save + load custom scenarios	Extra	Allows for custom scenarios to be easily reused.	Options to save and load at least 3 custom scenarios that have been created in the program.
6	Simplistic graphics	Desired	As it is an educational program, the graphics need not be too complex. However, to keep the program engaging to students some level of graphics is required.	Graphics are simple enough to not be distracting, but detailed enough that at a glance it can be understood what each object represents.
6A	Stars as background	Extra	Should be easy enough to implement but I need to make sure it does not interfere with visibility of orbit lines.	Small dots are present in the background to represent stars and more effectively show that it is outer space.
6B	Textures for planets	Extra	Not exactly necessary as they could just be coloured spheres, but again will make the simulation more engaging	Planets have at least one texture other than solid colour.
7	Accessibility needs	Desired	Program needs to be able to be accessed by everyone in a learning environment in order not to disadvantage any students.	Program is able to be easily navigated through simple (point and click) means. Program can be easily accessed and run.
7A	Navigable using just the mouse	Desired	Allows for the simulation to be controlled entirely using an interactive whiteboard, making demonstrations in lessons more engaging.	All features of the program are accessible via clicking or holding the mouse buttons.
7B	Able to be run on low-end computers	Desired	Makes the program available to be run in more classrooms.	May be able to be satisfied by requirement 7C. Program performs well (>20fps) on devices at a low (~2GB RAM) spec.
7C	Can be accessed online	Extra	Makes the program significantly more accessible, especially by less tech-literate users, as it doesn't have to be downloaded and run.	Program can be accessed by any computer with an internet connection.
7D	Desktop shortcut to run program	Extra	If the program won't be run on a web app, then a desktop shortcut is invaluable to easily run the program. Would be made redundant by requirement 7C.	Program can be run simply by clicking on a shortcut icon on desktop.

7E	Program window scales to any device resolution	Desired	Again, allows many more devices to use the software.	Program correctly sizes to all monitor sizes.
8	Camera movement	Desired	Allows for users to pan around to different parts of a solar system if necessary. Camera may just be able to focus on different objects rather than free movement due to mouse control constraint.	Camera is able to be moved in cardinal directions while physics remains consistent.
8A	Camera zoom	Extra	I have not worked with camera zoom in a 2D context before, but it would be valuable in further focusing on different objects.	Camera can be zoomed in and out, with physics remaining consistent.
8B	Camera focus on specific objects	Extra	Allows for users to focus on and follow an object of interest.	Camera tracks and follows an object in motion, keeping it in the centre of the screen.

Hardware/software requirements

Software requirements (and Python libraries):

- Python 3.11.0 (or above)
- Pygame 2.5.2 (or above)
- Numpy 1.26.3 (or above)

Minimum system requirements (this was the lowest spec device I had access to):

- Intel Core i3-8130U (2.2GHz)
- 8Gb RAM
- Intel UHD Graphics 620

However, if I do manage to set this simulation up to be able to be accessed online, all of the above requirements are invalid; the only requirement would be that the device has access to the internet.

Limitations

Feature	Reason	Justification
Creation of custom spacecraft	Time/outside scope	Would be unnecessary as the project aims only to demonstrate orbital mechanics.
Use of 3D graphics	Time/complexity	3D rendering would be taxing on devices if the program is run locally, decreasing the accessibility of the program.
Music and sound design	Outside scope	Would be difficult to find sounds that would not have any licensing issues; as this is an educational simulation, they are also unnecessary.

Factors such as atmospheric composition of planets	Outside scope	Completely unnecessary and would add no education value to the simulation. Would slow down processing times.
Lighting effects from stars	Complexity/time	Would be computer-intensive to calculate in real time and unnecessary to demonstrate the required concepts.
Availability on mobile devices	Outside scope/complexity	I have no experience with porting software to mobile platforms, and as this is intended to be used in a classroom setting, I do not think it would be necessary. It would however be interesting if the simulation is available on a website, as due to the mouse control requirement, there should be no reason the program should not be accessible on mobile devices. Nevertheless, I will not be specifically testing for this.

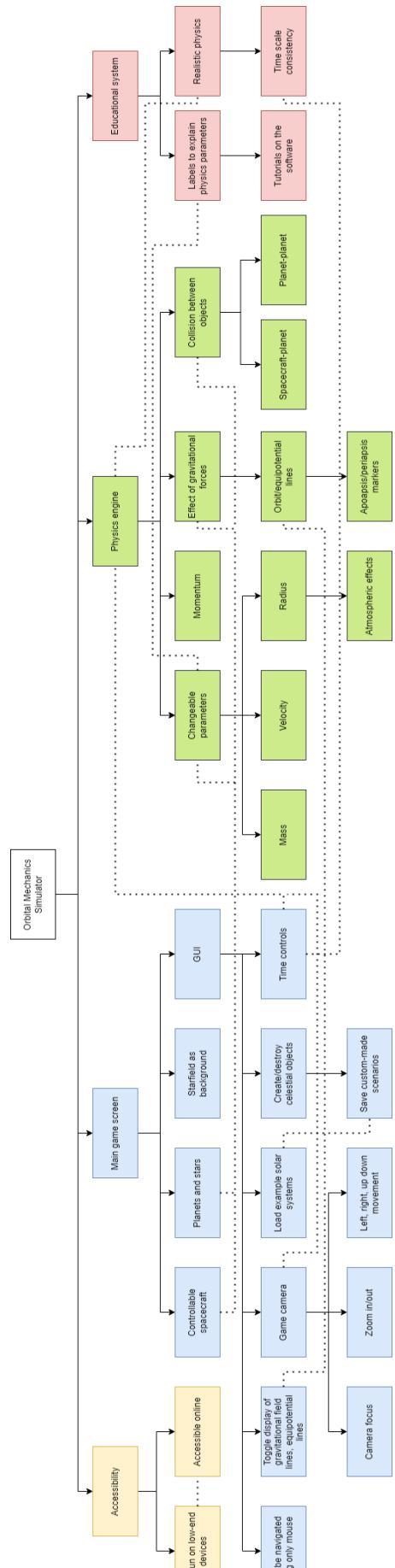
After discussing all of the above analysis and success criteria with my stakeholders, they were happy for me to continue to the design phase with these features.

Design

At program start, the user should be met with an example solar system and the options menu. I have decided to not create a main menu, as it would be unnecessary; the options that would have been present on it (such as the save and load feature) will be able to be accessed through another method in the simulation itself.

In terms of stars, I am planning on them being created as planet objects as they should behave in the same way, only including a temperature variable that is otherwise out of scope for the planets. Stars will emit no light and have no other effects, but there may be some calculations required later on for atmospheric height if I am to implement requirement 4A.

The starfield background (6A) will most likely be a set of white dots randomly placed at the start of the simulation so that their movement with the camera is consistent.



Hierarchy chart

This is the first iteration of the program I wish to develop. Optional/extraneous features are also displayed on this diagram.

The solution is broken up into 4 main sections: The menu and controls, the educational aspect of the system, the physics engine and accessibility.

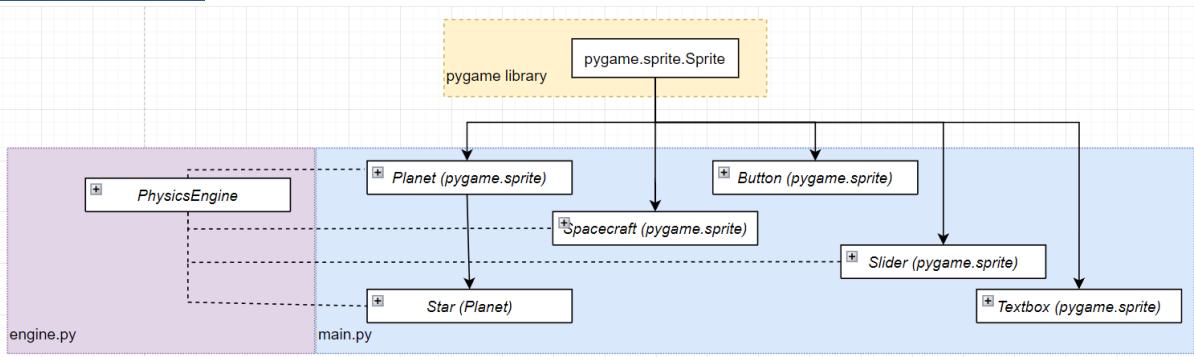
The menu and controls are an integral part of any software and have been decomposed as such. A build menu will be used to enable users to place new objects, and scenarios can be saved and loaded from files. Toggles for enabling and disabling certain aspects are also available.

The educational aspect of the system relates much to the physics engine; this is a tool to aid learning and is intended to be used in a classroom setting where a teacher would be able to explain concepts utilising my tool more effectively than I could through words in the software. Labels will be used to display parameters, and the ability to lock variables allows users to see how specific variables are affected by certain changes.

As the main focus of the software, the physics engine will loop over and read all the values of the objects that have physics applied to them and calculate and change variables accordingly each tick. When initial conditions need to be set up, such as atmospheric height, the engine will also be used to calculate these.

The program must also be accessible so that as many people as possible have access to this resource.

Class Diagrams



The main structure of the program.

Planet (pygame.sprite)
pygame.Vector2(): screenPosition
pygame.Vector2(): gamePosition
pygame.Vector2(): velocity
int: mass
float: direction
int: radius
float: scalingFactor
float: surfaceGrav
calcGravAtSurface(self.mass, self.radius)
applyForce(self)
onDelete(self, physObjects)
update(self)
<u>__init__</u> (self, position, velocity, mass, direction, radius)

The **planet object** includes vector2s for position on the screen, position in the game relative to all other objects in the scenario, and velocity. Floats are used to show direction, the surface gravity and the scaling factor and integers for mass and radius.

I also included a scaling factor variable for implementation of the camera zoom system, where the size of the planets on screen will need to be changed, as well as the way the engine calculates distance between the planets themselves.

The **surfaceGrav** variable corresponds to the output of **calcGravAtSurface()**, which is a purely informational value that instructs users of what the gravity affecting a person standing on the surface of the planet would experience. I decided to add this as it would only be calculated once and so would not be resource intensive.

applyForce() is intended to be called by the physics engine to move the position of the planet within the object itself, by changing the **velocity** variable.

onDelete() should remove the object from the **physObjects** group and then stop the draw function, before destroying the object itself. This would be called in the build function when the planet is removed from a scenario.

update() draws the object onto the screen and moves the sprite by adding the velocity to the position of it. For scaling with the camera zoom, I will copy the scaling factor of the physics engine class here.

In the **__init__()** method, I will set all of the internal variables to the values for each passed as parameters. I will also set up the surface for the planet to be drawn onto on the screen.

Star (Planet)
int: temperature
tuple: colour
getColour(self, self.temperature)
__init__(self, temperature, position, velocity, mass, direction, radius)

The **star object** is a subclass of the planet object. Its purpose is to provide a way for the physics engine to differentiate between planets and stars, as stars' luminosity needs to be calculated within the physics engine for certain calculations (atmospheric height); I can add these objects to their own group to pass into the physics engine. While the colour of planets can be chosen, the colour of stars will be given based on its temperature.

While temperature on a planet has little significance in this program, the temperature of a star is integral to its effects on the atmospheres of other planets, and its own colour. For these reasons, and the otherwise similarity to the planet class in how other information is stored and the object is drawn, I have chosen to make this a subclass of the planet object.

I considered adding a function to calculate the luminosity of the star and store it as a variable in the object. However, due to the only current use for this value being the atmospheric height of orbiting planets, it will rarely be used and as such when this value is needed it will instead be calculated directly by the physics engine.

`getColour()` is called every time a change is made to the temperature of the star. It references the temperature parameter against a list of RGB values for a range of temperatures, before applying it to the colour of the drawn object.

`__init__()` calls the `getColour()` method initially, and then initialises the parent class with the returned RGB value as a parameter to set the initial colour.

Spacecraft (pygame.sprite)
pygame.Vector2(): screenPosition
pygame.Vector2(): gamePosition
pygame.Vector2: velocity
int: mass
float: direction
float: scalingFactor
applyForce(self)
update(self)
turnLeft()
turnRight()
addVelocity()
removeVelocity()
__init__(self, position, velocity, mass, direction)

The **spacecraft object** contains more vector2s for position and velocity, an integer representing mass, and floats for direction and the scaling factor.

`applyForce()` and `update()` work in the same way as in the planet class, but they affect the spacecraft movement instead.

`turnLeft()` and `turnRight()` are intended to be called via either the Pygame events system or through the use of the spacecraft controls menu, to rotate the spacecraft. I use two separate methods here but in the code I will try to find a solution in which I could just pass a variable denoting the direction to turn in to make it a single method.

I would do a similar thing with `addVelocity()` and `removeVelocity()`, which both affect the velocity of the spacecraft.

In this object's `__init__()` method, I set the initial variables as before and set up the surface – it will be drawn and positioned in the exact same way as the planets class.

PhysicsEngine
float: timeScale
float: scalingFactor
calcGravOnObjects(spriteA, spriteB)
update(physObjects)
updateCamera(physObjects)
calcEquipotential(physObjects)
predictOrbit(physObjects)
getApoapsisAndPeriapsis(physObjects)
cameraFocus(sprite)
getAtmosphereHeight(physObjects)
collision(spriteA, spriteB)
__init__(self, physObjects)

The **physics engine object** is contained in a separate file for easier access.

`update()` is the method through which all calculations for gravitational forces between objects is done; each object is compared with all others in the scenario and the force of their attraction is calculated from their masses and the distance between them through `calcGravOnObjects()`. To change the speed of the entire simulation, I only need to change the frequency of this method being called in the main file.

In order to move the camera, I use the `updateCamera()` method to apply an offset force and move all objects in the same direction simultaneously to create the effect of the screen moving.

`calcEquipotential()` will need to be the most efficient I can make it as I think it will be the most intensive part of this program. It is intended to be called every tick

provided that the equipotential setting is on. It will check every part of the screen against a predetermined list of forces and draw lines between all of these points of equal force.

`predictOrbit()` will be used to calculate the orbit lines for objects, so that users are able to see how they will travel. This is another intensive calculation that must be computed every tick if the setting is on.

`getApoapsisAndPeriapsis()` is called when an orbit is calculated to find the apoapsis (furthest point) and periapsis (closest point) in an orbit around an object.

`cameraFocus()` will move the camera to track a certain object and behaves in much the same way as the offset for camera movement in the `update()` method, however this time the offset will be the movement of the object that is being focused on.

`getAtmosphereHeight()` is the method used to calculate the height of the atmosphere of an object using the mass of any nearby stars and the distance between a planet and that star. In the case of multiple stars being present, it will loop over all of them and sum the calculated heights to find the height of the atmosphere on the planet. In the case of there being no stars in the scenario, this method will be passed over.

In the `collision()` method, intended to be called when two objects intersect, I will calculate the change in momentum for both sprites passed as parameters. The actual check for collision will happen in the main program.

Finally, in the `__init__()` method, I will call all of the orbit prediction and apoapsis methods, as well as initially focusing the camera on the spacecraft. This method is intended to handle the initial processing at startup and will be called on the loading of a scenario.

Button (pygame.sprite)
bool: isClicked
pygame.image: imageClicked
pygame.image: imageUnclicked
update(self)
__init__(self, designation)
effect(self)

The **button object** is used to create the main menu buttons at the base of the screen. They each store a Boolean value to denote whether or not they are in the clicked state, and also store two Pygame images loaded from a file at the start of the program in order to switch and give a visual indication of when they are active.

`update()` is called whenever a mouse click is detected via the Pygame events system in the main running loop. It checks whether the mouse cursor is currently positioned over the button, and if so switches its state and image. If the button is part of a submenu such as the build menu, it checks whether or not the menu is open and shows/hides the image as such.

`__init__()` loads the images of the buttons to the variables corresponding to each at initialisation, in order that the image files in the program folder only have to be loaded in once at program start. The designation parameter is used to position the buttons and set up the collisions for them. It also sets the initial state of the button to unclicked and sets the image accordingly.

`effect()` is called when a button is clicked and uses a list of different actions to complete corresponding to the designation of that button object to create different functionality for different buttons.

Unlike the previous objects, the button does not draw itself to the screen in its update method; instead, when a button is created, it is added to a specific Pygame group for the buttons. It is then drawn using the `Pygame.group.draw()` method in the main running loop. This is also useful for updating all the buttons at once in the check for a mouse press, as the button group can be sent the update command instead of each button individually.

Slider (pygame.sprite)
pygame.Vector2(): screenPosition
update(self)
__init__()

The **slider object** is intended to be used as a visual indicator and control of the camera zoom. It contains another vector2 for position on the screen, the y coordinate for which will be used to determine the level of camera zoom requested.

The `update()` function is called on a mouse press by the Pygame events, and checks if the mouse is within the slider area. This will be represented on-screen by a box behind the actual slider button, which will be created in the same way as the button menu background. If it is within this region, the y value of its position will be changed to be the same as the mouse cursor's y position. The y value of the position of the slider will then be changed into a value between 0 and 1 based on its current position compared to its maximum and minimum possible position, before being passed through into the other relevant objects to be set as the scaling factor.

`__init__()` will be used to set the initial scaling factor in the physics engine, and to position the slider on the screen accordingly.

<code>Textbox (pygame.sprite)</code>
<code>pygame.Vector2(): screenPosition</code>
<code>string: textInput</code>
<code>update(self)</code>
<code>__init__(self)</code>
<code>recordText(pygame_events)</code>

The **textbox object** is used by the user to input information to change certain parameters of objects; multiple instances will be present on the side information panel.

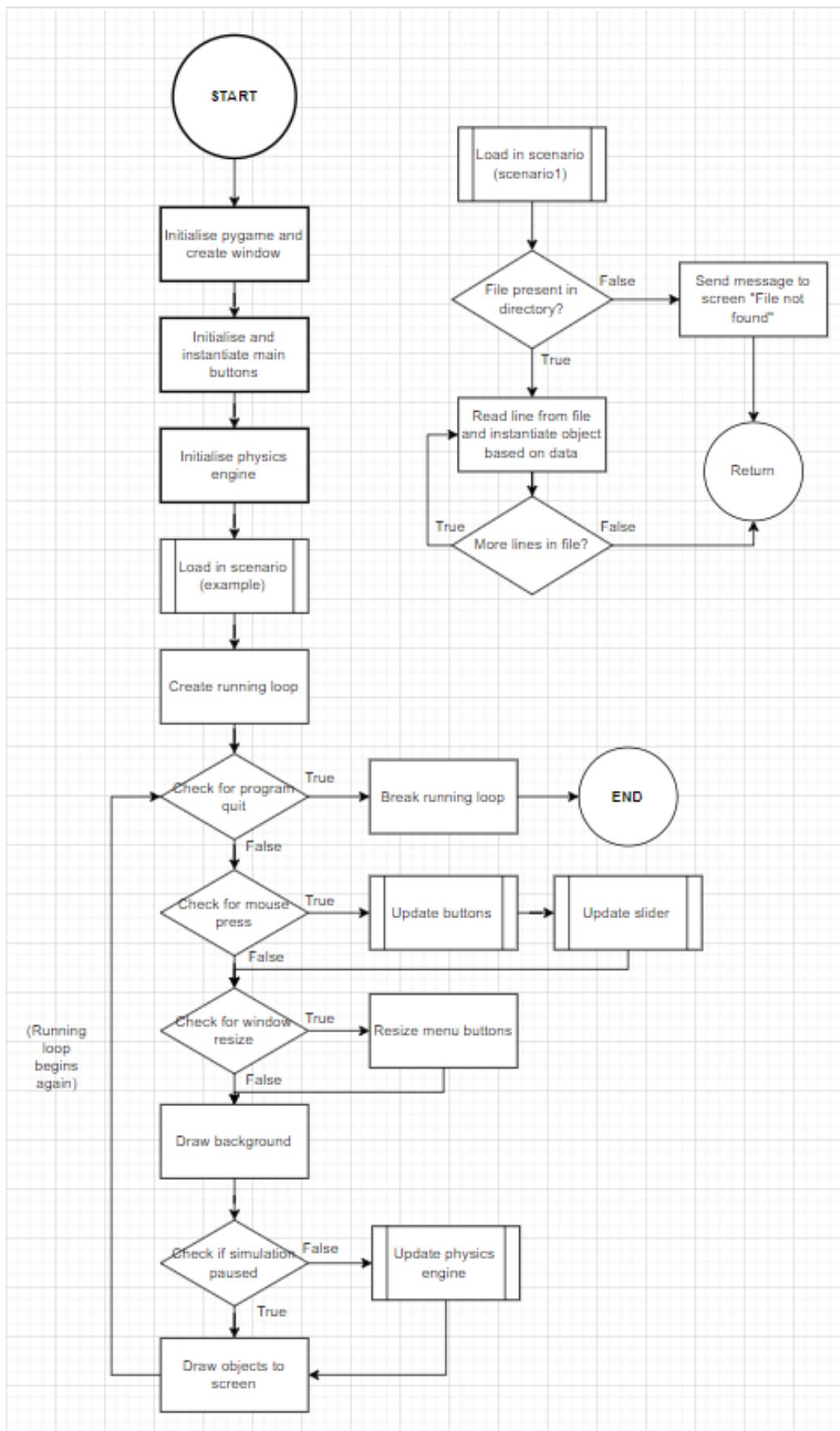
When clicked, the `recordText()` method will be called, which will record the key presses of the user and display them on the text box. This will keep happening until the enter key is pressed or another position on the screen has been clicked, at which point the `textInput` string is returned with the contents of the text box.

The `update()` method checks if the box has been clicked or is currently active, and calls the `recordText()` method if so.

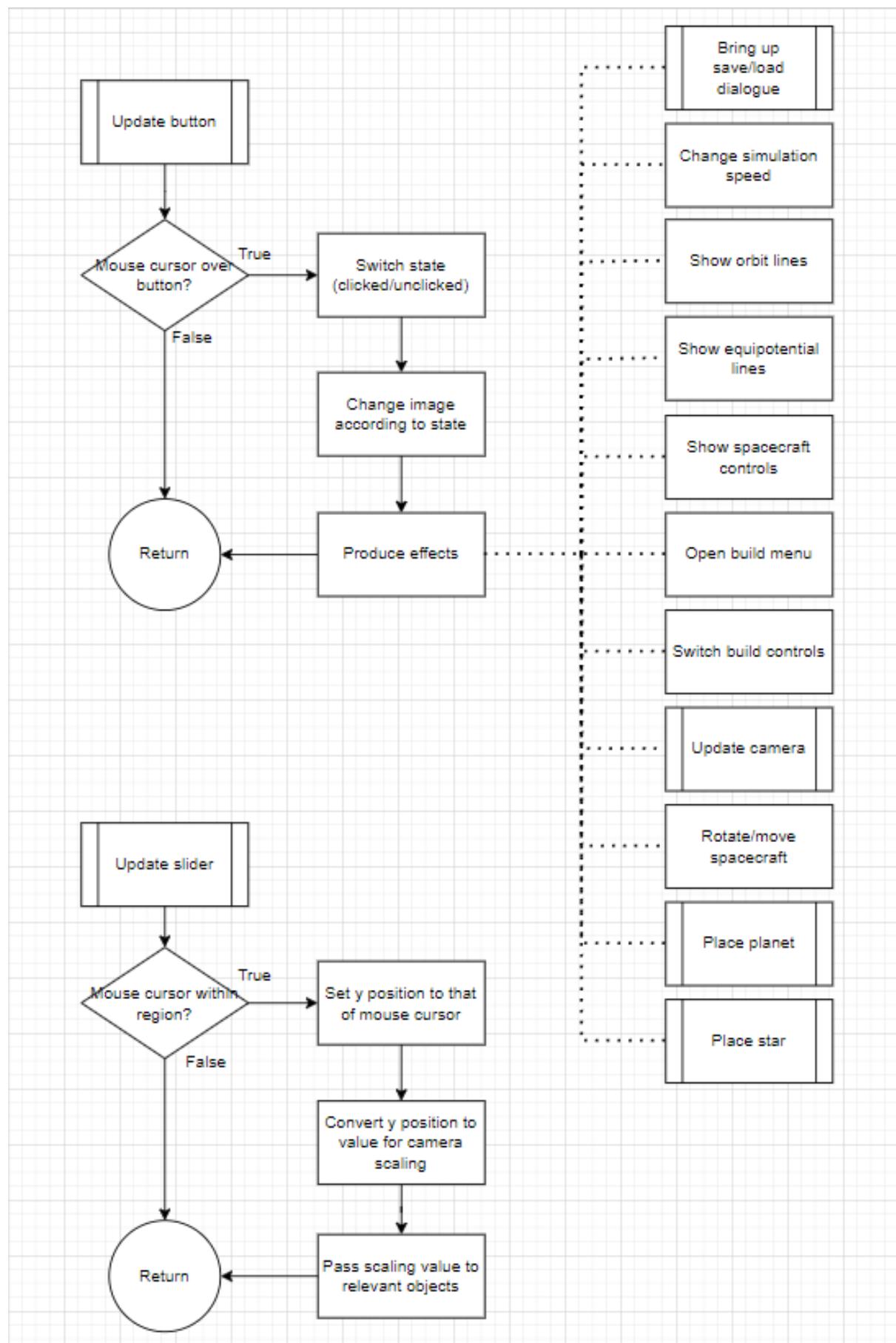
`__init__(self)` will position the button on the screen at game start and also sets up the collisions for where the box is clicked.

Algorithms/logic

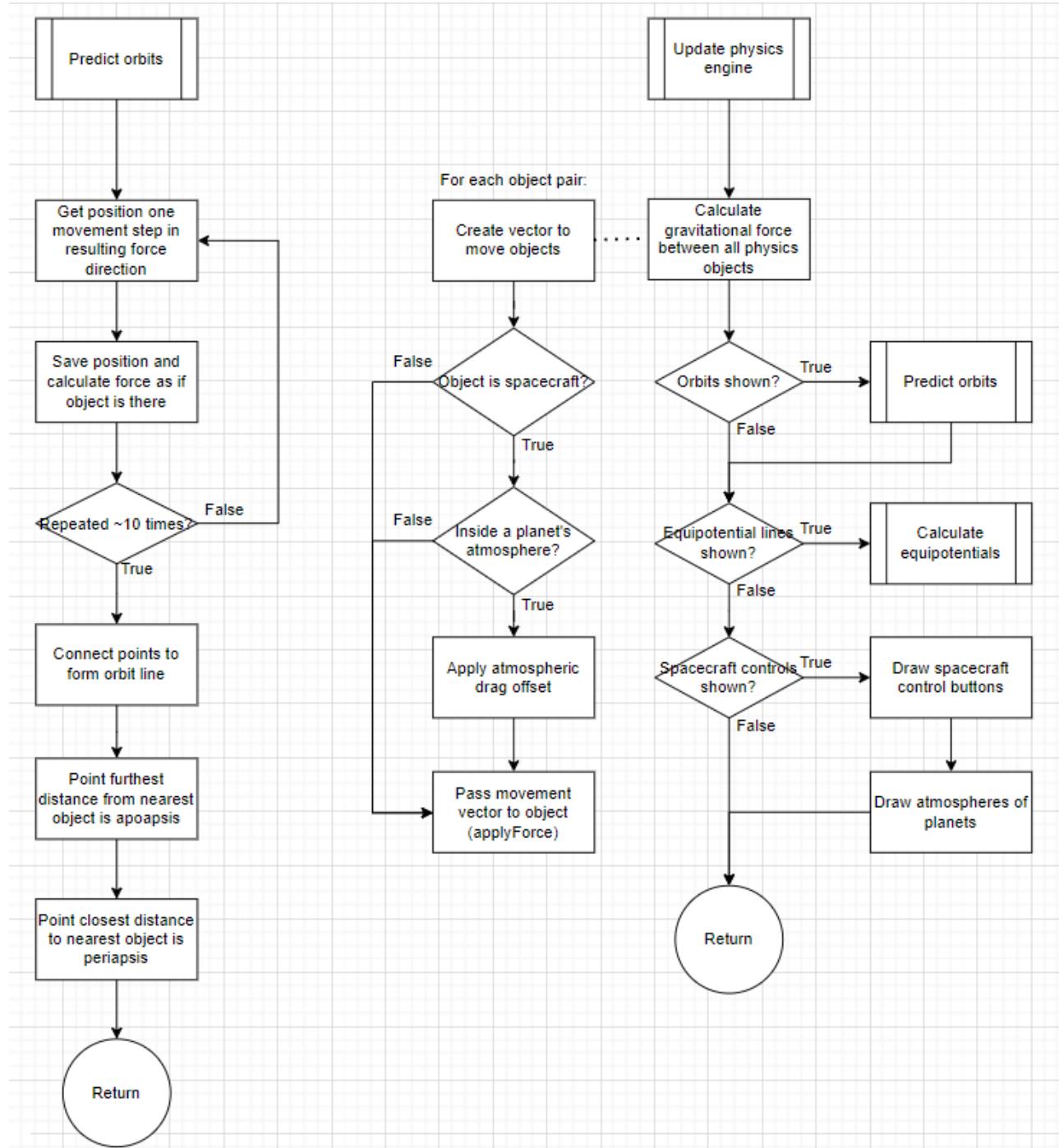
The main program start and running loop:



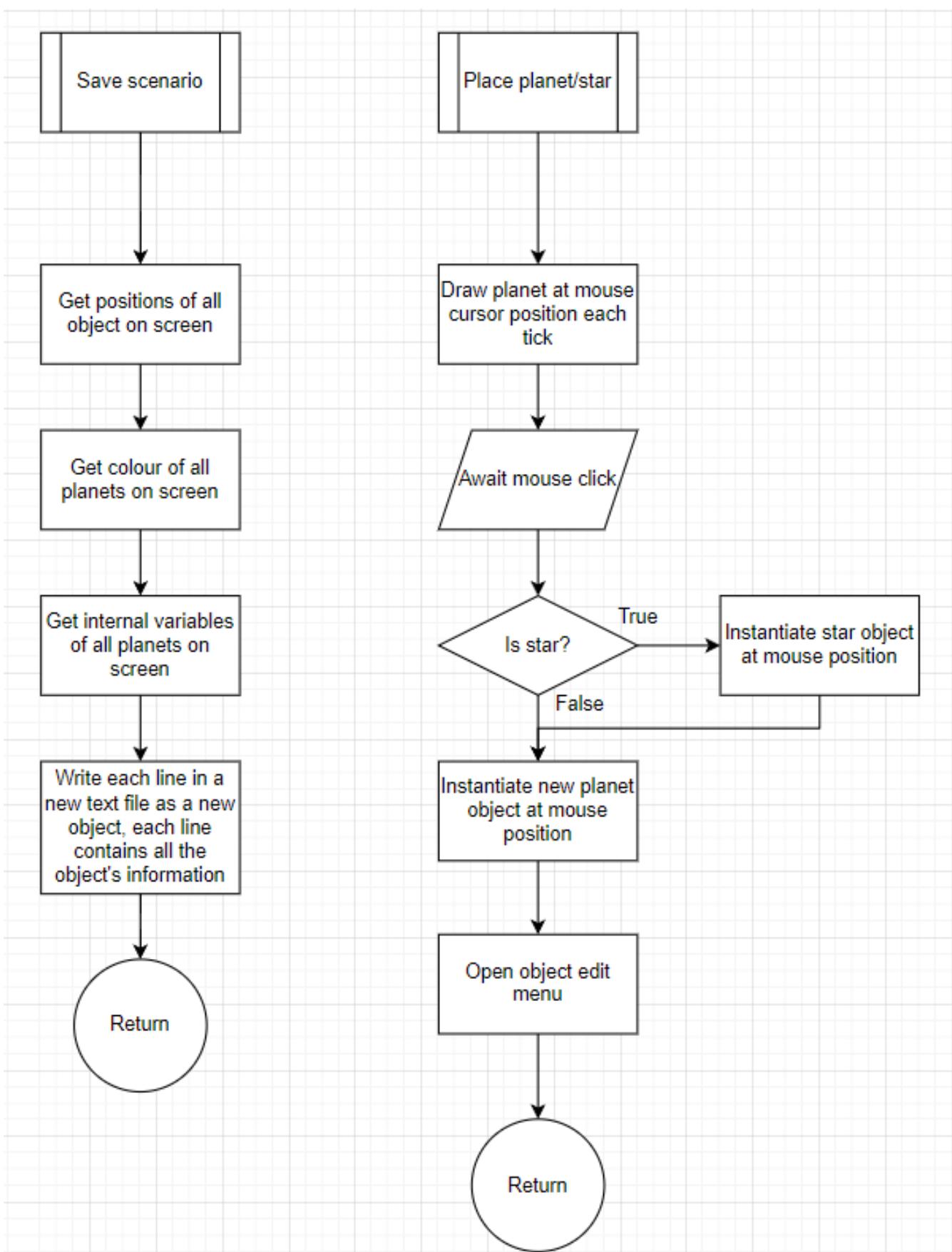
Update methods of buttons and camera slider:



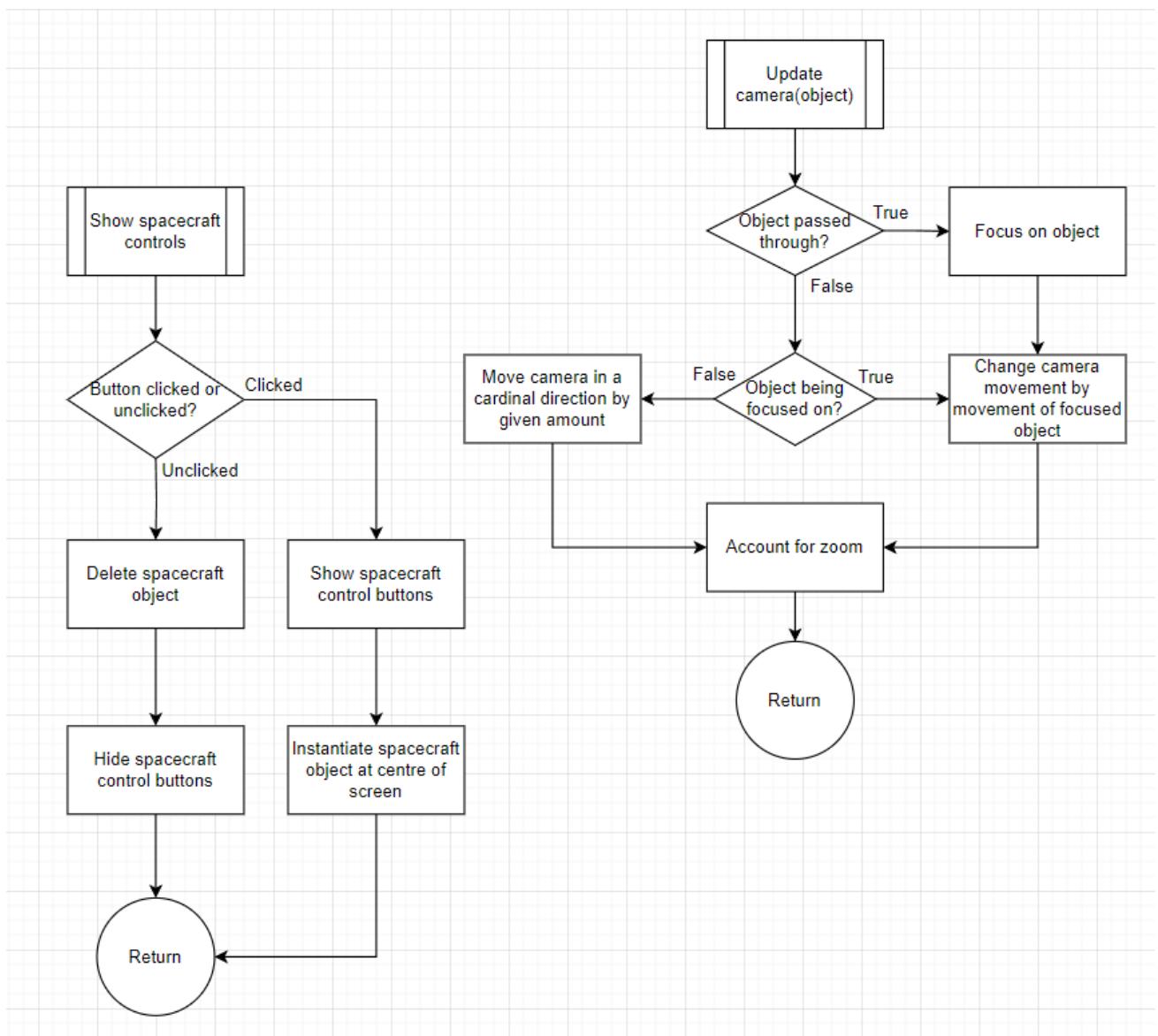
Physics engine and orbit prediction:



Save scenario, place planet or star:



Show spacecraft controls, update camera:



Data structures

main.py:

Variable name	Type	Description/justification
running	Boolean	Used to create a loop to run the Pygame tick cycle and other continuous processes.
screen	Pygame.Surface	Creates a surface that sprites are ‘drawn’ onto.
clock	Pygame.Clock	Keeps track of tickspeed for the game (ensures consistency in physics if time is lost due to low processing power).
buttonGroup	Pygame.Group	Used to store the objects that need to be updated when a mouse press is detected.
physObjects	Pygame.Group	Used to store objects that must be updated by the physics engine every game tick.
physEng	engine.PhysicsEngine	Allows the physics engine to be called and accessed from the main file.
selectedObject	Pygame.Sprite	Holds the sprite whose values will be drawn in a label to the side.
event	Pygame.Event	Used as part of the Pygame events system in order to catch key presses.
redco	Numpy.poly1d	Used to store the curve for red RGB component value against temperature. Part of star colour code.
greenco	Numpy.poly1d	Used to store the curve for green RGB component value against temperature. Part of star colour code.
blueco	Numpy.poly1d	Used to store the curve for blue RGB component value against temperature. Part of star colour code.

main.Planet:

Variable name	Type	Description/justification
colour	Tuple	Used to store an RGB value of the colour of the planet.
radius	Float	Used to store the radius and draw the corresponding circle onto the screen.
mass	Float	Used as part of mass calculations in the physics engine.
gamePos	Pygame.Vector2	Used to store the position of the object relative to the simulated world.
screenPos	Pygame.Vector2	Used to store the position of the object as drawn on the screen.
velocity	Pygame.Vector2	Used to store the current velocity of the object; this each tick increments its game position.
speed	Integer	Used to store the magnitude of the velocity of the object. Changing this variable in the simulation should result in the velocity changing by a proportional amount.
image	Pygame.Surface	Used to get the rect of the object.
rect	Pygame.rect	Used to detect collisions between objects.
correctedMass	Float	Used in calculations to ensure realism, while allowing for easy entering of data by users.
correctedRadius	Float	As above.

main.Planet.Star:

Variable name	Type	Description/justification
colour	Tuple	Used to store an RGB value of the colour of the star, created through <code>getColour()</code> method.
temperature	Integer	Passed as a parameter unique to the Star subclass, is fed as a parameter into the <code>getColour()</code> method to get an accurate colour.
getColour.red	Integer	Used to get the red part of the colour from the temperature against the red Numpy curve.
getColour.green	Integer	Same as above but green.
getColour.blue	Integer	Same as above but blue.

main.Spacecraft:

Variable name	Type	Description/justification
mass	Float	Used as part of mass calculations in the physics engine.
gamePos	Pygame.Vector2	Used to store the position of the object relative to the simulated world.
screenPos	Pygame.Vector2	Used to store the position of the object as drawn on the screen.
velocity	Pygame.Vector2	Used to store the current velocity of the object; this each tick increments its game position.
speed	Integer	Used to store the magnitude of the velocity of the object. Changing this variable in the simulation should result in the velocity changing by a proportional amount.
image	Pygame.Surface	Used to get the rect of the object.
rect	Pygame.rect	Used to detect collisions between objects.

I do not require a `correctedMass` variable for the spacecraft, as its mass will be on a much smaller scale. The physics engine will just use the mass instead of a `correctedMass` for any calculations.

main.Button:

Variable name	Type	Description/justification
designation	Int	I will need a way to differentiate which button is which inside each object, for use when positioning or triggering certain actions. To do this, I will set a button's designation at program start.
isClicked	Boolean	Holds current state of the button
imageUnclicked	Pygame.image	Hold the image of the button in its unclicked state, so that it can be easily switched.
imageClicked	Pygame.image	Hold the image of the button in its clicked state, so that it can be easily switched.
image	Pygame.image	Holds image of the button to be drawn to the screen.
rect	Pygame.rect	Collision box for mouse cursor, to track when a button needs to change state when mouse click detected.

main.Slider:

Variable name	Type	Description/justification
isClicked	Boolean	Holds current state of the slider, and if it should follow the mouse cursor's Y position.
image	Pygame.image	Holds image to be drawn to the screen.
rect	Pygame.rect	Collision box for mouse cursor.
yPosition	Float	Used to both draw the slider to the screen and to translate into a value for the zoom.

main.Textbox:

Variable name	Type	Description/justification
textInput	String	Holds the text that has been entered by the user while the button is active.
image	Pygame.image	Holds image to be drawn to the screen.
rect	Pygame.rect	Collision box for mouse cursor.
active	Boolean	Denotes if the button is active (is listening for text input)

engine.PhysicsEngine:

Variable name	Type	Description/justification
timeScale	Float	Represents the speed at which the program should make calculations and update them on the screen.
zoomScale	Float	Represents the level of zoom on the simulation that the camera should have.
updateCamera.focusedObject	Pygame.Sprite	Holds the object that is currently being focused on by the camera.
calcGravBetweenObjects.radius	Float	Holds the distance between the two objects passed as parameters for use in calculations.
calcGravBetweenObjects.force	Float	Holds the result of the force calculation between the objects.
calcGravBetweenObjects.vect	Pygame.Vector2	Added to velocity of the planet objects calculated for.

Library-specific structures:

`Pygame.surface` is created as the window that users interface with and that all sprites are drawn onto.

`Pygame.clock` keeps track of the tickspeed of the game and can act as an FPS counter.

`Pygame.group` is a class built into the Pygame library, essentially a list that sprites can be added to. This is necessary to be able to update all planets and the spacecraft in terms of the physics engine as it is otherwise difficult to add them to a traditional Python list to loop through.

`Pygame.image` is used to load images to sprites to then be drawn onto the screen.

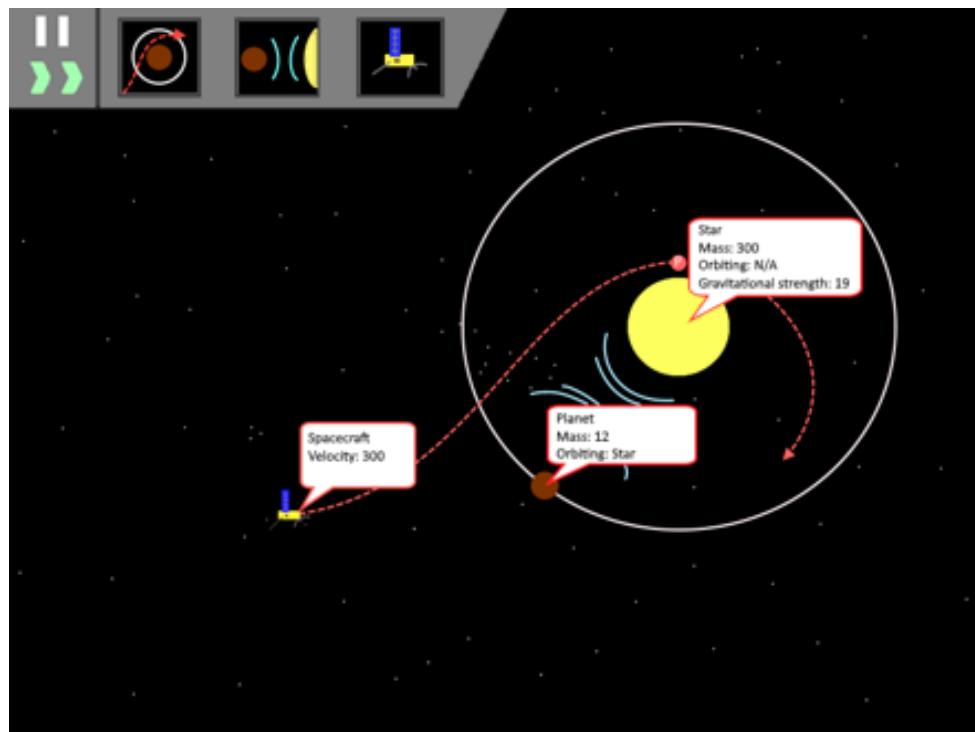
Pygame.event listens for keypresses and can run methods when requirements are met.

Numpy.poly1d holds plotted values of a 1-dimensional curve which can return a corresponding y-coordinate if a number is passed through.

Pygame.Vector2 holds two values representing a 2D vector and includes methods for manipulating and comparing vectors.

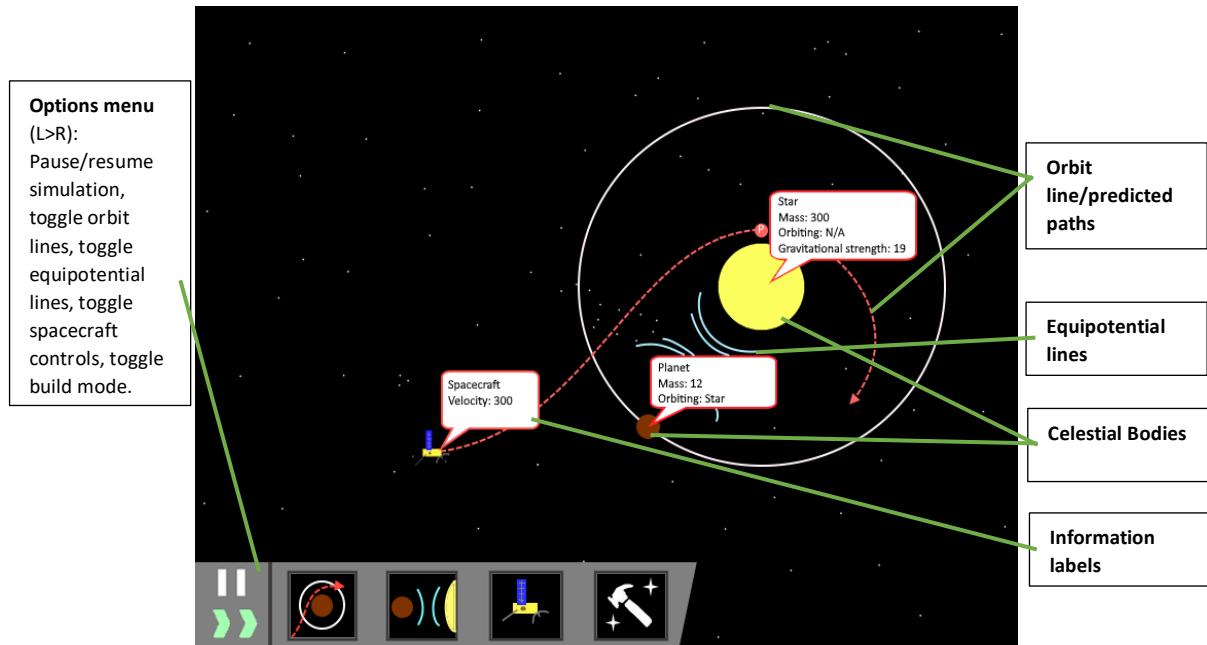
Pygame.rect acts as a collision mesh for a sprite and is generated from its image.

GUI

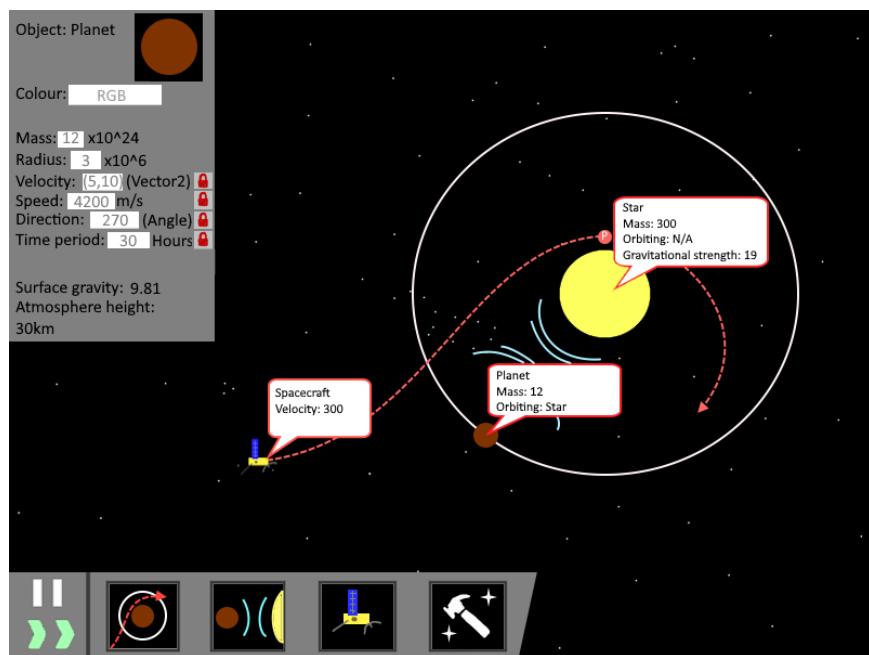


My initial sketch for this program was flawed; firstly, the position of the options menu at the top of the screen makes it difficult to reach on an interactive whiteboard. There is also no option to access tools for the planned build/edit menu. However, I was happy with the look of the rest of the program.

Below is my second rough sketch of the program. This one had a more accessible options bar at the bottom of the screen, as well as an extra button for the build mode options. However, there was still no mouse-accessible way to control any camera movement.



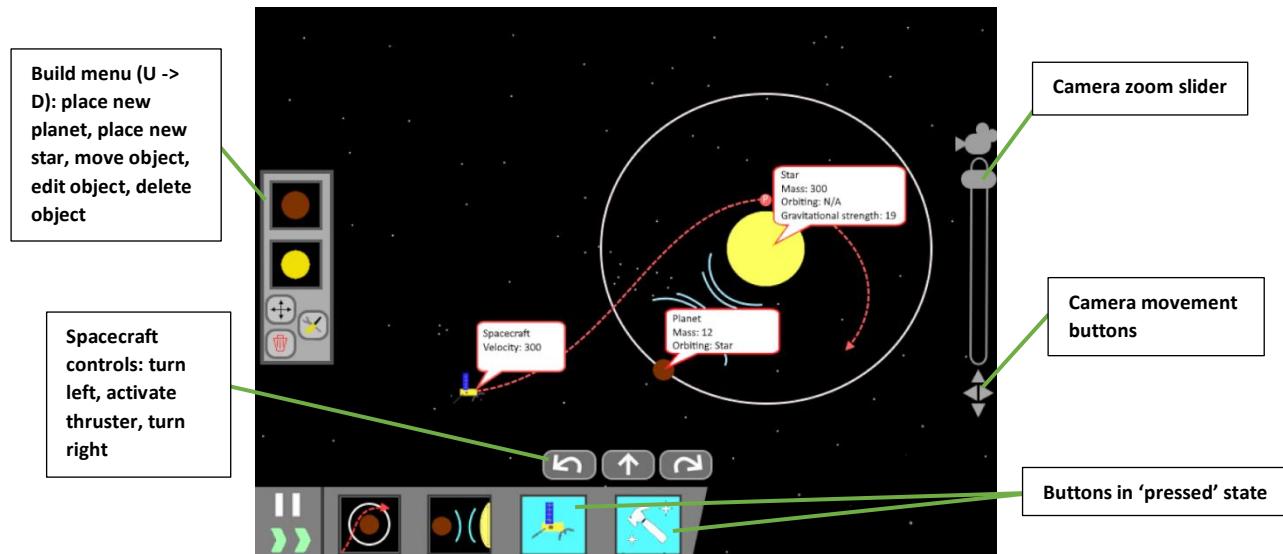
The third sketch was of the menu to change the parameters of objects in the simulation as it runs, which will show when an object is clicked, which includes options to lock certain parameters so that they do not change as the simulation is running. Certain variables such as mass and radius will not change as the program runs, and so do not require a button to lock them, and surface gravity and atmospheric height are calculated from the other variables input and so do not need to be able to be changed by the user.



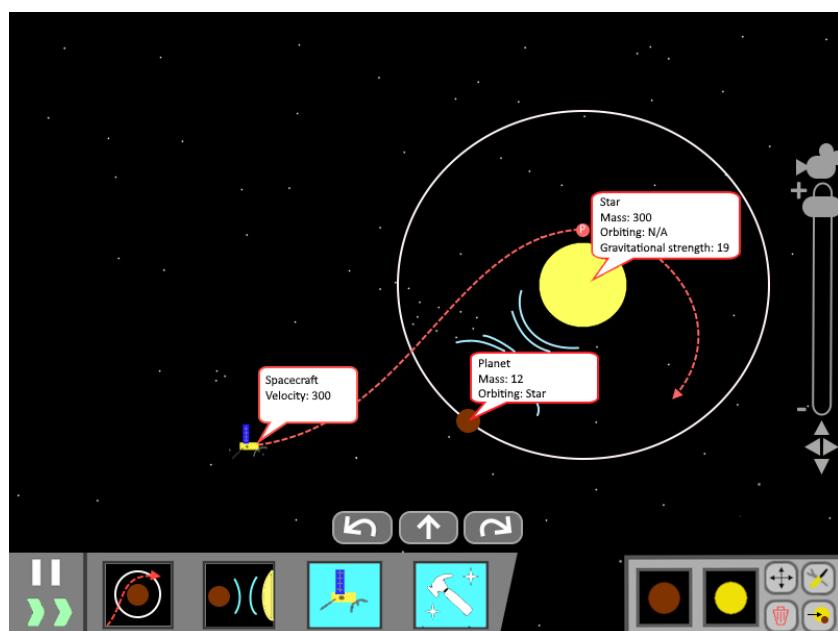
The fourth sketch incorporates a scroll bar for the camera zoom as well as buttons for movement at the right side of the program. The spacecraft control and build/edit mode buttons are in their blue 'pressed' state, and so the submenus for both are also visible on the screen.

The build/edit menu is on the left, with options for placing a new planet, a new star (effectively a planet but will be used as a part of atmospheric height calculations), and options to move, edit and delete objects present in the scenario.

The spacecraft control buttons are present just above the main options menu, with options to turn left or right and fire the engines. I decided not to include a backwards thruster, as it is more realistic for a spacecraft to have to turn around and use its main thruster to slow down or change direction.



After receiving feedback from stakeholders: Kai thought it would be useful to put a plus sign at the top of the zoom slider and a minus sign at the base in order to make it more obvious in which direction the slider zooms the camera. Oli said that it seems cluttered, so I will try moving the build menu down to the bottom right corner of the screen to better utilise that space. This version also includes a button to save and load scenarios, in the bottom right corner of the build menu.



When I went back to my stakeholders with this final design they were happy with it.

Usability

The simulation is intended to be run on classroom devices, many of which can be connected to an interactive whiteboard. Consequently, it would be ideal if the entire program is able to be controlled entirely using the whiteboard, which is to say through point-and-click controls. To account for the relative height of students, the controls should be positioned low or at the base of the screen to accommodate for the fact that these students may not be able to reach the top of an interactive whiteboard easily.

One area in which this full mouse control may not be possible is in the naming of files when saving a scenario, if that is implemented. However, as saving a scenario should be a relatively rare procedure, it would not disrupt a lesson by having the user need to briefly use a keyboard to name a file. One alternative solution to this would be to create an on-screen keyboard, but that is overly complex and is out of the scope of my project in the limited time I have.

Another area that has been mentioned by stakeholders is the use of a colourblind-friendly palette. For the best way of fitting this requirement, I will be using mainly black, white and grey for the UI, with any other colours having a high contrast between them and any adjacent colours. In the case of stars, the colour is determined by its temperature and is accurate to reality.

The program should be able to run without having to open the code in an editor; in the case of it being used as a web app, this should not be an issue as the program would be run when the webpage is first opened. If this program is to be run as an application, a desktop shortcut to run the program itself would be necessary to be able to be easily run by less tech-literate users.

Post-dev test plans

During development, I will be using mainly white box testing, with these post-dev test plans being for mainly black-box testing. These will also be used by stakeholders when beta testing, in addition to any other tests that they may wish to perform.

<i>Test #</i>	<i>Requirement(s)</i>	<i>Test description</i>	<i>Expected outcome</i>
1	1, 7E	Run program and resize window to screen.	Options menu should show on top of background, scaled. Window is correctly scaled to screen.
2	1	Interact with buttons on the options menu.	Menu buttons should open the corresponding menu/change the program accordingly.
3	1, 7A	Use interactive whiteboard to interact with buttons in options menu.	The buttons can be interacted with through the whiteboard's touch controls.
4	5B	Draw planet at game start.	Planet should be drawn to screen along with the options menu and background.
5	5B	Draw planet while game is running.	Planet should be drawn to screen in front of background.
6	5, 5B	Draw second planet in different location while game is running.	Second planet should also be drawn to screen.
7	5, 5B	Draw second planet at game start.	As above.
8	4	One planet object with physics properties.	No movement.

9	4, 4B, 5	Two planet objects created on top of one another.	No movement.
10	4, 4B, 5	Create two planet objects with physics properties, identical masses.	Planets should exert an equal force on each other and move as such.
11	4, 4B, 5	As above, but one planet has a greater mass.	Planet with lesser mass should experience a greater force of attraction and should move as such.
12	1E, 4	Play simulation in dual planet system and change simulation speed from minimum to maximum and vice versa.	Physics remains consistent. Planets slow down when simulation slows, and speed up when simulation sped up.
13	8	Camera movement by pressing/moving mouse in area.	Effect of the screen moving when controls pressed corresponding to the 4 cardinal directions.
14	8, 8A	Camera zooms in and out (no focus on objects).	Objects move and grow/shrink in size accordingly with physics consistent.
15	8, 8B	Cycle camera focus through different objects.	Camera moves between each and every object in the scenario with physics consistent.
16	8, 8A, 8B	Camera zooms in and out (focus on objects).	Objects move and grow/shrink in size accordingly with physics consistent.
17	3	Draw orbit lines for single planet system with a sideways velocity.	Orbit line drawn straight in direction of travel.
18	3, 4, 4B	Draw orbit lines for a dual planet system with both planets having equal upwards velocity.	Orbit lines bend to intersection point before correctly predicting collision/movement apart.
19	3, 3B	Same as above.	Apoapsis markers are present at point of intersection.
20	3, 3B	Draw orbit lines for dual planet system, one has velocity while other is stationary and has a higher mass.	Apoapsis and periapsis markers drawn at correct points along line.
21	3A	Draw equipotential lines for single planet system.	Concentric circles drawn around planet.
22	3A, 4B	Draw equipotential lines for dual planet system (equal mass).	Circular pattern with warped region in between planets.
23	3A, 4B	Draw equipotential lines for dual planet system (different masses).	Equipotential lines more grouped/larger diameter of circular pattern around planet of larger mass.
24	1F	Show information panel for each object in turn in a tri-planet system.	Correct information given for each planet.
25	4B, 4D	Dual planet system: lock all parameters for one planet, then play simulation.	No parameters change for locked planet, but do for unlocked planet. When switching panel between each

			planet, locked variables remain consistent.
26	1A, 1F, 4, 4B	Dual planet system: increase/decrease mass of one planet while simulation is running.	Force of attraction increases/decreases between the two objects. Change is reflected in the info panel for the object.
27	1B, 1F, 4, 4B	Single planet system: increase/decrease speed of planet while simulation is running.	Speed of object increases/decreases. Change is reflected in the info panel for the object. Velocity increases/decreases by correct amount in both component directions.
28	1C, 1F, 4, 4B	Single planet system: increase/decrease velocity of planet in one component direction while simulation is running.	Speed of object increases/decreases. Change is reflected in the info panel for the object. Velocity increases/decreases by correct amount in affected component direction, while remaining constant in the other.
29	1D, 1F, 4, 4B	Dual planet system: increase/decrease time period of one planet while simulation is running.	Orbital radius changes accordingly. Orbital velocity changes accordingly.
30	1, 2	Click spacecraft control show button, and rotate the spacecraft in both directions. Move spacecraft forwards.	Spacecraft control options show when corresponding button is clicked. Spacecraft moves in correct directions, and accelerates smoothly when respective button pressed.
31	2, 4	Single planet system: spacecraft is given horizontal velocity below the planet.	Spacecraft experiences force upwards and moves as such.
32	2, 4, 4A	Planet and star system: spacecraft moves into atmospheric region of planet.	Spacecraft experiences a frictional force inside the atmosphere, and slows down.
33	2, 4C	Single planet system: spacecraft impacts planet.	Spacecraft bounces off surface of planet elastically.
34	4, 4B, 4C	Dual planet system: equal masses and radii.	Planets move towards each other and gain momentum before colliding elastically and moving away from one another while slowing down.
35	1, 5A	Load each preset scenario in turn.	Each scenario is loaded correctly, with simulation paused.
36	1, 5C	Save custom scenario.	Scenario is saved in the correct location.
37	1, 5C	Load custom scenario just saved.	Scenario is loaded correctly.
38	1, 5C	Load custom scenario after closing and restarting program.	Scenario is loaded correctly.

39	6, 6A, 8	Start program, and move camera around.	Stars are represented on background, and move consistently when camera is moved.
40	6, 6B	Start program with any number of planets.	Textures for planets display correctly.
41	7B	Run program on lowest spec device that I have access to.	Program runs with acceptable performance (above 20FPS).
42	7C	Run program from a website via a web browser.	Program runs from website with acceptable performance (above 20FPS).
43	7D	Run program using a desktop shortcut.	Program starts correctly.

Development

Stage 1 – Main menu, creating and drawing planets/stars to the screen

Success criteria

NOTE: in all success criteria in each development section, extra features are marked with an (E).

REQUIREMENT	DESCRIPTION	SUCCESS CRITERIA	CHECK
1	Options menu	An easy to understand and navigable UI.	✓
5	Multiple planets/stars	Multiple planets and stars can be represented in the software at the same time.	✓
6	Simplistic graphics	Graphics are simple enough to not be distracting, but detailed enough that at a glance it can be understood what each object represents.	✓
7A	Navigable through use of mouse only	All features of the program are accessible via clicking or holding the mouse buttons.	✓
7E	Program is correctly scaled to any size device	Program correctly sizes to all monitor sizes.	✓

Testing plan

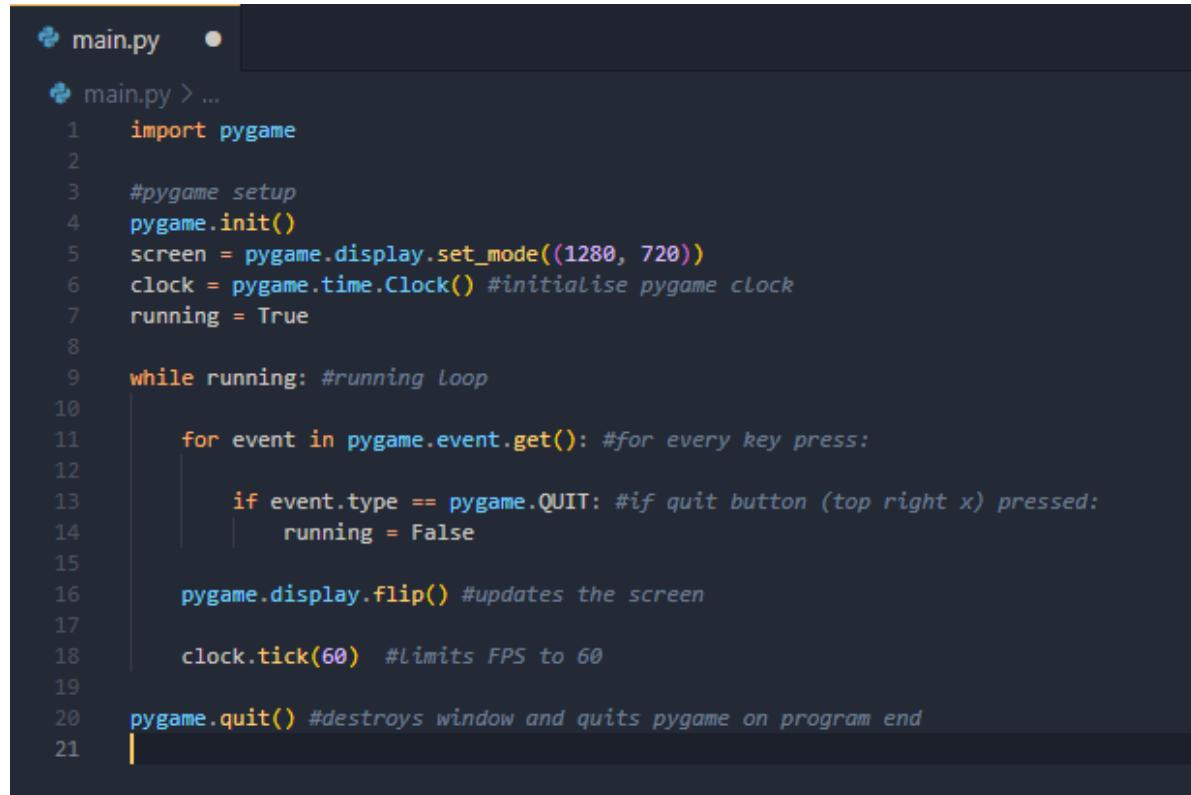
Test #	Test description	Expected outcome
1a	Planet.calcGravAtSurface(): Mass value of 1, radius of 1 (normal)	g returned as 6.67×10^{-11}
1b	Mass value of 5.972, radius of 1 (normal)	g returned as 9.81
1c	Mass value of 0, radius of 1 (boundary)	g returned as 0
1d	Mass value of 1, radius of 0 (erroneous)	Division by 0 error
1e	Mass value of 1, radius of 10000 (extreme)	g returned as 6.67×10^{-7}

1f	Mass value of -5, radius of 1 (extreme)	g returned as -333.5
2a	Planet draw function: Radius of 1 (normal)	Planet with radius 1 drawn on screen
2b	Radius of 0.005 (extreme)	Planet with radius 0.005 drawn on screen
2c	Radius of -1 (erroneous)	Error in Pygame draw function
2d	Radius of 0 (boundary)	Planet is instantiated, but cannot be seen
3a	Star.getColour(): Temperature of 6,000 (normal)	Yellow-white colour returned + drawn
3b	Temperature of 100,000 (extreme)	Blue colour returned + drawn
3c	Temperature of -1,000 (erroneous)	Error in Numpy
3d	Temperature of 0 (boundary)	Returns a reddish-purple colour + drawn

Development

'main.py' will be the root file for the program. I will be using visual studio code as my IDE due to my familiarity with the software.

The first step of development involves initialising Pygame. This will create a window that I can use to build everything from.



A screenshot of the Visual Studio Code interface showing the 'main.py' file. The code initializes Pygame, sets up a window, and creates a game loop with event handling and clock ticks.

```
main.py
1 import pygame
2
3 #pygame setup
4 pygame.init()
5 screen = pygame.display.set_mode((1280, 720))
6 clock = pygame.time.Clock() #initialise pygame clock
7 running = True
8
9 while running: #running Loop
10
11     for event in pygame.event.get(): #for every key press:
12
13         if event.type == pygame.QUIT: #if quit button (top right x) pressed:
14             running = False
15
16         pygame.display.flip() #updates the screen
17
18         clock.tick(60) #limits FPS to 60
19
20     pygame.quit() #destroys window and quits pygame on program end
21
```

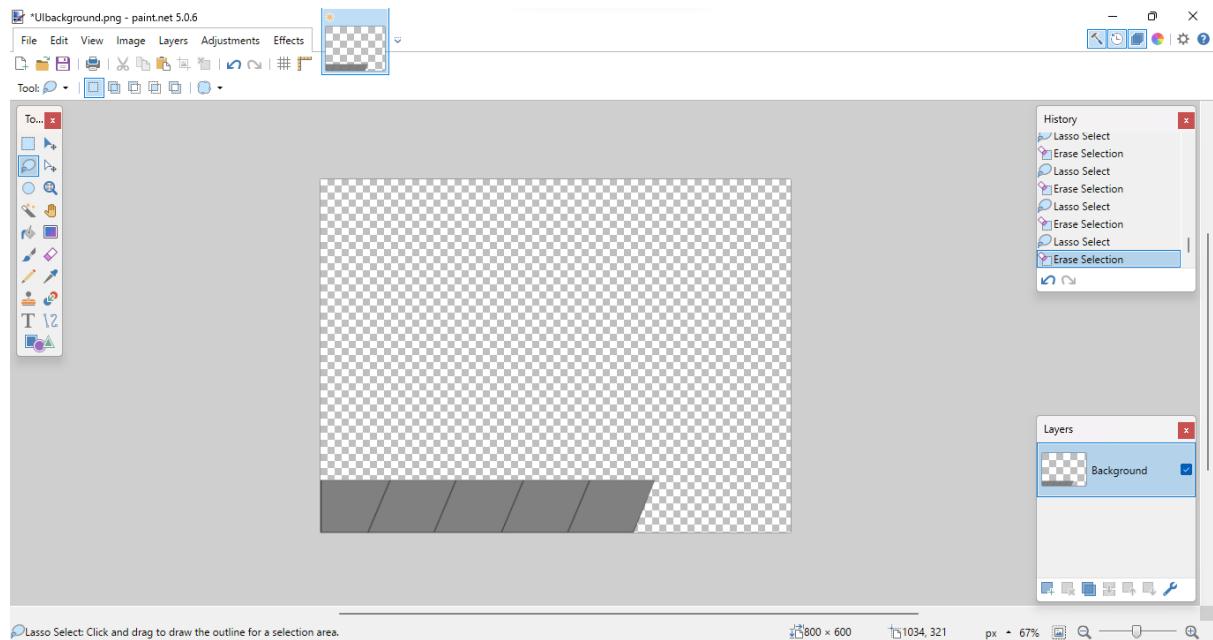
I fiddle with the screen display modes a bit to ensure that it scales to any device resolution. I will likely need to change this in the future to ensure that it works when I implement camera movement and draw objects.

```
4 pygame.init()
5 screen = pygame.display.set_mode(size=(500, 500), flags=pygame.SCALED|pygame.RESIZABLE) #sets up window to be able to be resized
6 clock = pygame.time.Clock() #initialise pygame clock
```

Next, I will create a background for the scene by filling with the colour black. I may change this in the future if I have time to instead paste an image for the background, but it is unnecessary at this point.

```
15
16     screen.fill("black") #background texture, may replace with image later
17
```

Now to display the options menu to the screen. I have opted to use an open-source image editing program called paint.net to draw the menus as I find it easier to use than other software.



Here is the menu base as drawn in paint.net. This will be always displayed on the base on the screen and will contain the buttons and icons of the menu itself.

To draw this onto the screen, it must first be loaded as an image into a variable outside of the running loop. This means that the image won't be loaded from file every time it is drawn.

```
24     img = pygame.image.load("UIbackground.png").convert()
25     img = pygame.transform.scale(img,(screen.get_width(),screen.get_height()))
26
27     running = True
28
29     while running: #running Loop
30
31         clock.tick(60) #Limits FPS to 60
32
33         for event in pygame.event.get(): #for every key press:
34
35             if event.type == pygame.QUIT: #if quit button (top right x) pressed:
36                 running = False
37
38
39             screen.fill("black") #background texture, may replace with image later
40
41             screen.blit(img, (0, 0))
42
43             pygame.display.flip() #updates the screen
44
45     pygame.quit() #destroys window and quits pygame on program end
46
```

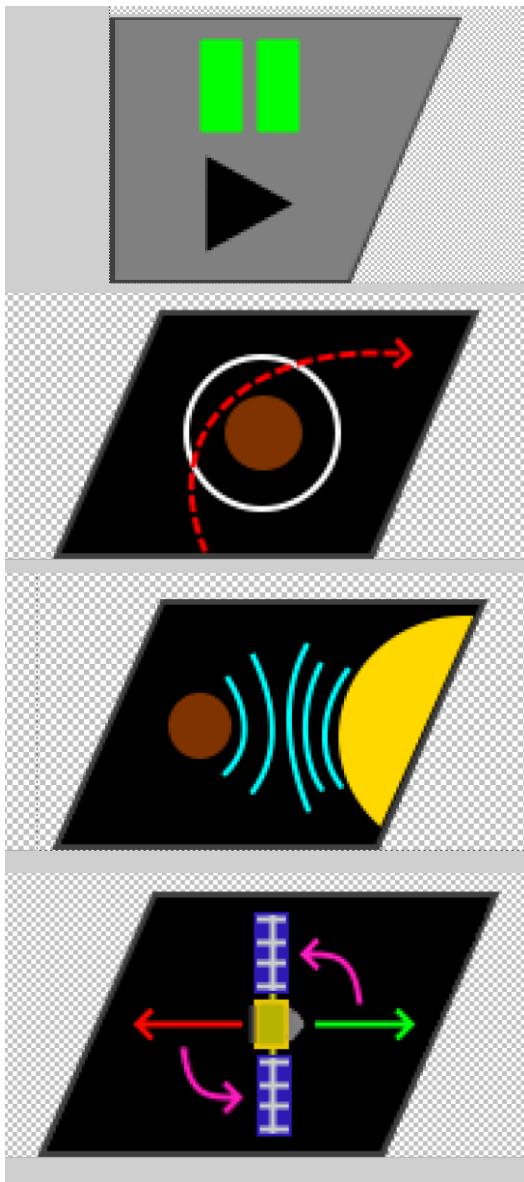
I initially had some trouble with the image not displaying correctly, but after adding the scale method (line 25) and removing the scaling flag (below) it was corrected. I also set a caption for the display, and set the default resolution to 720p/HD.

```
16     screen = pygame.display.set_mode(size=(1280, 720), flags=pygame.RESIZABLE) #sets up window to be able to be resized  
17     pygame.display.set_caption('Orbital Mechanics Simulator') #names window
```

Now that the UI background is displaying correctly, I must create separate buttons for each option. At this stage, these will just be placeholders. The challenge with this will be to link the buttons to scale with the options menu.

For this initial stage, I will just quickly draw these buttons in paint.net, which may look pixelated for now but can be easily changed later. This also allows me to change their appearance according to stakeholder feedback.

There are five buttons to be used with the program:

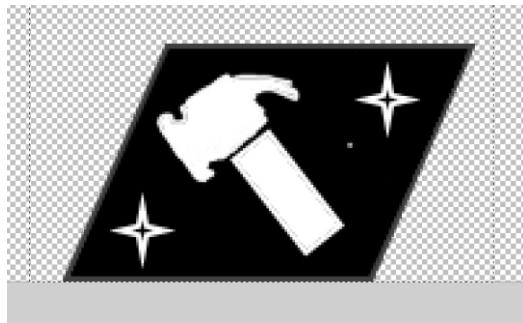


Play/pause: This will be used to pause and resume the physics calculations per tick and stop planets from moving.

Show/hide orbit lines: Orbit lines show the path of an object in an orbit, or its predicted direction of travel.

Show/hide equipotential lines: Unlike orbit lines, equipotential lines have the potential to clutter up the screen as they will be displayed across all space, and so the option to toggle them will be its own separate button to prevent confusion.

Toggle spacecraft controls: To change the speed and direction of the spacecraft, a controls menu is used. Keyboard input is not desired as it would not be accessible through interactive whiteboard controls.



Toggle build mode: Allows the user to access tools to edit objects in the simulation.

In addition to this, I also made separate versions of each of these buttons with the black background filled for light blue, to swap in to show when a button has been pressed.

To add the buttons to the screen, I initialise each one as a button class. To make for easy initialisation of all five buttons, I use a for loop to pass through the number of each button as its designation. This will allow me to easily set the images correctly.

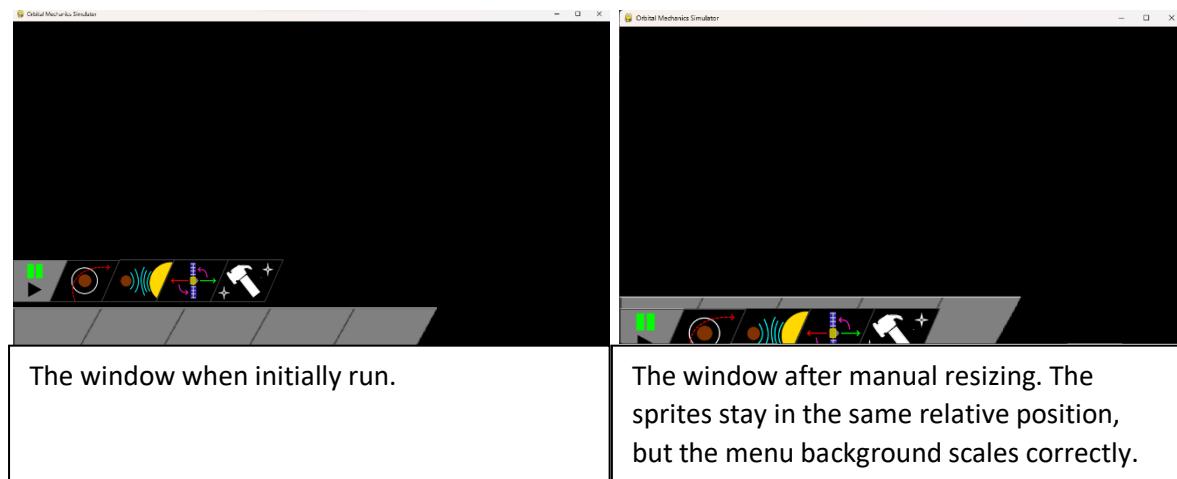
```
50     screen = pygame.display.set_mode(size=(1280, 720), flags=pygame.RESIZABLE) #set
51     pygame.display.set_caption('Orbital Mechanics Simulator') #names window
52     clock = pygame.time.Clock() #initialise pygame clock
53
54     #initialise buttons group
55     buttonGroup = pygame.sprite.Group()
56
57     for i in range(5): #create menu buttons
58         button = Button(i) #instantiate button
59         buttonGroup.add(button) #add to update group
60
```

```
class Button(pygame.sprite.Sprite):
    def __init__(self, designation): #init method
        pygame.sprite.Sprite.__init__(self) #initialise Pygame sprite
        self.designation = designation
        self.imageUnclicked = pygame.image.load("UIbutton" + str(designation) + ".png") #Load images from file
        self.imageClicked = pygame.image.load("UIbutton" + str(designation) + "Clicked.png")
        self.isClicked = False #set initial state
        self.image = self.imageUnclicked #set initial image
        #-----TODO: buttons scale to screen
        self.rect = self.image.get_rect() #get initial rect
```

Before I make the buttons scale to the screen, I will check that they display. Running for the first time:

```
File "e:\NEA (USB)\main.py", line 11, in __init__
    self.imageUnclicked = pygame.image.load("UIbutton" + str(designation) + ".png") #load images from file
                                                ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
FileNotFoundException: No file 'UIbutton0.png' found in working directory 'E:\NEA (USB)'.
PS E:\NEA (USB)> █
```

Of course, I have labelled the buttons in my folder as 1 to 5, not 0 to 4. Luckily, this is an easy fix with each button being renamed to fit this.



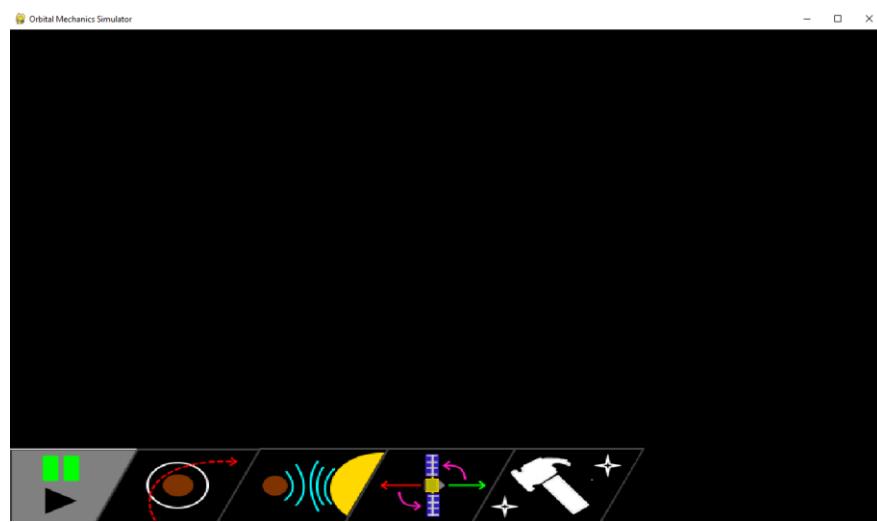
The buttons do display, and are in the correct order, but do not snap into the correct position or change in scale when the window is resized. As the draw function for the buttons are not present within the object and instead use the Pygame.sprite.group draw function, I loop over each button inside the main running loop whenever a window resizing event is called via the Pygame events system to scale them before drawing them. This uses the same scaling function that the menu background uses.

```
97     if event.type == pygame.QUIT: #if quit button (top right x) pressed:  
98         running = False  
99  
100    elif event.type == pygame.WINDOWRESIZED:  
101        for i in buttonGroup:  
102            i.image = pygame.transform.scale(i.imageUnscaled,(screen.get_width(),screen.get_height()))  
103
```

The buttons will be initially scaled in the initialisation function inside the object:

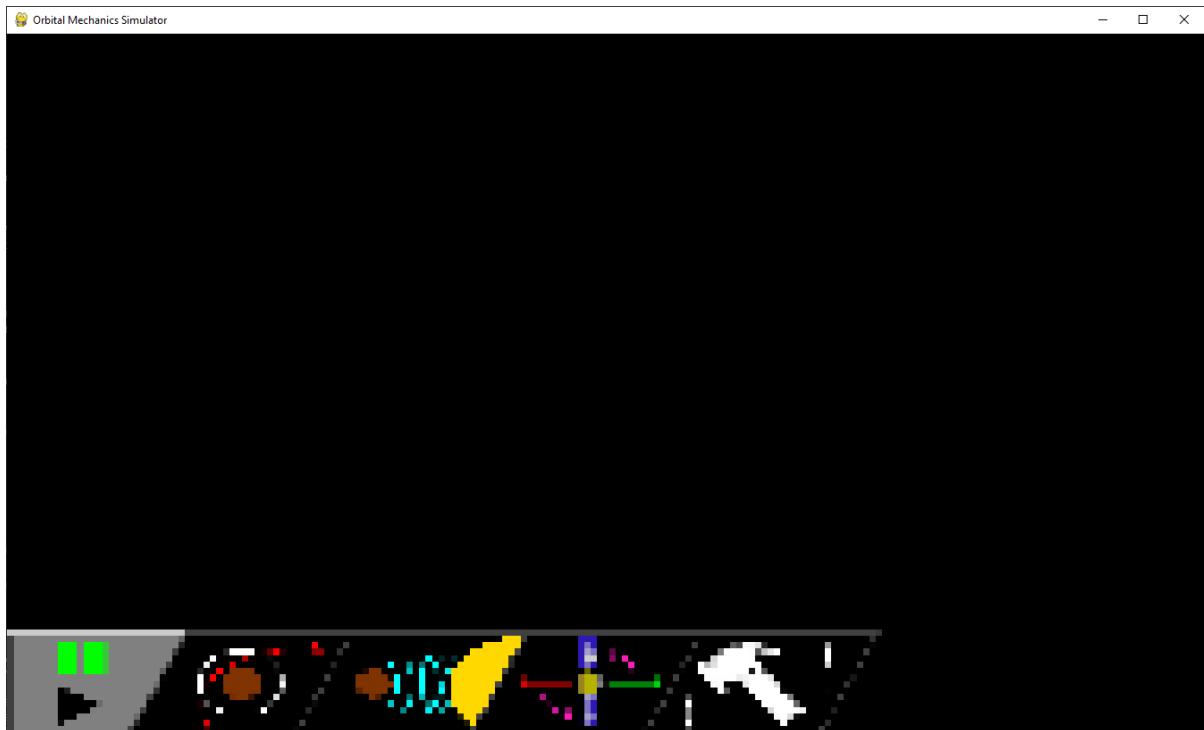
```
16     self.image = pygame.transform.scale(self.imageUnclicked,(screen.get_width(),screen.get_height()))  
17  
18
```

Running the program now:



The buttons lose some quality as they are stretched across the screen, and this effect intensifies as the window is stretched horizontally. However, this should not be an issue as it is not intended for the program to be run in a thin horizontal or vertical window, and as such I shall not be fixing it (at least not in this version).

The program also scaled correctly when I clicked the fullscreen button.



However, when I scaled the window to a small size and then large again, there was an extreme loss of quality in the button icons. In contrast to my previous comment about the window never being scaled to extreme vertical or horizontal resolutions practically, in low-resolution devices or even if the window was resized small enough accidentally, it would cause a permanent loss in this quality for the duration of the running of the program.

This issue was happening due to the `pygame.transform.scale()` function being applied to the current image of the button instead of the original one. This meant that the scaled quality of the image was kept, even after it had been downsized. To fix this, I changed the scaling code to scale from the original image instead; most of the `self.image` references are changed to `self.imageUnscaled`, which holds the original image of whichever state the button is currently in, which I will then scale from:

```
110     for i in buttonGroup:  
111         i.image = pygame.transform.scale(i.imageUnscaled,(screen.get_width(),screen.get_height()))
```

```
class Button(pygame.sprite.Sprite):  
  
    def __init__(self, designation): #init method  
        pygame.sprite.Sprite.__init__(self) #initialise Pygame sprite  
  
        self.designation = designation  
  
        self.imageUnclicked = pygame.image.load("UIbutton" + str(designation) + ".png") #load images from file  
        self.imageClicked = pygame.image.load("UIbutton" + str(designation) + "Clicked.png")  
  
        self.isClicked = False #set initial state  
        self.imageUnscaled = self.imageUnclicked #set initial image  
  
        self.image = pygame.transform.scale(self.imageUnscaled,(screen.get_width(),screen.get_height()))  
  
        self.rect = self.image.get_rect() #get initial rect
```

This fixed the issue, and there is now no loss in quality when the window is rescaled from a small size.

Having created the buttons, I now realise that the menu background will never be displayed above the buttons, and as such is no longer needed. Therefore, I removed the menu background and all functions associated with it.

To add functionality to the buttons, I use the Pygame events trigger system to detect when the mouse is clicked. So that the program wouldn't unnecessarily use processing power to constantly detect whether or not a button had been pressed, I made use of the Pygame events system as I have for the basic quit function and the resizing.

```
95     for event in pygame.event.get(): #for every key press:  
96  
97         if event.type == pygame.QUIT: #if quit button (top right x) pressed:  
98             running = False  
99  
100        elif event.type == pygame.WINDOWRESIZED: #if the window has been resized:  
101            for i in buttonGroup:  
102                i.image = pygame.transform.scale(i.imageUnscaled,(screen.get_width(),screen.get_height())) #scales options menu button  
103  
104        elif event.type == pygame.MOUSEBUTTONDOWN: #if the mouse has been clicked (or interactive whiteboard tapped)  
105            buttonGroup.update() #run checks for buttons
```

I will use the button update function to detect and change the image of the object using a method inside the object itself. This makes use of the inbuilt `rect.collidepoint()` feature of Pygame sprites to detect whether the mouse cursor is over the button when the mouse button is pressed.

```
23     def update(self):  
24         if self.rect.collidepoint(pygame.mouse.get_pos()):  
25             if self.clicked: #if button is in clicked state  
26                 self.image = self.imageUnclicked #change graphic  
27                 self.clicked = False #set logic to be in unclicked state  
28             else: #if button is not in clicked state  
29                 self.image = self.imageClicked #change graphic  
30                 self.clicked = True #set logic to be in clicked state
```

Running the program to test, clicking the buttons seems to have no effect. I commented in some debug lines to see what was going wrong:

```
33     def update(self):  
34  
35         print("br1")  
36  
37         if self.rect.collidepoint(pygame.mouse.get_pos()):  
38             print("br2")  
39             if self.clicked: #if button is in clicked state  
40                 print("br3")  
41                 self.imageUnscaled = self.imageUnclicked #change graphic  
42                 self.clicked = False #set logic to be in unclicked state  
43             else: #if button is not in clicked state  
44                 print("br4")  
45                 self.imageUnscaled = self.imageClicked #change graphic  
46                 self.clicked = True #set logic to be in clicked state  
47
```

After running the program and clicking once, the program output "br1", "br2", "br4" 5 times, showing that the button update was called at the right time, the button was trying to switch to the correct state and either all of the buttons were being updated on that single click, or a single button was being activated 5 times successively. I realised that the reason that the buttons would not

update their image is because the actual `self.image` variable that Pygame uses to draw the buttons to the screen was not changed, only the scaled image. Therefore, I added the image scaling line from the initialisation method at the end of the update method to change this.

Now, when I clicked once, all five of the buttons visually changed state, showing that the collision for the buttons is not working properly. Wherever I clicked on the screen, all of the buttons would change state. If I resized the window to be larger, the region that I could click to make the buttons to change state would shrink; this told me that there was an issue with the scaling of the rect that the `update()` method would compare against the mouse cursor position.

To first fix the collision scaling issue, I set the rect of the sprites as I have in the `__init__()` method whenever I rescale the screen. I realized that, because I set and rescale the image 3 times in the program, it would probably be a good idea to create a new method for each of the buttons to call when they need to be rescaled; this ensures that they will all rescale in the same way each time, and it will be easier to modify/maintain my code:

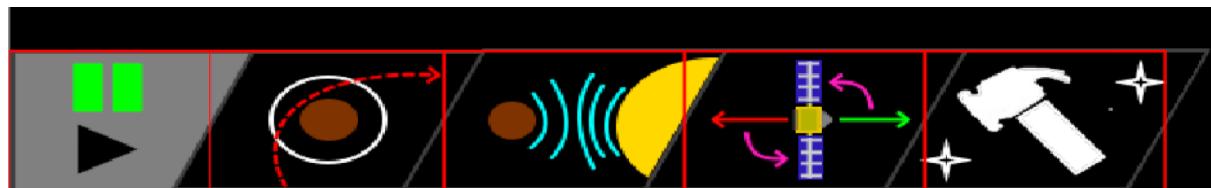
```
31  def rescale(self):  
32  
33      self.image = pygame.transform.scale(self.imageUnscaled,(screen.get_width(),screen.get_height())) #rescale to window size  
34      self.rect = self.image.get_rect() #gets collision rect for sprite  
35
```

I then changed every instance of the button resizing (in `__init__()`, `update()` and the events loop within the main running loop) to instead call this method. I ran the program to test, and my changes functioned correctly.

However, I still have the issue of the buttons all activating when any part of the screen is pressed; as the images of the buttons are the size of the entire screen (the parts around the actual buttons themselves have an alpha value of 0 and so cannot be seen), the rects generated from these images are the size of the full image, and so are present across the entire screen for each button.

To fix this, I will have to resize the rects over the buttons at the start of the program. I load the different images from each file using a number passed through as a designation, so for the rects I could position them along the x-axis in order according to the designation number, as the buttons should all be the same size.

Unfortunately, a rect, as short for ‘rectangle’, cannot be resized to become a specific shape as I thought previously. As the buttons I have designed are not rectangular in shape, there may be some overlap between the area that a button can be clicked with an adjacent button.



The red boxes above show how the rects will have to be positioned.

As the button designs are not final, I shall ignore the issue of the rects not lining up for now and instead take it into consideration when designing the final buttons.

When the buttons are scaled to the screen, the rects must be scaled as well. To do this, and position them on their respective button, I will first scale all of the rects of these main menu buttons into the bottom left corner of the screen before positioning them to the right as determined by their designation.

I can use the `rect.update()` function from Pygame to set the width, height and position easily with the width and height of the window. I can then position the rects (which have to be positioned by passing the coordinate of the top left coordinate of the rectangle) by passing the coordinate `[(height of rect),(width of rect * designation + offset)]`. An offset is required as the speed control button is a different width to the other buttons.

Using paint.net, I can see that the ratio of the height of the buttons to the scaled size of the window is around 5:1, which means that the height of the rect must be the height of the window divided by 6. Doing a similar thing with the average width of the buttons, the width of each rect must be the width of the window divided by 10. For now, I will set the offset as 1, as it will be changed when I can see it all working.

```
def rescale(self):  
    self.image = pygame.transform.scale(self.imageUnscaled,(screen.get_width(),screen.get_height())) #rescale to window size  
    self.rect.update((screen.get_height()/6), (screen.get_width()/10) * self.designation + 1, (screen.get_width()/10), (screen.get_height()/6))
```

Running the program now, the only button that displayed was the speed control button, and none of the others. I then realized that I got the left and top parameters the wrong way round in the code above.

Running the code again after fixing, I was met with a blank screen. After setting the background colour to white and still not seeing anything, I printed the coordinates of the rects of the buttons out to console for debugging, which showed me that they were scaling correctly but were just in the wrong position. I set up code to print my mouse position whenever I clicked in the program, and I found that the rects were at the top of the screen instead of at the bottom; I had neglected to account for the fact that Pygame rect coordinates are counted from the top, not the base. Luckily, this is easily fixed by subtracting the top coordinate from the window height.

The rects are now positioned correctly, but the buttons are still not showing. I tried changing the `convert_alpha()` in the image loading to just `convert()`, and this made a black bar appear across the screen on the same level as the buttons would be.



I then tried setting the alpha value of the surfaces to zero, which only made the black bar go away and return to a blank screen. I removed this change and reverted back to using `convert_alpha()`. Delving further into the Pygame documentation, I discovered that the rect of the sprite is the rect that the `group.draw()` method uses to position the sprite on the main surface. To solve this, I split the rect for the collisions of the buttons into a new variable, `collisionRect`. This will replace all

instances of rect in the button class except for the rect within the `__init__()` method, as a rect is still needed to draw the button to the screen.

```
class Button(pygame.sprite.Sprite):
    def __init__(self, designation): #init method
        pygame.sprite.Sprite.__init__(self) #initialise Pygame sprite
        self.designation = designation
        self.imageUnClicked = pygame.image.load("UIButton" + str(designation) + ".png").convert_alpha() #load images from file
        self.imageClicked = pygame.image.load("UIButton" + str(designation) + "Clicked.png").convert_alpha()
        self.isClicked = False #set initial state
        self.imageUnscaled = self.imageUnClicked #set initial image
        self.rect = self.imageUnscaled.get_rect() #get initial rect
        self.collideRect = pygame.Rect(0,0,0,0)
        self.rescale()

    def rescale(self):
        self.image = pygame.transform.scale(self.imageUnscaled,(screen.get_width(),screen.get_height())) #rescale to window size
        self.collideRect.update((screen.get_width()/10) * self.designation + 1, screen.get_height() - (screen.get_height()/6), (screen.get_width()/10), (screen.get_height()/6))
```

Running the program now, it functions correctly, and the buttons are finally displayed. The collision rects were positioned slightly wrong, so I adjusted the offset and the widths until I found the right values, which turned out to be no offset at all and a width divided by 7.

I have chosen not to complete the submenu buttons and the features/actions associated with pressing these main buttons in this stage of development, as it is not the focus and changes may have to be made to the buttons themselves, especially in light of the rects.

With the main menu buttons complete, I now start work on the planets class. I begin by setting up the initialisation method according to the design. I realized that the Vector2 variable by definition has direction, so the direction variable was unnecessary; it was subsequently removed. Until I have completed the camera functions, I will just use the given position as the position to draw the planet onto the screen.

In order to scale the mass and radius values to what they would be like in reality, without having to draw them on the screen as such, I will multiply the mass values by 10^{24} , and the radius values by 10^6 in any calculation method before I use them in equations.

```
43 class Planet(pygame.sprite.Sprite):
44
45     def __init__(self, radius, mass, velocity, colour, pos): #init method
46
47         pygame.sprite.Sprite.__init__(self) #initialise Pygame sprite
48
49         self.colour = colour
50         self.radius = radius
51         self.mass = mass
52         self.gamePos = pos
53         self.screenPos = pos #set position on screen initially, TODO: get position at start from camera
54         self.velocity = velocity
55
56         #correct values for use in calculations
57         self.correctedMass = self.mass * 10^24
58         self.correctedRadius = self.radius * 10^6
59
60         self.surfaceGrav = self.calcSurfaceGrav() #initially calculate the surface gravity
61
62         self.image = pygame.Surface([radius*2, radius*2]) #get surface to draw sprite onto
63         self.image.set_alpha(0)
64
65         self.rect = self.image.get_rect() #gets collision rect for sprite -----TODO: is
```

The first iteration of these planets will just be circles drawn on the screen, which satisfies the ‘simplistic graphics’ requirement.

The lines I used to initialise a single planet at game start were written just before the main Pygame running loop:

```
56     test = Planet(10, 1, pygame.Vector2(0,0), (150,150,150), pygame.Vector2(250,250))  
57     physObjects.add(test) #adds object to sprite group to update each tick
```

When running this code for the first time, it would return an error in the initialisation of the sprite. I then realized that there was an error in the name of the function in line 6; the constructor method should not be called `Planet()`, it should be `__init__()` instead.

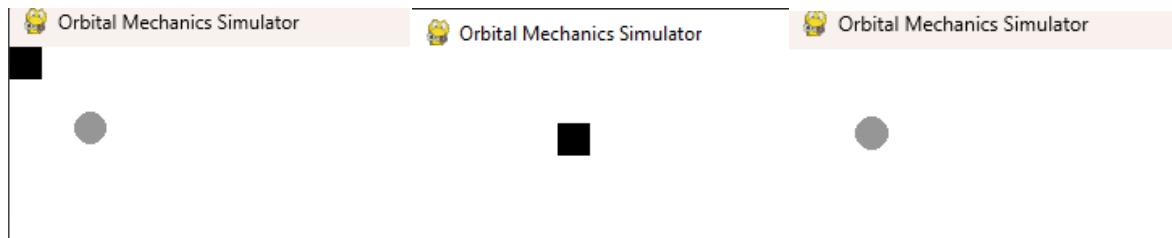
```
6         def __init__(self, radius, mass, velocity, colour, pos): #init method
```

I also noticed that the `self.image.fill` line was incorrect; Pygame draws sprites by placing them onto a surface, and I was drawing the circle of the planet onto this square. In this line, I was filling the square, not the circle, with the colour.

When running the program while filling the background with a different colour, in order to check for any issues with displaying sprites (order of display, whether or not anything else is displayed), I discovered that the surfaces of the planet sprites would generate as a black square in the top left corner of the screen; this happened as I was directly drawing the circles to the main screen surface instead of the object’s surface. This is not an issue at the moment as the surface of the object was planned to be used if textures for the planets are implemented, and drawing the circles onto the main screen surface has the same effect.

First, to make the surface move with the planet, I updated the rect (rectangular coordinates) of the surface each tick in the `update()` method using the `self.image.get_rect()` method, with the position of the object passed as the parameter for the centre.

The solution to the image of the box is to just set the alpha value of the surface to zero to make it invisible. I did this underneath the `self.image = Pygame.Surface(...)` line in the planet initialisation method, using the line `self.image.set_alpha(0)`.



Next to code are the `applyForce()` and `calcGravAtSurface()` methods. The latter was relatively easy, just making use of an equation to calculate acceleration due to gravity at the surface. This will be called whenever the mass or radius parameters are changed.

```
65     def calcSurfaceGrav(self): #calculate what the gravitational force would be on the surface  
66         #g = GM/R^2  
67  
68         g = (6.67*10^-11*self.correctedMass) / (self.correctedRadius^2)  
69         return g  
70
```

Acceleration due to gravity = gravitational constant * mass / radius²

`applyForce()` is the link between the object and the physics engine and is called in the physics engine update. At this point, it just takes in the force given by the gravity calculation function and adds it to the velocity. To accommodate for the direction the force should be applied in, the parameter passed through is a list, `vect`, with the 0-indexed item representing the magnitude of the force and index 1 representing the direction.

```
21     def applyForce(self, vect):
22         #f = ma
23         #a = f/m
24         #v = at (t is 1 in this situation)
25
26         #vect[0] = force
27         #vect[1] = direction
28
29         self.velocity += vect[1] * (vect[0] / self.mass)
```

Finally, `update()` is utilised by the Pygame.group structure to simultaneously apply commands to sprites each tick. This will be used to change the position of the sprite according to its velocity and draw the circle representing the planet at the resulting location. As the physics engine hasn't been written yet, this should just draw the circle and print the position of the sprite.

```
def update(self):
    self.gamePos += self.velocity #move sprite

    self.screenPos = self.gamePos #set position on screen to position in terms of game TODO: convert via camera function
    pygame.draw.circle(screen, self.colour, self.screenPos, self.radius) #draw planet

    #print(self.pos) #debug
```

To create the star object, I first completed the `__init__()` method:

```
73 class Star(Planet):
74
75     def __init__(self, radius, mass, velocity, pos, temperature):
76
77         self.temperature = temperature #set temperature variable
78
79         colour = self.getColour(temperature)
80
81         super().__init__(radius, mass, velocity, colour, pos) #initialise parent class
82
83     def getColour(self, temperature):
84         pass
85
```

Next, I imported Numpy into the program as it was a dependency for the code I was going to use to calculate the proper colour for the star.

```
3     import numpy as np
4
5     #setup for star colour function - global variables so that they do not have to be stored in multiple star objects.
6     redco = [ 1.62098281e-82, -5.03110845e-77, 6.66758278e-72, -4.71441850e-67, 1.66429493e-62, -1.50701672e-59, -2.42533006e-53
7     greenco = [ 1.21775217e-82, -3.79265302e-77, 5.04300808e-72, -3.57741292e-67, 1.26763387e-62, -1.28724846e-59, -1.84618419e-53
8     blueco = [ 2.17374683e-82, -6.82574350e-77, 9.17262316e-72, -6.60390151e-67, 2.40324203e-62, -5.77694976e-59, -3.42234361e-53
9
10    redco = np.poly1d(redco)
11    greenco = np.poly1d(greenco)
12    blueco = np.poly1d(blueco)
```

The lines of code to generate the curves to read the colour from are stated here, and the curves themselves are stored as global variables so that they do not have to be re-generated whenever a

new star object is created, or its colour is changed. Back in the star class, I paste the rest of the code for the function into the `getColour()` method:

```
83 def getColour(self, temperature): #gets colour for star (code for function by DocLeonard on Stack Overflow)
84
85     red = redco(temperature)
86     green = greenco(temperature)
87     blue = blueco(temperature)
88
89     if red > 255:
90         red = 255
91     elif red < 0:
92         red = 0
93     if green > 255:
94         green = 255
95     elif green < 0:
96         green = 0
97     if blue > 255:
98         blue = 255
99     elif blue < 0:
100        blue = 0
101
102     color = (int(red), int(green), int(blue))
103
104     return color
```

After completing the star and planet classes, I ran the program to check that they were functioning, with these objects being created:

```
141 testStar = Star(100, 1, pygame.Vector2(0,0), 5778, pygame.Vector2(250,250))
142 physObjects.add(testStar) #adds object to sprite group to update each tick
143
144 test2 = Planet(6.378, 5.972, pygame.Vector2(0,0), (180, 0, 180), pygame.Vector2(350,350))
145 physObjects.add(test2)
```

However, I received this error message:

```
File "e:\NEA\proj\main.py", line 95, in __init__
    super().__init__(radius, mass, velocity, colour, pos) #initialise parent class
    ~~~~~~
File "e:\NEA\proj\main.py", line 60, in __init__
    self.surfaceGrav = self.calcSurfaceGrav() #initially calculate the surface gravity
    ~~~~~~
File "e:\NEA\proj\main.py", line 71, in calcSurfaceGrav
    g = (6.67*10^-11*self.correctedMass) / (self.correctedRadius^2)
    ~~~~~~^~~~~~
TypeError: unsupported operand type(s) for ^: 'float' and 'int'
PS E:\NEA\proj> 
```

This seemed to be because the program was messing up on its order of operations; I first tried to fix this by moving the 10^{11} to the rightmost part of the equation. I received the same error. I then put brackets around the 10^{11} part, which worked as I got a different error message this time:

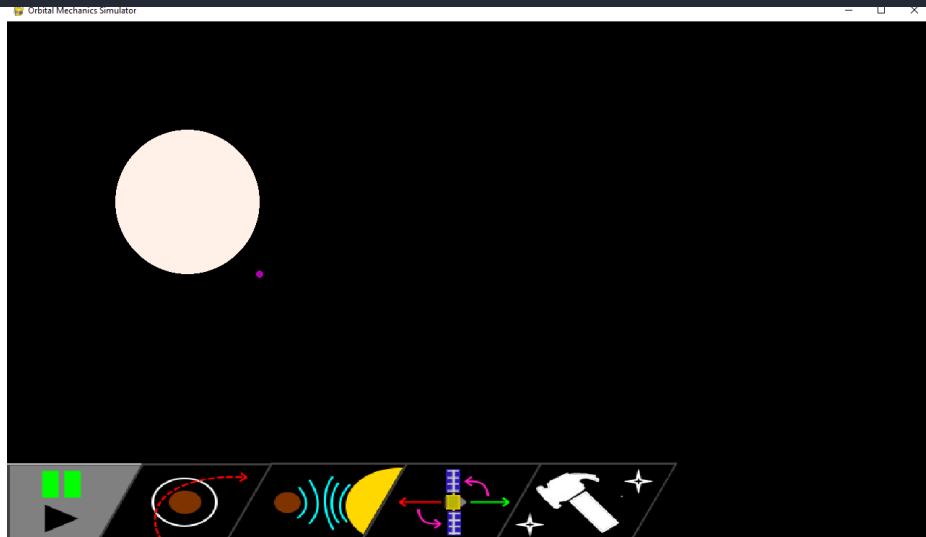
```
File "e:\NEA\proj\main.py", line 57, in __init__
    self.correctedMass = self.mass * 10^24
                                         ^
TypeError: unsupported operand type(s) for ^: 'float' and 'int'
PS E:\NEA\proj> 
```

I assumed it was for the same reason as before, so I put brackets around every instance of a number being raised to a power in the program.

```
File "e:\NEA\proj\main.py", line 60, in __init__
    self.surfaceGrav = self.calcSurfaceGrav() #initially calculate the surface gravity
    ~~~~~
File "e:\NEA\proj\main.py", line 71, in calcSurfaceGrav
    g = (6.67*(10^-11)*self.correctedMass) / (self.correctedRadius^2)
    ~~~~~~^~~~~~
TypeError: unsupported operand type(s) for ^: 'float' and 'int'
PS E:\NEA\proj> 
```

Yet another error. As this particular value was squared, I could just make the number multiply by itself directly.

```
g = (6.67*(10^-11)*self.correctedMass) / (self.correctedRadius*self.correctedRadius)
```



Success! The larger star object was displaying with the correct colour from the `getColour()` method, and both objects were the correct size relative to each other. I wanted to see the frames per second of this program, so I used the Pygame clock I initialised earlier and the `print(clock)` command to see that I was getting an average of 16 frames per second (FPS).

```
<Clock(fps=16.34)>
<Clock(fps=16.34)>
<Clock(fps=16.34)>
<Clock(fps=15.17)>
<Clock(fps=15.17)>
<Clock(fps=15.17)>
PS E:\NEA\proj> 
```

This confused me, as the physics engine hadn't been implemented yet and so there should be no calculations being completed every tick that would slow it down. I attempted to fix this by changing the `clock.tick` line in the main running loop, as this was the only thing I could think of that might be affecting this.

Setting it to 30 had little effect, as did setting it to 120 or removing the limit. Removing the clock altogether wasn't an option as it would remove the functionality of the clock and make it return 0 FPS each tick. In case the printing of the FPS every tick was affecting this value, I set it so that the FPS counter would only print once, when I clicked the mouse button; however, there was no change. I knew that this was not due to any low performance of the computer as I was running this program on a device that was more than sufficient for this task at the time.

I then removed the button draw line and the FPS limits in the main loop to check if the buttons were affecting performance, and the frames suddenly jumped to ~760FPS. This told me that the Pygame draw command was seriously affecting the performance of my program, and so I did some more research in the documentation for Pygame to see if my implementation was the most optimal, as well as any other potential workarounds.

I found that when I was initially loading the images of the buttons from file, they were not automatically converted to the pixel format of the screen; instead, the `blit()` function called by the `pygame.sprite.group.draw` method was having to convert this each tick. The solution to this was to add the `.convert_alpha()` suffix to the image loading lines to convert them on load.

```
<Clock(fps=60.61)>
<Clock(fps=60.61)>
<Clock(fps=60.61)>
<Clock(fps=60.61)>
PS E:\NEA (USB)\proj>
```

Now, my FPS is back at expected levels, and I can continue to stage 2 with great performance.

Testing + Bug fixing

<i>Test #</i>	<i>Test description</i>	<i>Expected outcome</i>	<i>Actual outcome</i>	<i>Pass/fail</i>
1a	Planet.calcGravAtSurface() : Mass value of 1, radius of 1 (normal)	g returned as 66.7	-0.83375	Fail (fixed)

This was a concerning result as it seemed so far off of the correct answer. I rechecked the logic, which was correct, so I then looked at the only possible source of the negative value: the 10^{-11} of the gravitational constant. As there were issues with the powers in this calculation before, I rechecked the Python syntax for powers in mathematical operations. It turns out my raising 10 to the power syntax was completely incorrect, as expected, and I changed this to the correct method of `**`.

This provided the correct result of 66.7.

<i>Test #</i>	<i>Test description</i>	<i>Expected outcome</i>	<i>Actual outcome</i>	<i>Pass/fail</i>
1b	Mass value of 5.972, radius of 6.3781 (normal)	g returned as 9.79	9.79181015558458	Pass
1c	Mass value of 0, radius of 1 (boundary)	g returned as 0	0.0	Pass
1d	Mass value of 1, radius of 0 (erroneous)	Division by 0 error	ZeroDivisionError: float division by zero	Pass?

```
def calcSurfaceGrav(self): #calculate w
    #g = GM/R**2
    if self.correctedRadius == 0:
        return 0
```

I decided that it was probably not a good thing to allow an error such as this to be present in the program, so I wrote a check to return 0 if this was the case.

<i>Test #</i>	<i>Test description</i>	<i>Expected outcome</i>	<i>Actual outcome</i>	<i>Pass/fail</i>
1e	Mass value of 1, radius of 10000 (extreme)	g returned as 6.67×10^{-7}	6.67e-07	Pass
1f	Mass value of -5, radius of 1 (extreme)	g returned as -333.5	-333.5	Pass
2a	Planet draw function: Radius of 1 (normal)	Planet with radius 1 drawn on screen	Planet with radius 1 drawn on screen	Pass
2b	Radius of 0.005 (extreme)	Planet with radius 0.005 drawn on screen	Planet was too small to appear, but other	Tentative pass, might need a

			processes worked as expected.	retest after camera zoom done
2c	Radius of -1 (erroneous)	Error in Pygame draw function	Pygame error: Invalid resolution for Surface	Pass?

Again, it would probably be a good idea to include a catch for any invalid data. In the case of the radius, I will just assume that data has been entered incorrectly, and to account for this I will take the absolute value for the radius when setting it in the planet initialisation method. Re-running the test, it has set the radius to 1 and drawn it as such.

Test #	Test description	Expected outcome	Actual outcome	Pass/fail
2d	Radius of 0 (boundary)	Planet is instantiated, but cannot be seen	Planet cannot be seen but is able to return a debug print statement in the update method.	Pass
3a	<code>Star.getColour()</code> : Temperature of 6,000 (normal)	Yellow-white colour returned + drawn	This colour was returned and drawn.	Pass
3b	Temperature of 210,000 (extreme)	Blue colour returned + drawn	This white colour was returned and drawn.	Fail (fixed)

A surface temperature of 210,000K is the highest recorded surface temperature of any known star. As stated in the post that I got this code from, the curve is only accurate up to 40,000K. Past this point, all stars should have the blue colour, so I will fix this by setting the RGB values of every star over 40,000K to be (0, 0, 255). This returned this blue colour.

<i>Test #</i>	<i>Test description</i>	<i>Expected outcome</i>	<i>Actual outcome</i>	<i>Pass/fail</i>
3c	Temperature of -1,000 (erroneous)	Error in Numpy	This white colour was returned and drawn.	Fail (fixed)

As with the previous test, the requested temperature is outside the range of the graph, however I will assume as with the planets' radii that the data was incorrectly entered as negative and take the absolute. This red colour was the result:

<i>Test #</i>	<i>Test description</i>	<i>Expected outcome</i>	<i>Actual outcome</i>	<i>Pass/fail</i>
3d	Temperature of 0 (boundary)	Returns a reddish-purple colour + drawn	This purple colour was returned and drawn.	Pass

I will count this as a pass, as no star is as cold as 0K (lowest discovered so far is 225 to 260K, or -48.5 to -13.5C) and the colours in between are going to be a redder purple.

At the end of testing, I re-ran all tests from this section in order to ensure that none of my bugfixing had made any other undesired changes.

Stakeholder Feedback

The stakeholders liked the look of the project so far, but there were some issues. First was the issue of the buttons not having collisions to cover the entire image of each button due to their parallelogram shape, which I will have to remedy by redesigning the buttons to be rectangular instead. They also reiterated that a camera zoom and movement would be a priority for them as it makes the entire simulation much more visible.

Stage 2 – Physics engine and camera movement

Success criteria

REQUIREMENT	DESCRIPTION	SUCCESS CRITERIA	CHECK
1E	Simulation speed controls (E)	Speed of every object in the simulation can be sped up or stopped while keeping physics consistent.	✓
4	Realistic Physics	Objects in the game behave as in real life, calculations used to determine forces must be accurate.	✓

4B	Planets exert gravitational force on each other	Planets' gravitational force on each other is considered when determining movement.	✓
5	Multiple Planets	Multiple planets and stars can be represented in the software at the same time.	✓
8	Camera movement	Camera is able to be moved in cardinal directions while physics remains consistent.	✓
8A	Camera zoom (E)	Camera can be zoomed in and out, with physics remaining consistent.	~
8B	Camera focus on objects (E)	Camera tracks and follows an object in motion, keeping it in the centre of the screen.	✓

Testing plan

<i>Test #</i>	<i>Test description</i>	<i>Expected outcome</i>
1a	calcGravBetweenObjects(): single object of mass 1	0 force calculated. No movement
1b	Two objects of mass 1 in same position	0 force calculated. No movement
1c	Two objects of mass 1 in different positions	Equal force calculated for both. Both move in a straight line towards one another before meeting at the midpoint between them.
1d	Two objects in different positions; one mass 1, other mass 20	Mass 1 planet moves towards mass 20 planet. Mass 20 planet very slowly moves towards mass 1 planet, before meeting at 1/20 th of the distance between them.
2a	Zoom function: slow zoom in and then out to maximum. 6 objects in ring around centre, all keep consistent physics and spacing.	All objects on screen move into the centre with zoom keeping consistency.
2b	Move slider to maximum value.	Value returned between 4.9 and 5.
2c	Move slider to minimum value.	Value returned between 0 and 0.1.
3	Zoom/focus functions: Focus on each planet in turn while running. Zoom in and out.	Planets and physics behave as predicted and consistently.
4	Time control function: Run the same scenario at two separate time settings.	Two objects take the exact same path on the two different time scale settings.

Development

To start, I created a new file called engine.py. I will use the `import engine` line in my main program file to utilise it. Inside the new file, imported the Vector2 function from Pygame then set up a class, `PhysicsEngine`, with the framework for the constructor, the update function and the function to calculate gravitational force between objects.

Inside the update function, I wrote code to take the group of planet sprites passed through as a parameter from the main program and then list through each pair to run them both through the

gravitational calculation function. The distance between the two sprites is calculated and passed into the gravitational force equation. The result is then returned to the `applyForce()` method on each object.

```

8     def update(self, objGroup):
9
10    for i in range(0, len(objGroup.sprites())): #for each sprite in group:
11
12        j = -1
13        while j < i: #for each other sprite:
14
15            self.calcGravBetweenObjects(objGroup.sprites()[i], objGroup.sprites()[j])
16            j += 1
17

```

In the calculate force function, I applied the equation $F = \frac{Gm_1m_2}{r^2}$ to calculate the force of gravitational attraction between the two objects. I also discovered that using $e-11$ instead of 10^{**-11} was a better way of writing exponentials.

```

18    def calcGravBetweenObjects(self, objA, objB):
19        #Force = G(mass1 * mass2)/radius^2
20        #G = 6.67*10^-11
21
22        radius = objA.gamePos.distance_to(objB.gamePos) #gets radius for calculation -----
23
24        if radius == 0: #if radius 0:
25            return Vector2(0,0) #set vector to 0 (prevents divide by 0 in next step)
26
27        force = 6.67e-11 * objA.correctedMass * objB.correctedMass/(radius*radius) #physics equation
28
29        print(force) #for debug
30
31        vect = objA.gamePos.move_towards(objB.gamePos, force) #creates vector to move object by
32
33        return vect
34

```

To apply this on the planet itself, we must go back to the planet class in the main file. After discovering the `move_towards()` function, I changed the `applyForce()` method so that a parameter was a vector (containing direction) instead of the list I had before, which was unnecessary. I divided this by the mass according to the equation $F = ma$ to find what value I should change the velocity by.

<pre> 22 def applyForce(self, vect): 23 #F = ma 24 #a = f/m 25 #v = at (t is 1 in this situation) 26 27 acc = vect / self.mass 28 29 self.velocity += acc 30 </pre>	$\text{Force} = \text{mass} * \text{acceleration}$ $\Rightarrow \text{Acceleration} = \text{force} / \text{mass}$ $\text{Velocity} = \text{acceleration} * \text{time}$ (as I am calculating this per tick, effectively $\text{velocity} = \text{acceleration}$)
---	---

It was at this point that I realized that to get the acceleration, I could just not multiply by the mass of the object that is being calculated for in the `calcGravBetweenObjects()` function, as I am just dividing by it in the `applyForce()` function anyway. I delete line 27 in the main file (shown above), change 'acc' in line 29 to 'vect', and change line 31 in the engine file:

```

31           force = 6.67e-11 * objB.mass/(radius*radius) #physics equation

```

This will save two calculations per object in the physics engine calculations and should have some optimising effect.

I also realized that the `applyForce()` function itself was redundant, as in the physics engine I could instead just directly change the velocity. I deleted `applyForce()` and changed `calcGravBetweenObjects()`:

```

18     def calcGravBetweenObjects(self, objA, objB):
19         #Force = G(mass1 * mass2)/radius^2
20         #G = 6.67*10^-11
21
22         radius = objA.pos.distance_to(objB.pos) #gets radius for calculation - increases this number as the other numbers have also been increased.
23
24         if radius == 0: #if radius 0:
25             return Vector2(0,0) #set vector to 0 (prevents divide by 0 in next step)
26
27         force = 6.67e-11 * objB.mass/(radius*radius) #physics equation
28
29         print(force) #for debug
30
31         vect = objA.pos.move_towards(objB.pos, force) #creates vector to move object by
32
33         #f = ma
34         #a = f/m
35         #v = at (t is 1 in this situation)
36         objA.velocity += vect
37

```

In order for the speed controls to be able to be implemented later, I will multiply the velocity in the planet object by the engine's time scaling factor. This value will be set as 0 at program start in the `__init__()` method for the engine.

```

40     def effect(self):
41
42         match self.designation:
43
44             case 0: #time control button
45
46                 if physEng.timeScale == 1:
47                     physEng.timeScale = 0
48                 else:
49                     physEng.timeScale += 1
50
51             case _:
52                 print("ERROR: BUTTON DESIGNATION NOT RECOGNISED")
53

```

to make it easy for me to add varying degrees of time speedup, just by changing the maximum value (in the if statement) and the increment value. I could also theoretically make this produce an anti-gravity effect by letting the time scale go to a negative number, but this is outside the scope of the project.

I ran the program to see what the effects of my gravity calculations were. At program start, the two test objects I instantiated were not moving due to the time scale being set to pause, but after clicking the speed control button it switched state before the planets shot off of the side of the screen moments before I received this error message as the program crashed:

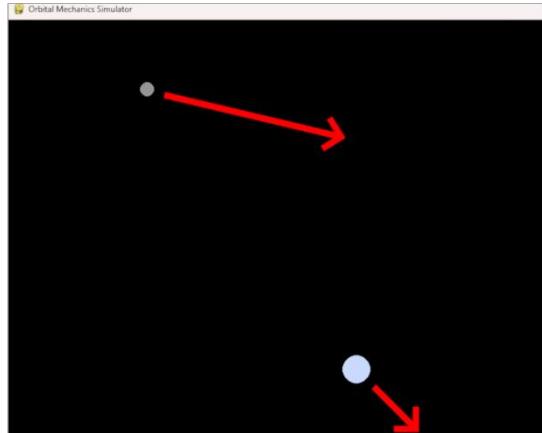
```

Traceback (most recent call last):
  File "e:\NEA (USB)\proj\main.py", line 229, in <module>
    physObjects.update() #apply results to objects
    ~~~~~~
  File "C:\Users\chick\AppData\Roaming\Python\Python311\site-packages\pygame\sprite.py", line 555, in update
    sprite.update(*args, **kwargs)
  File "e:\NEA (USB)\proj\main.py", line 130, in update
    self.rect = self.image.get_rect(center=self.screenPos) #update rect, centred at the planet's position
    ~~~~~~
TypeError: invalid rect assignment, expected value between -2147483648 < x < 2147483647
PS E:\NEA (USB)\proj> 

```

The force calculated by the engine was too much for this scale; the objects were going outside of the range of rects functioning. I looked back at my equations and calculations and found that while I had increased the mass of the objects by a factor of 10^{24} , I had neglected to change the distance

between the objects for the purpose of the calculation. As I had increased the radius of the planet and star objects by a factor of 10^6 , I tried increasing this distance by the same amount. However, this did not seem to have any effect. Printing the value for the force calculated for each planet as well as their positions on screen, I found that although a small force was being applied, which was decreasing with distance, the planets would still be flung very far in a short amount of time. I was also concerned with the fact that both planets would fly into the positive coordinate direction; as they are both the same mass, and have no initial velocity, they should meet in the middle of where they are positioned at the start. I suspect that the Pygame `move_towards()` method is to blame for this.



To test this further, I slowed the time scale to 0.0001 and ran the program. This time, the planets moved slowly enough that I could see where they were moving.

The red arrows show the direction of movement of the planets. Both are moving in the complete wrong direction. As the documentation on the Pygame `move_towards()` method is sparse, I decided that it would be simpler for me to create my own version of this function so that I understood it fully. This should be relatively simple as it will just involve finding the unit vector between the two objects.

After doing some maths, I found that the calculation to create this vector was

$$\left[\frac{B_x - A_x}{\sqrt{(B_x - A_x)^2 + (B_y - A_y)^2}}, \frac{B_y - A_y}{\sqrt{(B_x - A_x)^2 + (B_y - A_y)^2}} \right], \text{ where } B_x \text{ and } A_x \text{ are the gamePos X coordinates of object}$$

B and A respectively, and B_y and A_y are the same for the Y gamePos coordinates. This was implemented in the `calcGravBetweenObjects()` method to replace the `vect = objA.movetowards(...)` line.

```
vect = Vector2((objB.gamePos[0]-objA.gamePos[0])/math.sqrt((objB.gamePos[0]-objA.gamePos[0])**2+(objB.gamePos[1]-objA.gamePos[1])**2),  
(objB.gamePos[1]-objA.gamePos[1])/math.sqrt((objB.gamePos[0]-objA.gamePos[0])**2+(objB.gamePos[1]-objA.gamePos[1])**2)) * force #create
```

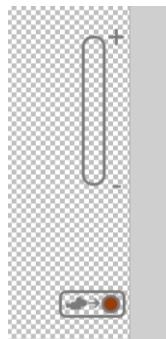
The code is one line but is very long, so for readability here I have split it across these two lines.

I then ran the program to test, and both objects began to slowly accelerate towards each other, as expected. When the objects came into contact in the middle, the force being added to their velocity kept increasing until they flung each other in the opposite direction. This is accurate in terms of calculations; the fix for this is to implement collisions for the objects, but this is an extra feature for a later stage. I added some sideways velocity to one of the objects and reset the program, to ensure they did not pass through one another again, and the objects both had an equal gravitational force on one another and would change the direction of the force according to the distance and angle between the two objects.

Sometimes, the physics would be inconsistent within the program even when no code was changed. This happened when I paused the simulation, as while it was paused the calculations for the force on the objects was still being applied and changing their velocities. To fix this, I made sure that the `calcGravBetweenObjects()` method would not be called in the physics engine update method if the time scale value was 0.

I then added a third object, and the simulation behaved as required. However, the objects tended to eventually move out of the bounds of the screen; to fix this, a camera system is needed to be able to follow the objects.

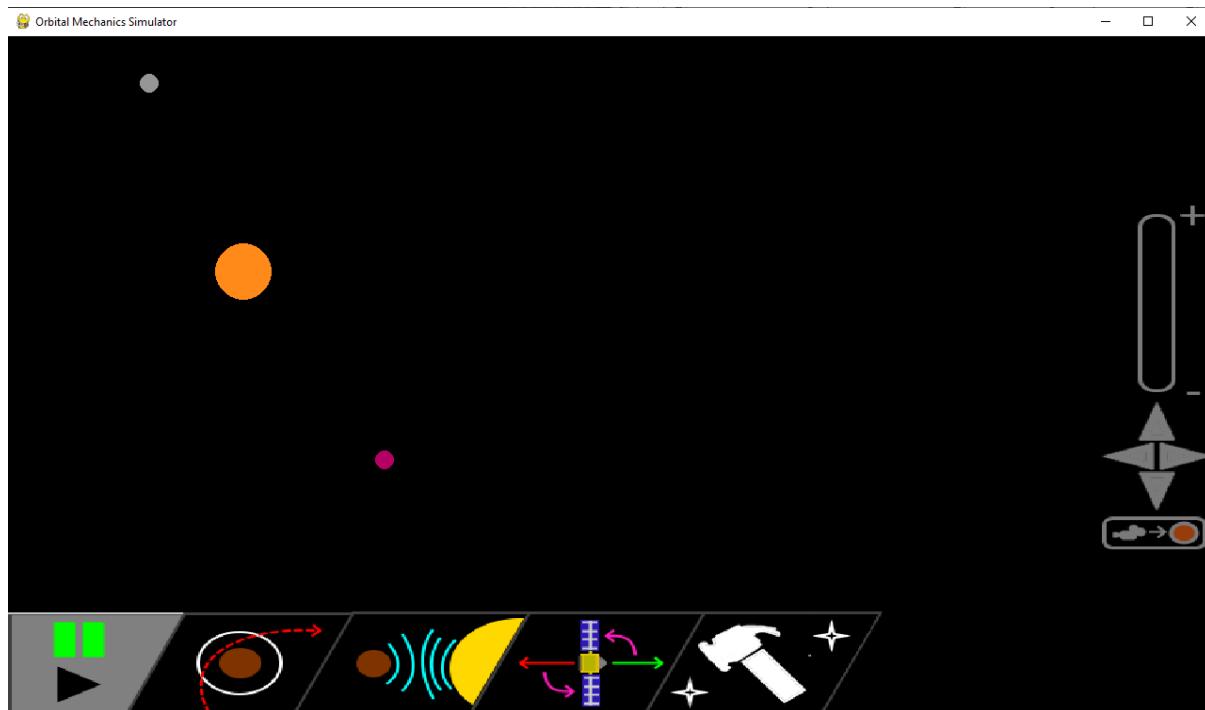
To begin, I need to create four more buttons in order to control the direction in which the camera would move. The arrow keys on the keyboard will also be available for this purpose.



For the buttons, again created in paint.net for this version, I made four grey triangles as each cardinal direction for controlling the camera with a mouse press. For the camera focus button, I decided to merge it with the static background for the slider object for the camera zoom, as this was meant to just be pasted on the background and have no effect. As I am doing collisions for the buttons independently, I can position the collision rect above the actual button instead of over the button and the slider background. Shown on the left is the image for this focus button with the slider above. Subject to change depending on stakeholder feedback, and in a final version I will need to create an image with a higher quality.

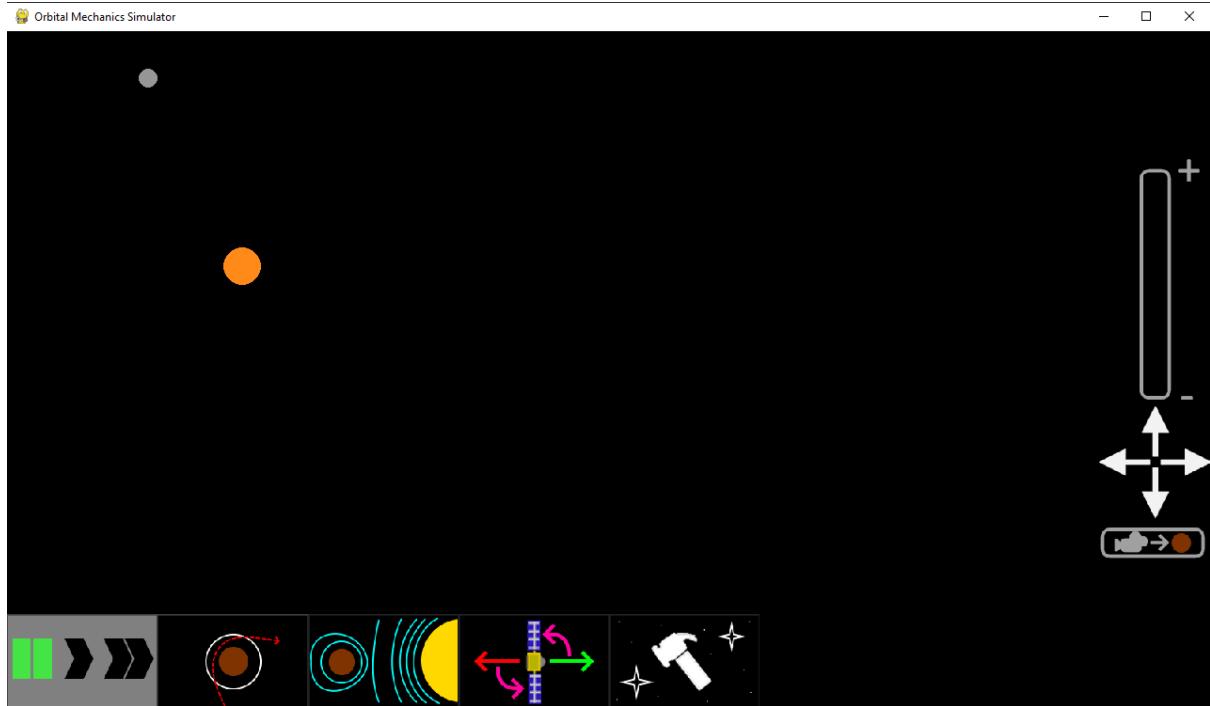
These images were named in file as `UIButton(designation)`, with designation being the numbers 5-9 for the designation in the program of each button. Each button also has an image for `UIButtonClicked(designation)` to provide visual feedback on press.

In order to display these buttons on screen, I only have to change the range of the loop initialising the buttons from 5 to 10.



The buttons look a bit squished, and I think that is because I messed up with the size of the screen when creating them in paint.net. I will redraw them, as well as experimenting with increasing the pixel resolution of the drawing in the hopes that it will improve in quality. I will also take this opportunity to redraw the main menu buttons to be square, as the stakeholders have requested. The new images will all be drawn on a 1600x900 canvas to fit with the default resolution of this software.

The time control button currently has only two options associated with it; play and pause. I would like a third option, to speed up the simulation, when the button is clicked for a second time. In order to accommodate for these three states, I will need to check if the designation of the button is 0 in the button initialisation and if so, load the third state image. In the update method I will also need to check the designation to change the state accordingly. As the case statement for the button effects is called through the update method, I can move the code relating to case 0 into the update method to be run when the designation is 0.



This updated GUI has rectangular buttons to perfectly fit the collision boxes, and the camera movement controls have changed in design to be more intuitive.

I changed the button collision rects to fit the main buttons perfectly, and then wrote the code to create the collision rects for the new buttons. While the main buttons could have their collision rects positioned based on their designation and their position on the screen, I will have to set each new collision box individually, as the buttons are not positioned in a straight line. Due to the nature of the camera movement control buttons, however, I may be able to position them in pairs.

```
def rescale(self):
    self.image = pygame.transform.scale(self.imageUnscaled,(screen.get_width(),screen.get_height())) #rescale to window size

    if self.designation < 5: #for main menu buttons only:
        self.collideRect.update((screen.get_width()/8) * self.designation, screen.get_height() - (screen.get_height()*(5/36)),
                               (screen.get_width()/8), (screen.get_height()*(5/36))) #update rectangular coordinates for mouse cursor detection

    elif self.designation == 6 or self.designation == 7:
        self.collideRect.update(screen.get_width() - (screen.get_width()* 0.05625), (screen.get_height() *0.079) * (self.designation + 1),
                               (screen.get_width()*0.0225), (screen.get_height() *0.08)) #update rectangular coordinates for mouse cursor detection

    elif self.designation == 8 or self.designation == 9:
        self.collideRect.update(screen.get_width() - (screen.get_width()* 0.0425 * (self.designation - 7)), (screen.get_height() *0.6156),
                               (screen.get_width()*0.0425), (screen.get_height() *0.0411)) #update rectangular coordinates for mouse cursor detection
```

This updated `rescale()` method checks the designation of the buttons and positions them as such.

```
def updateCamera(self, direction, objGroup): #function to produce uniform movement

    match direction:

        case "N": #north
            for i in range(len(objGroup.sprites())): #for each physics object:
                objGroup.sprites()[i].gamePos[1] -= 50

        case "S": #south
            for i in range(len(objGroup.sprites())): #for each physics object:
                objGroup.sprites()[i].gamePos[1] += 50

        case "E": #east
            for i in range(len(objGroup.sprites())): #for each physics object:
                objGroup.sprites()[i].gamePos[0] -= 50

        case "W": #west
            for i in range(len(objGroup.sprites())): #for each physics object:
                objGroup.sprites()[i].gamePos[0] += 50
```

In the physics engine class, I create the updateCamera() method to provide the functionality of the buttons. This takes a direction input (N, S, E or W) corresponding to each of the direction buttons and translates this into movement on the screen by applying an offset to each physics object.

Testing the program now, the buttons worked in changing the position of the two test objects on the screen, without affecting any physics effects. However, as with the main menu buttons, the ‘clicked’ state would stay until the button was clicked again, and if the mouse button was held down then the movement effect would only be applied once.

To fix this, I will have to create a function that circumvents the normal button update method for these specific buttons; this new mouseHoldEnd() function will be called on the Pygame event for unclicking the mouse button. In the method, the state of the affected buttons will return to normal.

```
def mouseHoldEnd(self):

    if self.designation >= 6 and self.designation <= 9: #if a camera movement button:

        self.imageUnscaled = self.imageUnclicked #change graphic
        self.isClicked = False #set logic to be in unclicked state

        self.rescale() #update button image
```

```
case 6: #camera south
    while self.isClicked:
        physEng.updateCamera("S", physObjects)

case 7: #camera north
    while self.isClicked:
        physEng.updateCamera("N", physObjects)

case 8: #camera east
    while self.isClicked:
        physEng.updateCamera("E", physObjects)

case 9: #camera west
    while self.isClicked:
        physEng.updateCamera("W", physObjects)
```

To make the buttons apply continuously when held down, I only need to create a while loop with the self.isClicked variable as the condition inside the button checks.

When I ran this to test, the program crashed when I clicked any of the camera movement buttons. This must be due to the program getting caught in an infinite loop, or the buttons calling the updateCamera() method too frequently.

To fix this issue, instead of using a while loop, I will call another new method, mouseHoldUpdate(), every tick in the main running loop. This checks if the self.isClicked value of the respective button is enabled and if that button is one that should repeat its effect when being held down (using designation) and calls the effect() method if so..

```
def mouseHoldUpdate(self):
    if (self.designation >= 6 and self.designation <= 9) and self.isClicked: #if a camera movement button, and is currently active:
        self.effect() #apply movement effect
```

Testing this now, the objects present in the scenario all move smoothly, the buttons function as expected and a high performance is maintained. I then implemented the keyboard arrow key controls for this feature, by checking for the inputs in the Pygame events system and calling the corresponding methods.

On testing this, the movement is not continuous when you hold down the arrow key, unlike with the on-screen button. This is due to the Pygame events system only firing once on each event, so instead I will check for key presses outside of the events system.

```
#####CAMERA MOVEMENT INPUTS#####
if pygame.key.get_pressed()[pygame.K_DOWN]: #down arrow key
    physEng.updateCamera("N", physObjects)

elif pygame.key.get_pressed()[pygame.K_UP]: #up arrow key
    physEng.updateCamera("S", physObjects)

elif pygame.key.get_pressed()[pygame.K_RIGHT]: #down arrow key
    physEng.updateCamera("E", physObjects)

elif pygame.key.get_pressed()[pygame.K_LEFT]: #down arrow key
    physEng.updateCamera("W", physObjects)
```

This code is present in the main running loop, just after the events system is checked and just before the screen is rendered.

After testing this solution, the arrow keys on the keyboard now work identically to the buttons in the program.

Now I will work on the camera focus feature. The first thing to do is set up the collision rect for the focus button. I realised while looking at the rescale function that it was much more readable to write it as another match...case statement.

```
def rescale(self):
    self.image = pygame.transform.scale(self.imageUnscaled,(screen.get_width(),screen.get_height())) #rescale to window size
    match self.designation:
        case 1 | 2 | 3 | 4 : #main menu buttons (not including time control):
            self.collideRect.update((screen.get_width()/8) * self.designation, screen.get_height() - (screen.get_height()*(5/36)), (screen.get_width()/8), (screen.get_height()*(5/36))) #update rectangle
        case 5: #camera focus button:
            pass
        case 6 | 7: #north/south camera movement buttons:
            self.collideRect.update(screen.get_width() - (screen.get_width())* 0.05625, (screen.get_height())*0.079) * (self.designation + 1), (screen.get_width())*0.0225, (screen.get_height())*0.08) #update rectangle
        case 8 | 9: #east/west camera movement buttons:
            self.collideRect.update(screen.get_width() - (screen.get_width())* 0.0425 * (self.designation - 7), (screen.get_height())*0.6156, (screen.get_width())*0.0425, (screen.get_height())*0.0411))
```

After positioning the rect correctly, I have also added it to the statement in `mouseHoldEnd()` by changing the lower bound on the designation to 5. This is because, like the other camera movement buttons, it is not a toggle. Unlike the other buttons however, this one's effects will not be repeated as it is held down, so it will not be added to the `mouseHoldUpdate()` method.

Next, I will make a way to cycle through each physics object each time the focus button is clicked. This is present in the button `effect()` method, and loops through one object in the list of physics objects each time when pressed.

```
case 5: #camera focus

    self.objectNumber += 1 #increment object to focus on

    if self.objectNumber == len(physObjects.sprites()): #if reached end of list:
        self.objectNumber = -1 #Loop back
        physEng.updateCamera(0, physObjects) #set to 0 (no object focused on)
    else:
        physEng.updateCamera(physObjects.sprites()[i], physObjects)
```

```
match direction:
    case "N": #north
        for i in range(len(objGroup.sprites())): #for each physics object:
            objGroup.sprites()[i].gamePos[1] -= 5

    case "S": #south
        for i in range(len(objGroup.sprites())): #for each physics object:
            objGroup.sprites()[i].gamePos[1] += 5

    case "E": #east
        for i in range(len(objGroup.sprites())): #for each physics object:
            objGroup.sprites()[i].gamePos[0] -= 5

    case "W": #west
        for i in range(len(objGroup.sprites())): #for each physics object:
            objGroup.sprites()[i].gamePos[0] += 5

    case _: #object to focus on passed through
        self.focusedObject = direction
```

These physics objects will be passed through the updateCamera() method as the direction parameter; the match...case statement is set up to set any value of the direction other than the four cardinal directions as the object to be focused on by the camera.

To actually give the effect of following the requested object, we will need to apply a force to that object to make it stay in place at the centre of the screen. Consequently, we will also need to apply this force to all other objects in the scenario. This will be done by taking the velocity of the focused object and subtracting it from the velocity of every object in the scenario, including itself.

```
def cameraFocus(self):
    if self.focusedObject is not None: #if there is an object to be focused on
        return self.focusedObject.velocity * self.timeScale
    else:
        return 0
```

To do this, we will take the velocity of the focused object and apply it to the screen position variable of each physics object to offset it. This will be calculated in the form of the cameraFocus() function, called in

the planet class when drawing to the screen, which will return the velocity of any focused object, or 0 if there is none.

I have gone a while without testing this, so let's see if it will actually work. On pressing the focus button for the first time, I am greeted with this:

```
Traceback (most recent call last):
  File "e:\NEA (USB)\proj\main.py", line 305, in <module>
    buttonGroup.update() #run checks for buttons
    ^^^^^^^^^^^^^^^^^^

  File "C:\Users\chick\AppData\Roaming\Python\Python311\site-packages\pygame\sprite.py", line 555, in update
    sprite.update(*args, **kwargs)
  File "e:\NEA (USB)\proj\main.py", line 150, in update
    self.effect() #apply effect of button
    ^^^^^^^

  File "e:\NEA (USB)\proj\main.py", line 89, in effect
    physEng.updateCamera(physObjects.sprites()[i], physObjects)
    ^^^^^^^^^^^^^^

TypeError: list indices must be integers or slices, not Button
PS E:\NEA (USB)\proj>
```

I have used i instead of self.objectNumber. Testing again, the button pressed without issue, so I tried to unpause the simulation to see if the camera would track an object. However, while everything

else seemed to be working, the time control button had no response when clicked. After putting some debug print statements in to see what was being executed, I found that the button wasn't registering when it was being clicked, leading me to suspect an issue with its collision rect.

I then found that when I switched the button rect updates in the rescale() method, I forgot to include designation 0 in the menu button case. With this fixed, I can get back to testing the focus feature.

Nothing really seemed to change when running, except that the physics got slightly inconsistent. I put in a print statement to tell me the colour of the object the camera was focused on so that I knew which one to look at, and what was supposed to be happening with it. The test objects I was using were a small grey planet and a larger orange star.

I set the focus on the star and unpause the scenario. Instead of staying stationary as expected, it began to move with a greater velocity than normal towards the planet. I rechecked my code for applying the velocity of the planet back to itself to cancel it out, and there seemed to be no logical errors, so I printed out the velocities of the star as given by the physics engine and as given by the star itself, just before they are cancelled out. I am expecting these to both be the same value.

```
SELFVEL: (150, 150, 150): [1.55849, 0.538011]
ENGVEL: [0.0557539, -0.269005]
SELFVEL: (255, 138, 26): [0.0576943, -0.2734]
ENGVEL: [0.0576943, -0.2734]
SELFVEL: (150, 150, 150): [1.55461, 0.546799]
ENGVEL: [0.0576943, -0.2734]
PS E:\NEA (USB)\proj> █
```

As shown here, the velocity of the orange star (255, 138, 26) is always equal to the velocity of the star according to the engine, so there is no disconnect there.

The only other operation applied to the velocity before it is applied to the game and screen positions is its multiplication by the time scale factor.

```
self.gamePos += self.velocity * physEng.timeScale #move sprite
self.screenPos = self.gamePos - physEng.cameraFocus() #set position on screen to position in terms of game, and offset any focused object
pygame.draw.circle(screen, self.colour, self.screenPos, self.radius) #draw planet
```

As cameraFocus() just returns the velocity of the focused object multiplied by the time scale again, and the velocities in the physics engine and the star object are the same, these values should cancel out and produce no change in the screenPos variable. I ran the program again, this time printing the result of (velocity * timeScale – cameraFocus) from the physics objects – I am expecting half of these to say 0.

```
[0, 0]
[-0.263463, 0.996952]
[0, 0]
[-0.273756, 0.995863]
PS E:\NEA (USB)\proj> █
```

The function and offset is working as intended; so where is this change in the screenPos variable coming from if the gamePos variable is always constant for the star? I ran the program again, printing the screenPos, which showed that it was still changing.

Then, I moved the subtraction of the cameraFocus value up to where the gamePos was incremented in the planet update() method and ran the program. This produced the desired effect of no movement on the focused object. This made no sense. I was completing the same operation in the next line down, with a completely different effect.

To check if this was any kind of rounding error, I subtracted the velocity and time scale directly in the (screenPos = ...) line to see if the desired effect was achieved. It was not. Adding brackets around each expression also made no difference.

<pre>self.gamePos += self.velocity * physEng.timeScale - physEng.cameraFocus() #move sprite self.screenPos = self.gamePos #- physEng.cameraFocus() #set position on screen to position pygame.draw.circle(screen, self.colour, self.screenPos, self.radius) #draw planet</pre>	Working (but camera movement affects gamePos)
<pre>self.gamePos += self.velocity * physEng.timeScale #move sprite self.screenPos = self.gamePos - physEng.cameraFocus() #set position on screen to position pygame.draw.circle(screen, self.colour, self.screenPos, self.radius) #draw planet</pre>	Not working
<pre>self.gamePos += self.velocity * physEng.timeScale #move sprite self.screenPos = self.gamePos - self.velocity * physEng.timeScale #physEng.cameraFocus() pygame.draw.circle(screen, self.colour, self.screenPos, self.radius) #draw planet</pre>	Not working

I could not understand the problem here, but the gamePos direct change did not seem to affect anything else. I would like to avoid offsetting the gamePos variable here if at all possible, as it may affect the functionality of the camera zoom if implemented. However, in the absence of any other current solution, I will work on more features and fix this if a problem arises in the future.

Another part of the camera focus system is moving the object in focus to the centre of the screen. To do this, I will offset every physics object in the scenario when an object is first focused on in such a way that the focused object is at the centre; this will be done in much the same way as the normal camera movement, by directly changing the gamePos of all physics objects.

```
case _: #object to focus on passed through
    self.focusedObject = direction

    #move focused object to centre of screen
    for i in range(len(objGroup.sprites())):
        objGroup.sprites()[i].gamePos -= self.focusedObject.screenPos - Vector2(screen.get_width()/2, screen.get_height()/2)
        #get distance from object to centre of screen. offset by value needed to centre focused object
```

```
def updateCamera(self, direction, objGroup, screen):
    self.focusedObject = 0 #clear object focus

    match direction:

        case 0: #no direction passed
            return #focus is cleared.
```

This is part of the cameraUpdate() method in the physics engine. Also at the beginning of this method, before the match...case statement, I will clear the self.focusedObject as it is set later on if the focus button calls this method, and I would like the camera movement buttons to break out of the focus feature.

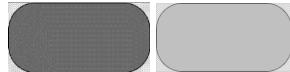
I also set a case 0 for this method as when the focus button loops through the objects, it can pass through a 0 value for the direction to clear the focused object in the physics engine.

Testing this now, only the object being focused on moves to the centre of the screen. After much staring at the code, I decided to create a buffer variable for generating the vector to move the objects by when initially focusing, called focusBuffer. This, somehow, fixed the issue; I suspect some code must be executed at the same time as others, and this discrepancy made the variables go funny.

Now the focus feature is complete, I will work on the slider class to implement the camera zoom function. I will make the image of the slider button not the size of the entire screen but instead only the size of the clickable button itself. This allows me to get the collision rect just by using rect = image.get_rect().

```
class Slider(pygame.sprite.Sprite):  
  
    def __init__(self):  
  
        pygame.sprite.Sprite.__init__(self) #initialise Pygame sprite  
  
        self.clicked = False #set initial click state  
  
        self.image = pygame.image.load("sliderButton.png").convert_alpha() #load image  
        self.rect = self.image.get_rect() #get rect from image  
  
  
    def update(self):  
  
        if self.rect.collidepoint(pygame.mouse.get_pos()): #if mouse cursor is intersecting collision rect:  
            #TODO  
            pass
```

I began to realise that I would also need `mouseHoldUpdate()` and `mouseHoldEnd()` methods, and that to draw the object I would need to add it to its own group – all of these factors led me to realise that I could just create the slider as another instance of the button object, with designation 20 (10-19 were missed out in so that I could implement more non-slider buttons).



I drew the images for the buttons in paint.net and named them with the designation 20 for the button class to register them.

One difference with these buttons as opposed to the other buttons added previously is that they must move up and down and will not stay in one place. The way I scaled all of the buttons up to this point is by drawing them all in the position they would be on the screen in a full image, but I will have to draw these buttons in a different way so that they can move on the screen.

Consequently, I will have to scale these buttons in a different way to all the previous ones. In the `rescale()` method, I wrote in a check if the buttons' designation was 20 or above (a slider) and if so, scaled the buttons as such.

```
if self.designation < 20: #if not slider button:  
  
    self.image = pygame.transform.scale(self.imageUnscaled,(screen.get_width(),screen.get_height())) #rescale to window size  
  
else: #if slider button:  
  
    self.image = pygame.transform.scale(self.imageUnscaled,(screen.get_width()/22,screen.get_height()/22)) #rescale to window size
```



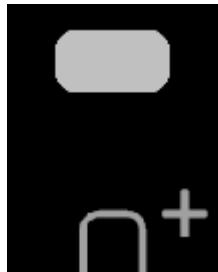
Orbital Mechanics Simulator



The button is being created and scaled successfully, but now I have to position the button in the same place on the screen.

To do this, I will move the main rect of the sprite that is blit to the screen on draw. This takes place in the `rescale()` method. I set the collide rect as the same, as the size of the rect is the same as the size of the entire image.

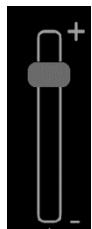
```
case 20: #zoom slider:
    self.rect.update(screen.get_width() - (screen.get_width()* 0.069), (screen.get_height()*0.079), (screen.get_width()/22), (screen.get_height()/22))
    self.collideRect = self.rect
```



The button is positioned above the slider and responds when pressed in the correct area. Now, to make it move down, I will create a zoomFactor variable and compare it against the scaled Y value of the screen to determine the button's position. The rescale() method for changing the Y axis value must be changed to do this; while the position of previous buttons on the screen used a multiplier of the designation to position them easily (as with the main menu buttons), this slider will use the value of the zoomFactor as its positioning multiplier. The maximum zoom value I would like this program to have is 5x with the minimum being 0.5x, so with an initial zoom value of 1x, the slider needs to be positioned 1/5th of the way down the slider bar at program start.

```
case 20: #zoom slider:
    #Slider top buffer: height * 0.2
    #Slider area: (height * 0.5 - height * 0.2) = height * 0.3
    #zoom factor max. = 5x
    #0.3 / 5 = 0.06

    self.rect.update(screen.get_width() - (screen.get_width()* 0.069), (screen.get_height())* 0.2 + physEng.zoomScale * 0.06 * screen.get_height()),
    self.collideRect = self.rect #create collision rect
```



To calculate the initial position, I took the height of the screen above the slider area and added the proportion of the zoom factor over the area of the slider. This produces a scalable slider button that is 1/5th of the way down the slider. However, there is a logical error here: the slider button is at the wrong end of the slider. To fix this, I subtracted the zoom value positioning from the slider height before the buffer addition.

At this point, I realised that in the rescale() method, I was constantly calling the screen.get_height() and get_width() methods, which could affect performance. To remedy this, I created two new local variables, screenHeight and screenWidth, which would be set at the start of the rescale() function. This also made the code much more readable.

```
case 0 | 1 | 2 | 3 | 4 : #main menu buttons (not including time control):
    self.collideRect.update((screenWidth/8) * self.designation, screenHeight*(31/36), screenWidth/8, screenHeight*(5/36)) #update rectangular coordinates for mouse

case 5: #camera focus button:
    self.collideRect.update(screenWidth*0.9081, screenHeight*0.7333, screenWidth*0.0869, screenHeight*0.0422) #update rectangular coordinates for mouse

case 6 | 7: #north/south camera movement buttons:
    self.collideRect.update(screenWidth*0.94375, screenHeight * 0.079 * (self.designation + 1), screenWidth*0.0225, screenHeight*0.08) #update rectangular coordinates for mouse

case 8 | 9: #east/west camera movement buttons:
    self.collideRect.update(screenWidth - screenWidth * 0.0425 * (self.designation - 7), screenHeight*0.6156, screenWidth*0.0425, screenHeight*0.0411)
```

In the effect() method, I will set the Y position of the button to that of the mouse cursor, and then use this position to determine what the zoomFactor should become. To find the position of the slider, I can solve the calculation used to find the slider position based on the zoom value of 1 when I initially positioned the button.

This gives $Z = \frac{0.5H-P}{0.06H}$, where H is the screen height and P is the Y value of the slider (in this case, the Y value of the cursor).

```
case 20: #zoom slider

    screenWidth = screen.get_width()
    screenHeight = screen.get_height()

    self.rect.update(screenWidth * 0.931, pygame.mouse.get_pos()[1], screenWidth/22, screenHeight/22)

    #change zoom value based on Y position:
    physEng.zoomScale = (0.5 * screenHeight - pygame.mouse.get_pos()[1])/(0.06 * screenHeight)
    print(physEng.zoomScale)
```

The print statement there is for testing purposes. I ran the program and moved the slider between both ranges to check this.

```
3.4490740740740744
3.4953703703703707
3.5416666666666667
3.5648148148148153
3.5648148148148153
3.5648148148148153
3.5648148148148153
```

The slider moves smoothly, and the zoom value changes as expected, so I am happy that the slider system is working to produce the intended value.
The only thing left to do with the slider is to clamp it beyond the end of the slider bar area so that it cannot move or return a value past these points. The simplest way to achieve this is by including a check before any of the code in the previous image is applied, so that code is not run if the mouse cursor is above or below a certain value.

```
if screenHeight * 0.5 > pygame.mouse.get_pos()[1] > screenHeight * 0.2: #checks if mouse cursor is within bounds of the slider area:
```

Testing this, the slider clamps at the correct values. If I quickly move the cursor outside the bounds of the slider area, the slider may stop but it will never move into the wrong position.

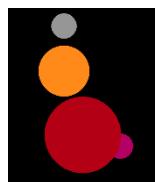
One other thing that I have noticed while using this system is that when the slider is pressed, its position jumps so that the mouse cursor is at the top of the button. To fix this, I need to offset it by half of the button's size.

After doing this, however, the checks for button position and the conversion for position to zoom value were incorrect. Luckily, it was an easy fix with only the check value and two of the calculation values needing to be increased (0.5 → 0.55, 0.06 → 0.07).

```
0.09920634920635032
1.1278456123176193e-15
0.09920634920635032
0.21746921746921955
```

One concern I did have was the chance that the zoom factor could be set to 0; this could cause some issues later on with scaling and moving the physics objects. In testing, this was the lowest value I observed with the zoom factor. To try and mitigate this, I will increase the check value for the mouse position while keeping all other calculations the same. Setting the value for the check to 0.53, the minimum zoom value should now be 0.3.

I am confident that the slider has been fully implemented; now, I have to apply the zoom scaling factor to all objects.



The first obvious change when an object is zoomed into is that the size of it increases. To achieve this affect, I multiply the radii in the planet draw method by the zoom scale.

This produced the desired effect when the slider is moved.

The second effect when zooming in or out is the movement of objects away from the centre of the screen – objects that are further away will move a greater distance. The challenge with this is to keep all physics consistent.

My first attempt with this was by multiplying the horizontal and vertical components of the gamePos variable by the zoom scale factor when setting the screenPos in the planet update() method.

```
self.gamePos += self.velocity * physEng.timeScale - physEng.cameraFocus() #move sprite, and offset any focused object  
self.screenPos = (self.gamePos + pygame.Vector2((self.gamePos[0] - screen.get_width()/2) * self.zoomScale, (self.gamePos[1] - screen.get_height()/2) * self.zoomScale))
```

This produced a zoom effect that was close to what I wanted, but the distances between the planets would change. However, the physics seemed to remain consistent. I realised that the issue was that I was adding the gamePos for some reason; I removed this. I also had the equation mixed up, so I rearranged it to produce this:

```
self.screenPos = pygame.Vector2(screen.get_width()/2 + self.gamePos[0] * physEng.zoomScale, screen.get_height()/2 + self.gamePos[1] * physEng.zoomScale)
```

This produced the desired effect; all of the objects would hold the same distance from each other when the zoom slider was increased or decreased. One interesting side effect of implementing this system was that the camera is initially positioned at gamePos (0,0), which I have no issue with.

I then tested this system with the camera focus and movement systems as well; while the camera movement system still worked as intended, the focus seemed to be broken. This was likely due to the previous issue of having to do the camera focus offset in the gamePos update.

I tried moving the cameraFocus subtraction to the end of the screenPos calculation line, which moved the camera to the right place when focus button was clicked, however the camera then did not follow the movement of the focused object. When I tried changing the zoom again, the camera jumped to the wrong places.

Checking the focus offset with small zoom increments and with the centre of screen offset disabled, I found that the more you zoom in, the more the camera will overshoot the position of the object. This means that I must factor in the zoom offset when changing the gamePos of all objects when initially focusing, and I suspect while focusing on that object in motion as well.

While the calculations for the focus are correct, I need a way to apply them to the screenPos of physics objects. This is done by adding an initialFocusOffset, calculated by the physics engine to the screenPos when drawing an object, replacing my method of directly changing the gamePos of objects in the physics engine updateCamera() method.

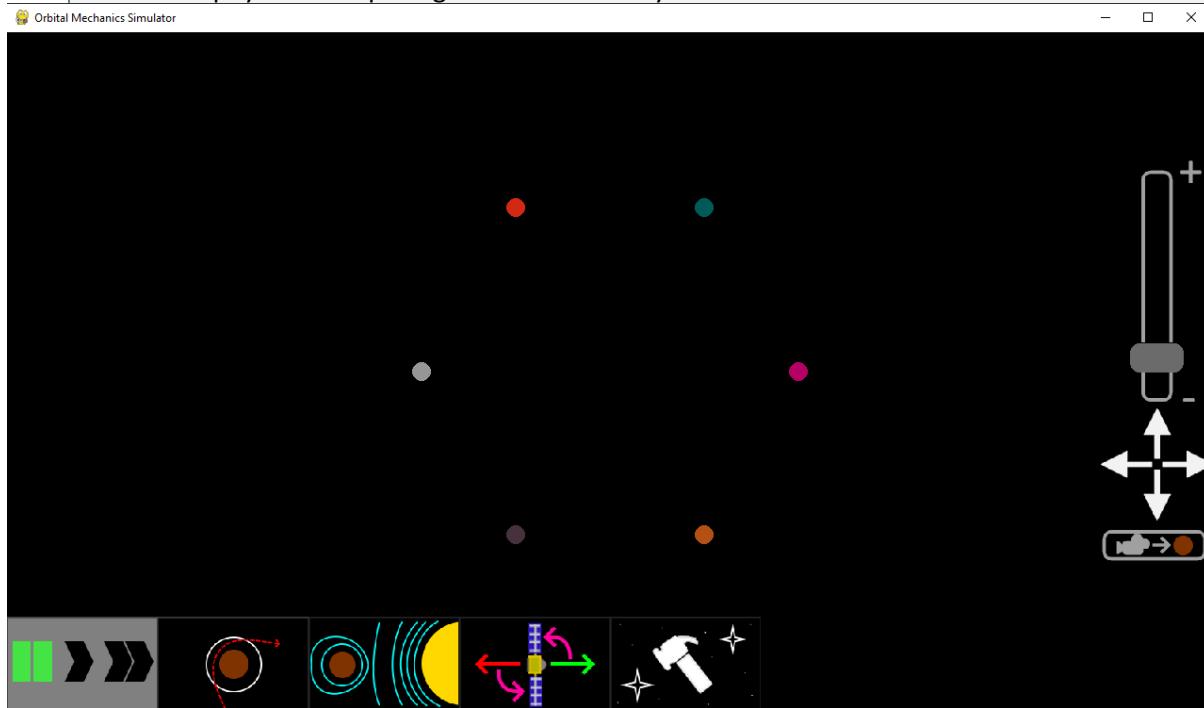
```
case _: #object to focus on passed through  
    self.focusedObject = direction  
  
    self.initialFocusOffset = Vector2(screen.get_width()/2, screen.get_height()/2) - self.focusedObject.screenPos
```

Now the camera initially moves to the correct position, and the focus is kept. Testing while zooming in and out keeps physics consistent, but sometimes the focused object does not stay at the centre of the screen. To fix this, I re-ran the initial offset calculation for the object whenever the zoom is changed. The system also remains consistent when the window is resized, and so I am happy to say that the feature has been implemented fully.

Testing + Bug fixing

Test #	Test description	Expected outcome	Actual outcome	Pass/fail
1a	calcGravBetweenObjects(): single object of mass 1	0 force calculated. No movement	The planet did not move in place.	Pass

1b	Two objects of mass 1 in same position	0 force calculated. No movement	Only one planet displays, but I can see in the output terminal that the position of both planets is the same.	Pass
1c	Two objects of mass 1 in different positions	Equal force calculated for both. Both move in a straight line towards one another before meeting at the midpoint between them.	As expected.	Pass
1d	Two objects in different positions; one mass 1, other mass 20	Mass 1 planet moves towards mass 20 planet. Mass 20 planet very slowly moves towards mass 1 planet, before meeting at $1/20^{\text{th}}$ of the distance between them.	As expected.	Pass
2a	Zoom function: slow zoom in and then out to maximum. 6 objects in ring around centre, all keep consistent physics and spacing	All objects on screen move into the centre with zoom keeping consistency.	As expected.	Pass



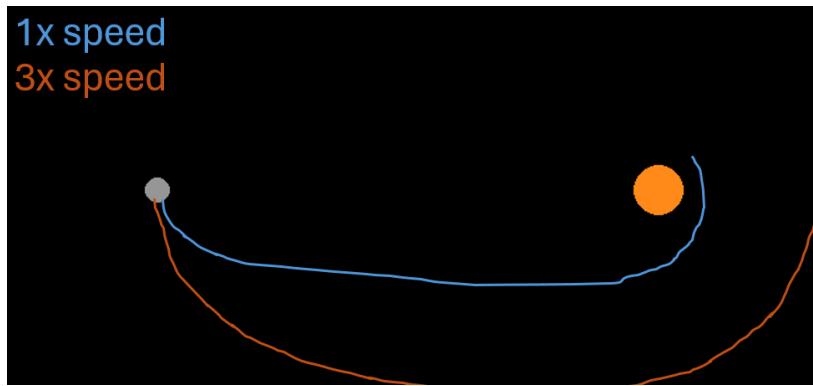
The six planets, all with same mass. All met at the centre of the screen as expected.

Test #	Test description	Expected outcome	Actual outcome	Pass/fail
2b	Move slider to maximum value	Value returned between 4.9 and 5.	4.98015873015873 1	Pass
2c	Move slider to minimum value	Value returned between 0.28 and 0.32.	0.31746031746031 855	Pass – conditions

of test
changed

This test's conditions were changed due to concerns about the effects of setting the zoom to 0. I think that 0.3x zoom is more than small enough for this software.

Test #	Test description	Expected outcome	Actual outcome	Pass/fail
3	Zoom/focus functions: Focus on each planet in turn while running. Zoom in and out	Planets and physics behave as predicted and consistently.	As expected.	Pass
4	Time control function: Run the same scenario at two separate time settings.	Two objects take the exact same path on the two different time scale settings.	Two objects took different paths.	Fail

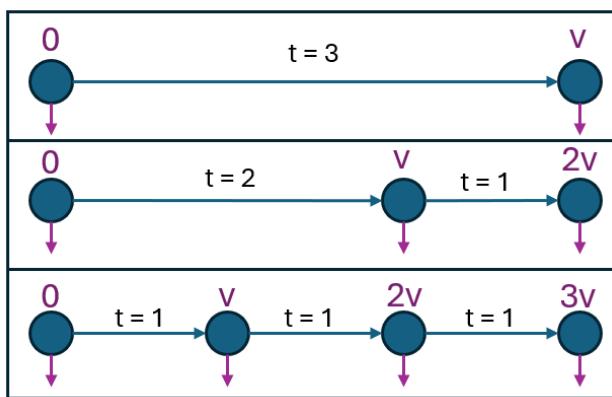


The object in the higher time scale took a much wider path than the one on the low time scale. The time scale works by increasing the distance travelled per tick by the object, so it misses out on any velocity calculations it would have had in the positions inbetween.

My first idea to fix this was to create a set step for movement, that was not affected by the time scale at all. The time scale should only ever affect the rate of updates to the objects' movement.

This could be achieved by changing the `clock.tick` line in the main running loop to speed up and slow down the entire program, without having to affect the calculations at all; the issue with this is that it would also slow the responsiveness and general functionality of the buttons and other parts of the program, so I could not do this.

Another potential solution is to increase velocity by the time scale when it is applied to the object to account for the lost increase in velocity due to the missed calculations at the points moved past. This can be done by multiplying the velocity by the time scale as it is added; this is different to the multiplication in the `gamePos` change in that the velocity will now be affected by the time scale.



In these three examples, there is an object that has a change in velocity v that is applied downwards each calculation step. In the bottom example, the time scale is 1 and so the three calculation steps produce the correct velocity of $3v$ at the end. In the middle example, the time scale is switched from 2 to 1, and the final velocity is calculated as $2v$. In the top example, the time scale is 3 and as such only one calculation step has happened.

This shows that by multiplying the velocity by the time scale as described, this should in theory solve the inconsistency. Running the program to test, it ran more consistently than before but there were still situations where changing the time scale would have different outcomes. The most visible effects were in the 6-planet system with the star as well, where some planets would collide and some wouldn't at different time scales.

Due to the complexity of the issue and the time it would likely take to completely fix, I will revisit this after I complete more of the main features of the program. For now, I will leave it as a known bug.

Finally, to check if I was still meeting the good performance requirement after coding such a calculation-intensive section of the program, I printed the FPS of the program as it was running.

```
<<Clock(fps=61.35)>>
<<Clock(fps=61.35)>>
<<Clock(fps=61.35)>>
<<Clock(fps=61.35)>>
<<Clock(fps=61.35)>>
<<Clock(fps=59.88)>>
```

These numbers were recorded with the 6-planet scenario from the previous tests; the lowest number I saw was 57FPS, which at a limit of 60 is very good performance, so I am still satisfied that that requirement is fulfilled.

Stakeholder Feedback

The stakeholders were enthusiastic about the new button designs, which they said looked a lot cleaner. However, multiple stakeholders said that they would like the time scale button to be replaced with a slider, as it would allow for a greater level of control over the speed of the simulation.

Another thing that Kai in particular would like to see is an indicator of the position of the object that is currently being focused on. This may be a solution to another piece of stakeholder feedback from Oli, who said that it was difficult to tell whether or not an object was being focused on, especially when the simulation was paused.

Stage 3 – Orbit/trajectory and equipotential lines

Success criteria

REQUIREMENT	DESCRIPTION	SUCCESS CRITERIA	CHECK
3	Orbit lines	Accurate lines that predict a reasonable distance ahead what the trajectory of an object will be.	✓
3A	Equipotential lines (E)	Accurate equipotential lines between objects that can be toggled on or off.	✓

3B	Apoapsis and periapsis markers (E)	Accurate apoapsis/periapsis markers displayed at correct points on an orbit line.	✓
-----------	------------------------------------	---	---

Testing plan

Test #	Test description	Expected outcome
1a	Orbit line drawing: show orbit lines, then unpause simulation	Orbit lines are calculated correctly and drawn on screen.
1b	Orbit lines aren't calculated when not shown	<code>predictOrbit()</code> is never called when button untoggled.
1c	Single call for orbit line calculations per tick	Method is called only once.
1d	Correct number of points calculated in future	<code>generatePointSet()</code> called the correct number of times.
2a	Apoapsis and periapsis calculations: master sprite determined correctly for each object (7-object system)	Master sprite correctly determined for all objects, and orbit markers displayed as such.
2b	Correct positions determined for apoapsis and periapsis markers	Apoapsis markers at closest points, periapsis markers at furthest points from master sprites
3a	Equipotential lines: all coordinates on screen are examined when calculating lines	First point examined is (0,0), last point is (screen width, screen height)
3b	Equipotential lines aren't calculated when not shown	<code>calcEquipotential()</code> never called when button untoggled.
3c	Single call for equipotential calculations per tick	Method is called only once.

Development

Firstly, I am going to display the game position of the object currently focused on, as per stakeholder feedback from the last development stage. To do this, I will need to use the `pygame.font` module.

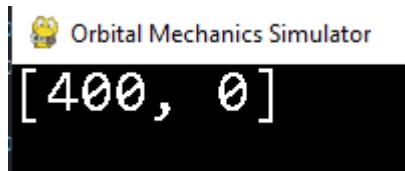
Inside the main running loop, I will create and paste the coordinates of the focused object (if there is one) to the screen in the top left corner. I have chosen to locate the display here in order to not stuff all the buttons and icons into two sides of the screen, and it will make it more prominent to the user.

I will need to load a font to use just before the main running loop before I can draw it to the screen; I have chosen to use Consolas for now, but this can be easily changed if stakeholders want a different font.

```
#draw objects to screen
physObjects.draw(screen)
buttonGroup.draw(screen)

if physEng.focusedObject != 0: #if there is an object in focus:
    screen.blit(font.render(str(physEng.focusedObject.gamePos), False, (255, 255, 255)), (0,0))

pygame.display.flip() #updates the screen
```



Running the program to test, the coordinates display correctly. However, they do not change as the object moves. I printed the gamePos of the focused object, which showed that the gamePos was not changing as the simulation ran. This was again due to the annoying physEng.cameraFocus() being subtracted from it to create the offset when focusing. Once more I moved it to the screenPos set calculation.

Doing this once again broke the camera focus as well as the zoom functions, however interestingly moving the zoom slider would fix the focus feature until it was released.

The zoom slider is just recalculating the initial focus offset every tick while it is activated, while the camera focus applied the acceleration of the focused object to every other object to produce a focus effect. I tried changing the focus system to instead recalculate the initial focus offset every tick, like the zoom function.

Where I added the initialFocusOffset to the screenPos in the planet draw function, I instead called the cameraFocus() method from the physics engine and removed any other references to this in the planet update() method. In the cameraFocus() method, I performed the same initialFocusOffset calculation as done in the cameraUpdate() method.

As the calculations are done every tick, this means that I do not need an initial focus offset, so I can remove any instances of this variable.

Running the program now, both the zoom and the focus features are back to working correctly, and the gamePos coordinates are updating while an object is in focus. One side effect of these changes is that the planets, when moved by the camera function, have their gamePos changed instead of their screenPos.

To remedy this, I created a new Vector2 variable inside the physics engine cameraUpdate() method called movementOffset, which will be added to the screenPos of each physics object before they are drawn to the screen. This will be instead of looping through each physics object and changing their gamePos directly. This works and keeps the gamePos consistent. The object group no longer has to be passed through into this method as a parameter anymore.

Another issue is that when the camera moves out of focus, it is sent back to gamePos (0,0). This is another easy fix, with the movementOffset being set to what was the initial focus offset after the object to focus on is passed through in the cameraUpdate() method.

```
def updateCamera(self, direction, screen): #function to produce uniform movement for all physics objects in cardinal directions
    self.focusedObject = 0 #reset object focus
    match direction:
        case 0: #no direction passed/clear direction
            return
        case "N": #north
            self.movementOffset[1] -= 5
        case "S": #south
            self.movementOffset[1] += 5
        case "E": #east
            self.movementOffset[0] -= 5
        case "W": #west
            self.movementOffset[0] += 5
        case _: #object to focus on passed through
            self.focusedObject = direction
            self.movementOffset = Vector2(screen.get_width()/2, screen.get_height()/2) - self.focusedObject.screenPos
```

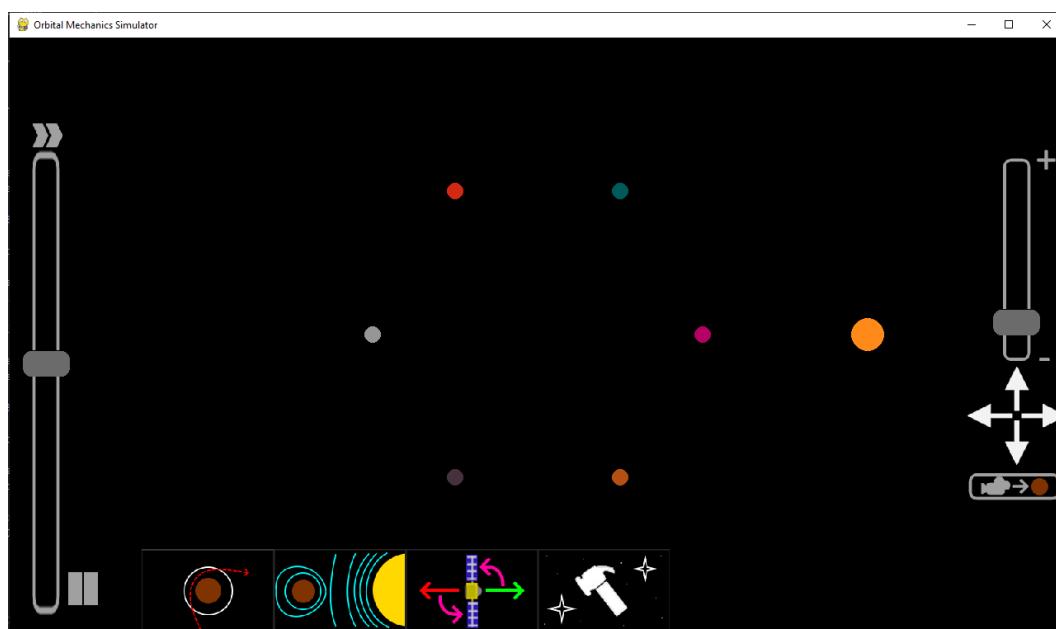
I have also discovered that scrolling while the simulation slows it down. I have no clue why this is happening, as there is nothing to do with the scroll wheel controls coded into this, but if I had to guess I would say that the Pygame events system is checking every instance of the mouse wheel being scrolled. As the mouse wheel should not have to be scrolled while the program is running, I will ignore this issue for now and will come back to it later if I have time after development is done (or if it becomes a more prominent issue).

Running the FPS counter, I ran an average of 40 frames per second when in fullscreen mode (still ~60FPS in window). This is still within the acceptable range, but if the performance worsens then this previous work will be an area to improve upon.

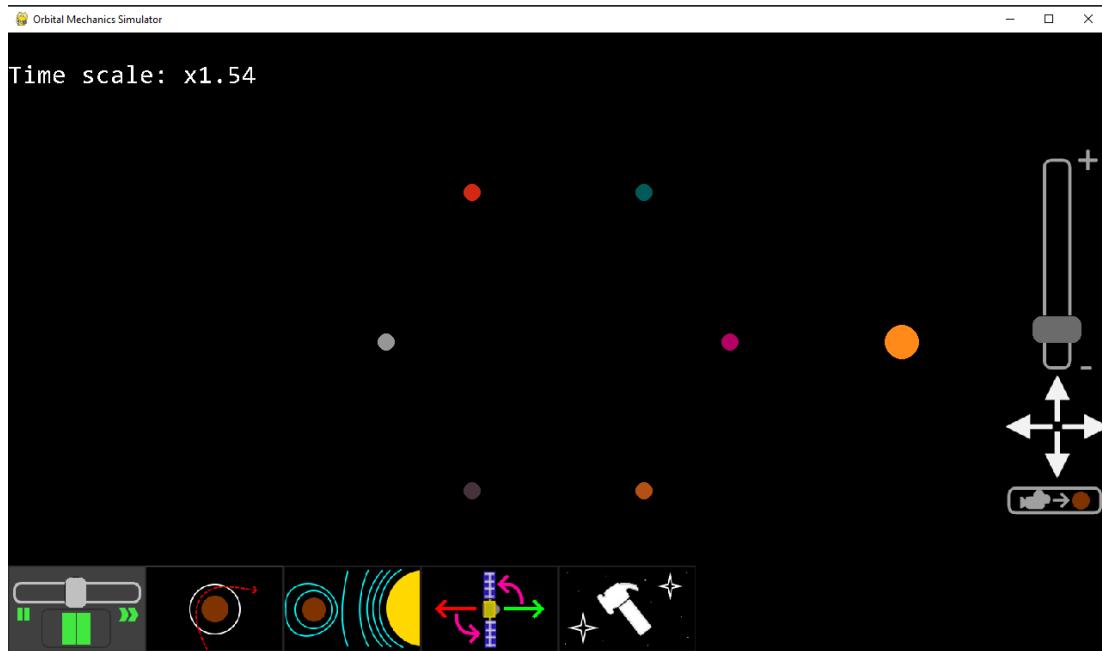
The other stakeholder feedback I received last development stage was to make the time scale button into a slider. This will have designation 21.

In the `effect()` and `rescale()` methods, I copied the zoom rect slider positioning code and switched it to use the time scale variable from the physics engine, and for it to display on the left side of the screen. I also need to change the translation of values between the slider and the variables, as I intend for the time scale to be able to be set to 0 unlike the zoom scale and reposition the slider to a new position on screen.

To create a background for the slider, I will draw more onto the focus button background as it is already being used for the zoom slider background, meaning that it will be easier to position.



I'm not so sure about the size and position of this slider, so I will experiment with making it smaller to not clutter up the GUI. After consulting quickly with stakeholders, this was the solution I came up with:



There is also a dedicated button to pause the simulation independent of the slider, with designation 0. I initially had trouble getting it to show; this was because button 5 was being drawn over button 0 in the button group draw function. To fix this, I instantiated button 0 after all other buttons had been instantiated. In order to make this pause button independent of the slider, so that the slider could be manipulated while the simulation was paused.

```
def getTimeScale(self):
    if self.isPaused:
        return 0
    else:
        return self.timeScale

def effect(self): #applies effects of buttons
    match self.designation: #compare designation:
        case 0: #pause button
            physEng.isPaused = self.isClicked #sets physics engine to be paused/unpaused based on current state of button
```

To do this, I had to create a new method inside the physics engine, `getTimeScale()`, that is called by each physics object when performing calculations to change their `gamePos` instead of directly using the time scale from the physics engine.

I also experimented with drawing the value of the time scale to the screen, by drawing the text to the screen in the `effect()` method of the time scale slider. This would make the value of the time scale appear in the top left. This value would then disappear when the slider was not pressed.

```
screen.blit(font.render(f"Time scale: x{str(physEng.timeScale)}", False, (255, 255, 255)), (0, screenHeight*0.05)) #draw time scale value on screen
```

I really liked this feature, so I also implemented it for the zoom slider.

Now that the stakeholder feedback has been implemented, I can actually begin work on the orbit lines. As outlined in the design phase, this feature will work by calculating the positions of objects a number of steps in the future before joining these points to create a line.

While in my design I called the calculations for the orbits via the physics engine update() method, I realised now that the calculated values could not be easily returned in order to be drawn to the screen. Consequently, I will be calling the physics engine's predictOrbit() method from the main running loop instead.

```
if buttonGroup.sprites()[0].isClicked: #if orbit Lines are being shown (orbit Lines are first in buttonGroup list as pause button is created later):
    for i in physObjects.sprites(): #for each physics object:
        pointsToDraw = physEng.predictOrbit(i, 10, screen) #calcualte orbit
        #-----  
        TODO: draw to screen
```

Creating the predictOrbit() method in the physics engine, the first step is to get the position of the object passed through one movement step in the future. This will use the same calculation as in the planet update() method.

```
def predictOrbit(self, object, points, screen):
    pointArray = [] * points #create array of points for number of points to Look ahead for
    futurePos = object.gamePos

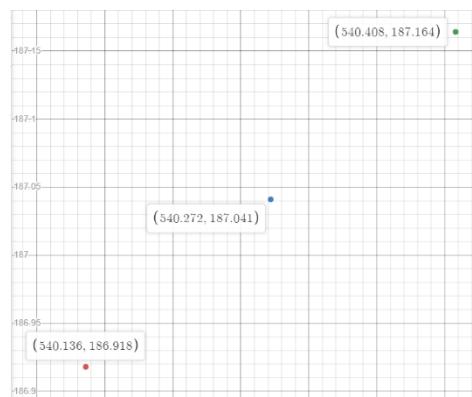
    for i in range(points): #number of points to plot ahead in orbit
        futurePos = futurePos + object.velocity * self.timeScale #calculate future movement of sprite in game world
        pointArray[i-1] = Vector2(screen.get_width()/2 + futurePos[0] * self.zoomScale, screen.get_height()/2 + futurePos[1] * self.zoomScale)

    return pointArray
```

Running for the first time, I got an error for my list being out of range in the predictOrbit() function when I clicked the orbit line button. I swapped around the variables, now subtracting 1 from points in the range line, and ran the program again. I received the same error. I printed the pointArray variable to see what its value was, and it showed the array being created was only one unit long. I put a 0 inside the square brackets when creating the pointArray, and received this output when the program was run with i as 4:

```
[<Vector2(540.136, 186.918)>, <Vector2(540.272, 187.041)>, <Vector2(540.408, 187.164)>, 0]
PS E:\NEA (USB)\proj>
```

This was one line in a range of lines and represented the future movements of a single object.



I plotted these points on an online graphing calculator, which showed that the movement of this object would be linear. However, the actual movement of this object would be affected by the other objects in its vicinity, which I was not considering. Consequently, I added a call to the calcGravBetweenObjects() method, which I changed to return the vector of force calculated to accommodate for this change. In the update() method, I directly changed the velocities of the objects using the result returned to ensure that the program would function in the same way.

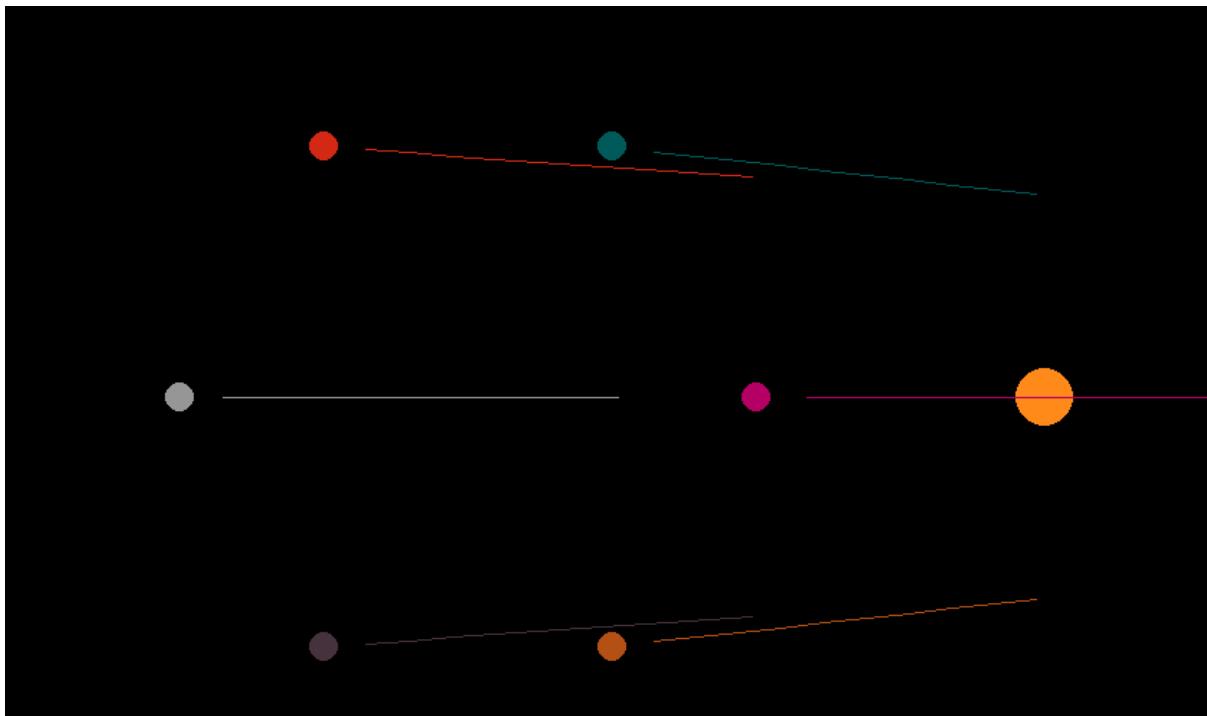
```
def predictOrbit(self, object, objectGroup, points, screen):  
  
    pointArray = [0] * points #create array of points for number of points to look ahead for  
  
    futurePos = object.gamePos  
    velocity = object.velocity  
  
    for i in range(points): #number of points to plot ahead in orbit  
  
        #calc grav again w velocity  
        for j in objectGroup.sprites(): #for each sprite:  
  
            k = 0  
            while k < len(objectGroup.sprites()): #for each other sprite:  
  
                velocity = self.calcGravBetweenObjects(j, objectGroup.sprites()[k])  
                k += 1  
  
            print(velocity)  
  
        futurePos = futurePos + velocity #calculate future movement of sprite in game world
```

The other issue with this output was that the coordinate of the last point would not be calculated, which was again an issue with my ranges which was very easy to fix.

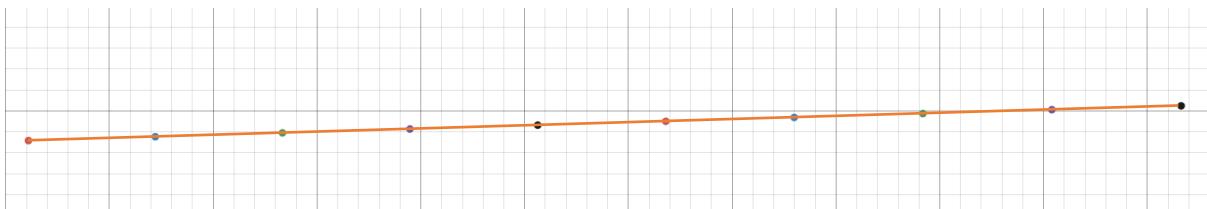
Now I have an array containing the three correctly predicted points, I will draw each one onto the screen with the colour of the planet that they are calculated from (to ensure that it is obvious which planet has which orbit line) inside the main running loop:

```
for obj in physObjects.sprites(): #for each physics object:  
  
    pointsToDraw = physEng.predictOrbit(obj, 100, screen) #calculate orbit  
  
    pygame.draw.lines(screen, obj.colour, False, pointsToDraw) #draw orbit lines on screen
```

Running the program now, the lines do not show on the screen. Printing out the velocity after calculating it in predictOrbit(), it returned a value of 0. This was all due to me missing a + sign, and just setting the velocity to the result of the calcGravBetweenObjects() method instead of adding to it.



The points are now being drawn, which is good, however the fact that they are still linear is suspicious. Additionally, the lines aren't centred (easily fixed by making the first point in the sequence be the centre of an object), and as the simulation runs the lines keep extending indefinitely, which shouldn't be a problem as long as the lines are correct. I again plotted the points generated as the `pointArray` for one of the objects, which should have resulted in a curve.

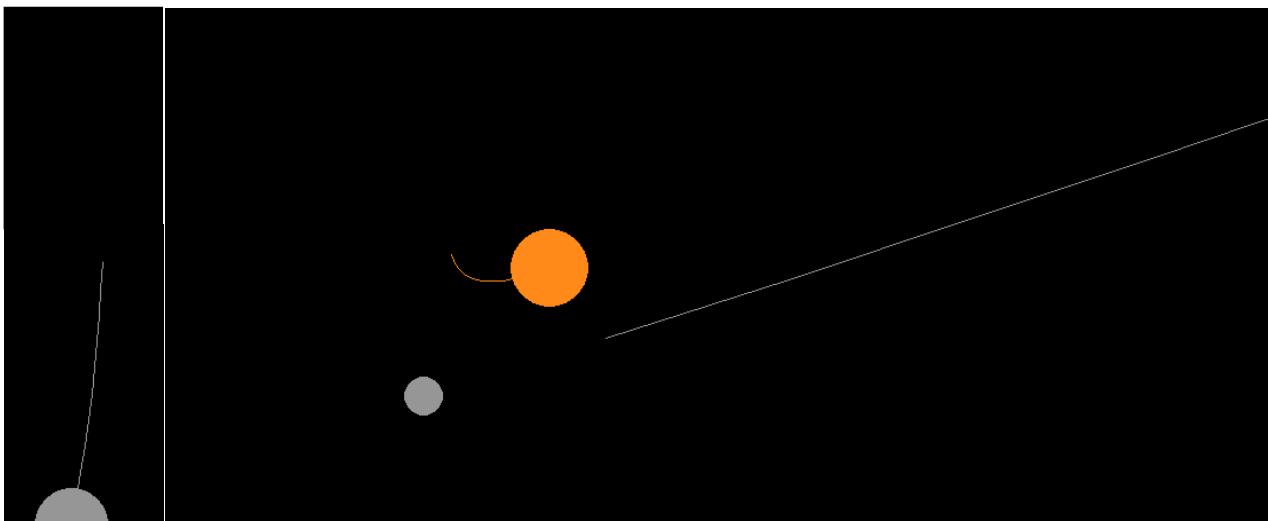


It looks like even after accounting for the gravitational effects, after the first point the velocity does not seem to be calculated correctly.

Also, for some reason, changing “`futurePos = futurePos + velocity`” to “`futurePos += velocity`” affects the movement of the planets even when the game is paused, so I will just leave it as the first one.

To simplify things, I switch back to using just two objects for testing this. When I ran the program, everything worked fine up until the point when I clicked the orbit lines button. This caused both objects to gain velocity to the right, instead of moving towards one another. This was again fixed by changing an instance of `+=` in the `predictOrbit()` method to just `+` with the variable repeated. I do not know how this is any different logically, but it works.

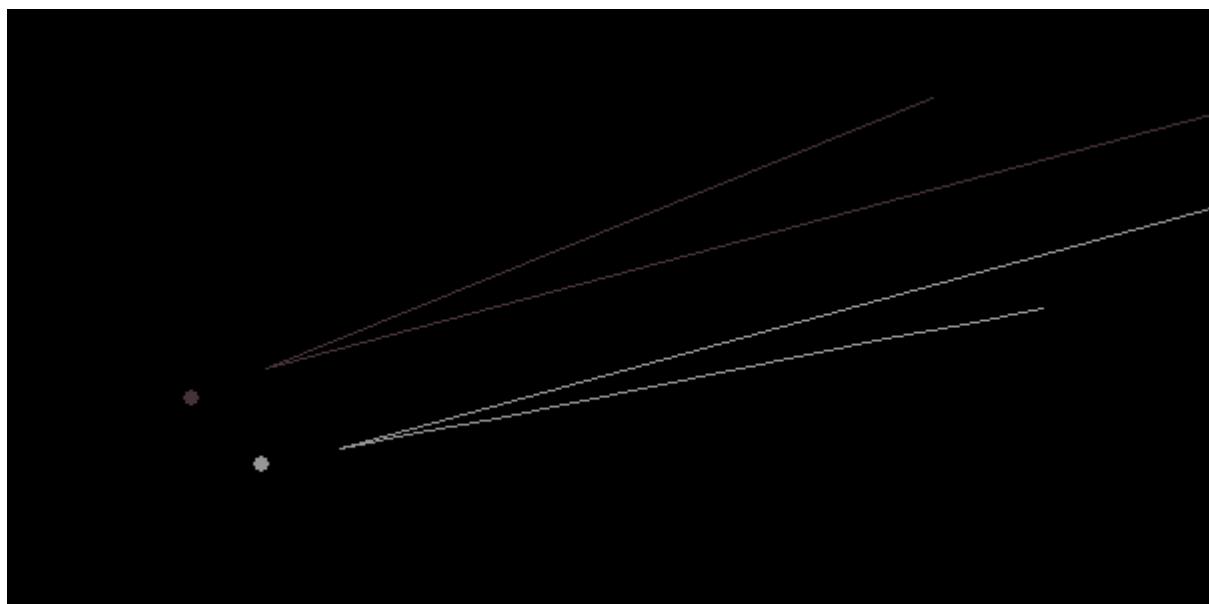
After fixing this, the line begins to move slowly to a different orientation when the simulation is paused, and the orbit lines are turned on. When unpause, the object moves in the direction the line is pointing to, even when I know that the velocity I gave it was straight downwards. While letting the planet pass by the star, I took a screenshot to see if the line was still straight while the object's direction was changing so quickly.



This showed that the prediction path was turning, but so gradually that it was usually not visible. To get a longer and more curved line, I multiplied the velocity when it was added to the futurePos by 15 to space out the points. Now when the planet passed the star, the line became stretched and did not correctly predict the planet's movement. The orbit line for the star also incorrectly predicted that it would move upwards and away from the planet.

All of the issues with the velocity calculated in this method and the velocity in terms of the simulation seem to have stemmed from the designation of the futurePos and velocity internal variables as the object.gamePos and object.velocity; to combat this, I will instead pass through these two as parameters instead of passing through the object itself.

This had zero effect. Also, when I zoomed in or out the simulation would still run but the orbit lines seemed to freeze in place. I re-added the other planets, and added a line to insert the position of the planet at the beginning of the list of points for its orbit line to ensure that the lines would connect with the planet in question. This resulted in a lot of very interesting orbits, such as the one below.



The behaviour of these objects changed completely when I disabled orbit lines. When I focused on an object, a second line for each object was drawn and moved according to the position of the screen instead of the planets (as was the case above); I suspect that I am somehow passing the parameters by reference instead of by value. After researching the way that Python handles this, I

found that the changes from `+=` to `=` were necessary for ensuring that only the local variable was changed. I then attempted to create a buffer variable for the velocity of the objects to try and mitigate the risk of the object being passed by reference, but this seemed to have little effect.

I decided it might be better to rethink my strategy of creating these predictions. As it is now, the program looks a number of steps ahead in a planet's movement and does not consider the future movement of the other planets in the scenario. Instead of calculating each planet's full orbit line individually, I could instead calculate the movement of all planets one movement step in the future and repeat this process for the length of orbit line I require.

To do this, I need to create a new 2D array in the format [[A, B], [C, D], [E, F]] where each letter represents a Vector2 coordinate; A, C and E would be the current positions of the objects, and subsequent letters in the same sublist would be the points for that prediction. Each sublist represents each individual object and allows me to return the points for each object's line in a single variable, as well as calculating their future effects on each other based on all objects' future movements.

The subarrays are created for the number of points needed, and this is multiplied by the number of objects that the orbit lines are to be calculated for. The first point to be drawn will be the centre of the object.

To get the point one step in the future, I will again utilise the `calcGravBetweenObjects()` method and simply add the result to the previous point to find its position in the future. I will then be able to utilise these generated points to calculate the next set, and so on until the orbit points have been generated.

I will also need the velocities of the objects in an array for this, as they will change as the calculations progress.

```
def predictOrbit(self, sprites, points, screen):
    pointArray = [[0] * (points + 1)] * len(sprites) #create array of points for number of points to look a
    velocityArray = []

    for i in range(len(sprites)):
        pointArray[i][0] = sprites[i].gamePos
        velocityArray.append(float(sprites[i].velocity)) #get velocities of all sprites in object group

    for i in range(points): #for the requested number of points:
        self.generatePointSet(i, pointArray, sprites) #generate a set of points for i steps in the future

    #print(pointArray) #debug
    #CONVERT TO SCREENPOS
    return pointArray
```

I have also created a `generatePointSet()` method to calculate each set of points more easily.

```
def generatePointSet(self, pointLevel, array, sprites, velocities):
    for i in range(len(sprites)): #for each sprite:
        for j in range(len(sprites)): #for each other sprite in group:
            array[i][pointLevel] = array[i][pointLevel - 1] + self.calcGravBetweenObjects(sprites[i], sprites[j]) + velocities[i]
```

After setting the initial values in the `pointArray` to be an empty Vector2 instead of a single 0, I ran the code and got this output:

Orbital Mechanics Simulator – Sam Prince

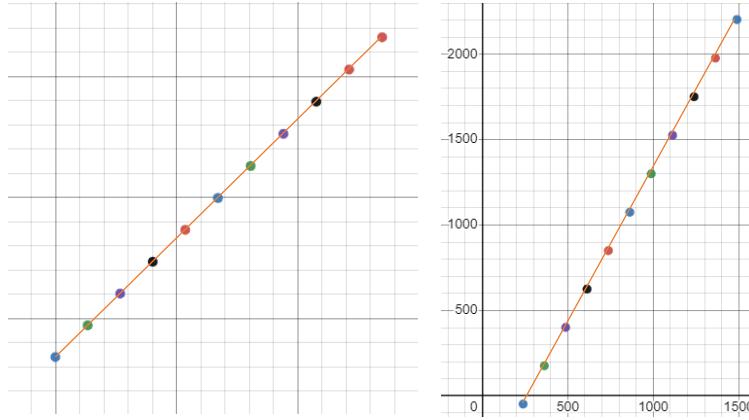
Centre number 64859 | Candidate number 6103

```
[[<Vector2(0.439114, 0.429846)>, <Vector2(0.878228, 0.859693)>, <Vector2(1.31734, 1.28954)>, <Vector2(1.75646, 1.71939)>, <Vector2(2.19557, 2.14923)>, <Vector2(2.63468, 2.57908)>, <Vector2(3.0738, 3.08892)>, <Vector2(3.51291, 3.43877)>, <Vector2(3.95202, 3.86862)>, <Vector2(4.39114, 4.29846)>, <Vector2(4.39114, 0.0)>], [<Vector2(0.439114, 0.429846)>, <Vector2(0.878228, 0.859693)>, <Vector2(1.31734, 1.28954)>, <Vector2(1.75646, 1.71939)>, <Vector2(2.19557, 2.14923)>, <Vector2(2.63468, 2.57908)>, <Vector2(3.0738, 3.08892)>, <Vector2(3.51291, 3.43877)>, <Vector2(3.95202, 3.86862)>, <Vector2(4.39114, 4.29846)>, <Vector2(4.39114, 0.0)>], [<Vector2(0.439114, 0.429846)>, <Vector2(0.878228, 0.859693)>, <Vector2(1.31734, 1.28954)>, <Vector2(1.75646, 1.71939)>, <Vector2(2.19557, 2.14923)>, <Vector2(2.63468, 2.57908)>, <Vector2(3.0738, 3.08892)>, <Vector2(3.51291, 3.43877)>, <Vector2(3.95202, 3.86862)>, <Vector2(4.39114, 4.29846)>, <Vector2(4.39114, 0.0)>], [<Vector2(0.439114, 0.429846)>, <Vector2(0.878228, 0.859693)>, <Vector2(1.31734, 1.28954)>, <Vector2(1.75646, 1.71939)>, <Vector2(2.19557, 2.14923)>, <Vector2(2.63468, 2.57908)>, <Vector2(3.0738, 3.08892)>, <Vector2(3.51291, 3.43877)>, <Vector2(3.95202, 3.86862)>, <Vector2(4.39114, 4.29846)>, <Vector2(4.39114, 0.0)>], [<Vector2(0.439114, 0.429846)>, <Vector2(0.878228, 0.859693)>, <Vector2(1.31734, 1.28954)>, <Vector2(1.75646, 1.71939)>, <Vector2(2.19557, 2.14923)>, <Vector2(2.63468, 2.57908)>, <Vector2(3.0738, 3.08892)>, <Vector2(3.51291, 3.43877)>, <Vector2(3.95202, 3.86862)>, <Vector2(4.39114, 4.29846)>, <Vector2(4.39114, 0.0)>]]  
PS D:\NEA (USB)\proj>
```

I then realised that I had not implemented a way for the velocities of each object in their new positions to be accounted for.

```
for j in range(len(sprites)): #for each other sprite in group:  
  
    velocities[i] += self.calcGravBetweenObjects(sprites[i], sprites[j])  
  
array[i][pointLevel] = array[i][pointLevel - 1] + velocities[i] #set point at Level in array as predicted point
```

To fix the issue of the last points in the array not being calculated, I added 1 to the number of points being generated. I also increased the start value of the point array changes to be 1 to avoid overwriting the start point.



I plotted the output of the new code into Desmos to find that it was another linear output. However, this was for an object that moved in a linear way at the start of the program, so I ran the simulation before pausing it again and then activating the orbit line calculations. I got another linear result, showing me that it was still not working correctly.

What was however promising was that unlike in the previous iteration, the planets would move in the right direction instead of being inconsistent when the orbit lines were enabled. This showed that I was on the right track.

In order to more easily visualise the orbit lines while testing, I re-implemented the draw function. I would also need to convert the positions of the points, currently in a gamePos position, to the screenPos format in order to draw correctly to the screen.

I will need to do this in an identical way to the conversion in the planet class. To make the code simpler and more modular, I decided to combine the cameraFocus() method with the positioning code in the planet class into a single getScreenPos() method in the physics engine. This will be called by any object or code that needs to convert a gamePos into a screenPos, which will ensure that the software is consistent.

```
def update(self):  
  
    self.gamePos += self.velocity * physEng.getTimeScale() #move sprite in game world  
  
    self.screenPos = physEng.getScreenPos(self.gamePos, screen)  
  
    pygame.draw.circle(screen, self.colour, self.screenPos, self.radius * physEng.zoomScale) #draw planet  
  
    self.rect = self.image.get_rect(center=self.screenPos) #update rect, centred at the planet's position
```

Above: the new planet update() method, calling the physEng.getScreenPos() method. Below: the getScreenPos() method in the physics engine, changed from the cameraFocus() method.

```
def getScreenPos(self, gamePos, screen): #convert gamePos to screenPos

    screenPos = Vector2(screen.get_width()/2 + gamePos[0] * self.zoomScale, screen.get_height()/2 + gamePos[1] * self.zoomScale)

    if self.focusedObject != 0: #if there is an object to be focused on:

        self.movementOffset = Vector2(screen.get_width()/2, screen.get_height()/2) - self.focusedObject.screenPos #change movement offset

    screenPos += self.movementOffset #change position by movement offset

    return screenPos
```

Running to test, the positions were being calculated correctly and the zoom function worked as normal. However, the focus method would shake the screen. I commented out the line resetting the movement offset every calculation, which fixed the jittering but made the focus function have no effect aside initially moving the camera to the focused planet's position.

If the object that is being focused on is calling this method, it is recalculating its screen position before using its previous position to offset itself, which may result in this effect. To account for this, I can pass the object calling the function as a parameter to check if it is the focused object, and if so skip this recalculation for the movement offset. This did not work.

I then tried to change this so that instead of updating every time the method was called, it updated in the physics engine update() method instead. This fixes the stuttering until the simulation is unpause but overshoots the object to be focused on. I kept the code for the movement offset update inside this method, as it is called less often than if kept in the camera method.

I printed the offset and tested as it changed to find that it was adding the offset to the previous offset; to fix this, I added the offset of the previous focused object, which made the focus system work again when the simulation was paused. In an attempt to make it work again when the simulation was running, I added the offset when calculating it in the update() method, which fixed the issue.

Checking again the output of the pointArray, it changes the value of every sublist when only one item in a single sublist is changed. Looking online, it seems that this is due to the way I first initialise the array with the same sublist object. This is simple to fix:

```
pointArray = [[Vector2(0,0) for i in range(points + 1)] for j in range(len(sprites))]
```

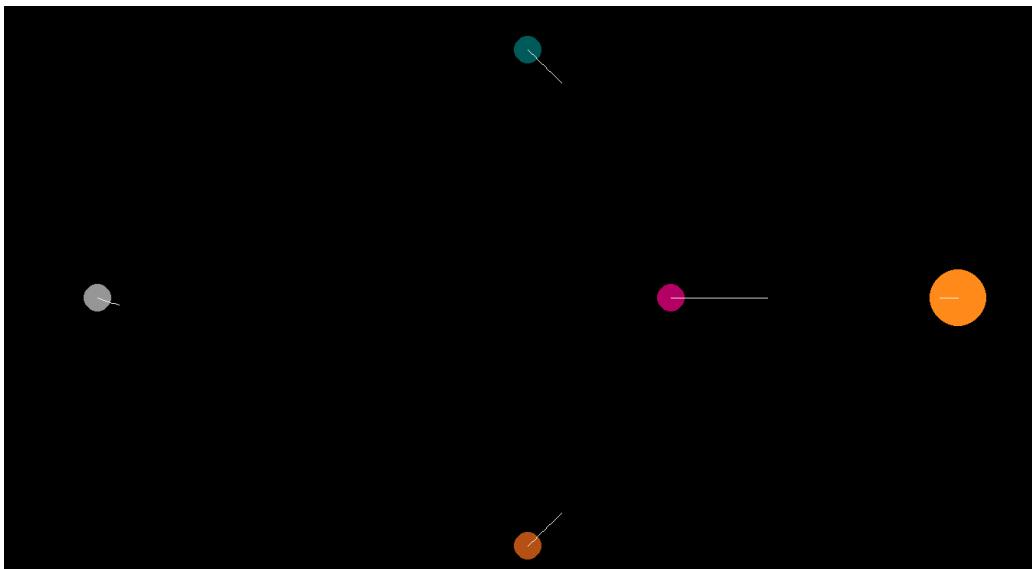
Back in main.py, to draw the lines to the screen is very similar to from before, instead this time I have to loop over every list in the 2D array. Another side effect of using this method over the last one is that I will have to use a single colour for all orbit lines.

```
if buttonGroup.sprites()[0].isClicked: #if orbit lines are being shown (orbit button is pressed)

    pointsToDraw = physEng.predictOrbit(physObjects.sprites(), 10, screen) #

    for i in pointsToDraw: #for each object's orbit line:

        pygame.draw.lines(screen, (255,255,255), False, i) #draw orbit line
```



I was very pleased with this output, as the lines were pointing in the right direction, and the line on the grey planet (which had some downwards velocity) was visibly curving. The simulation was paused, however the lines were moving and extending, so I was worried that the orbit lines were again affecting the velocities of the planets.

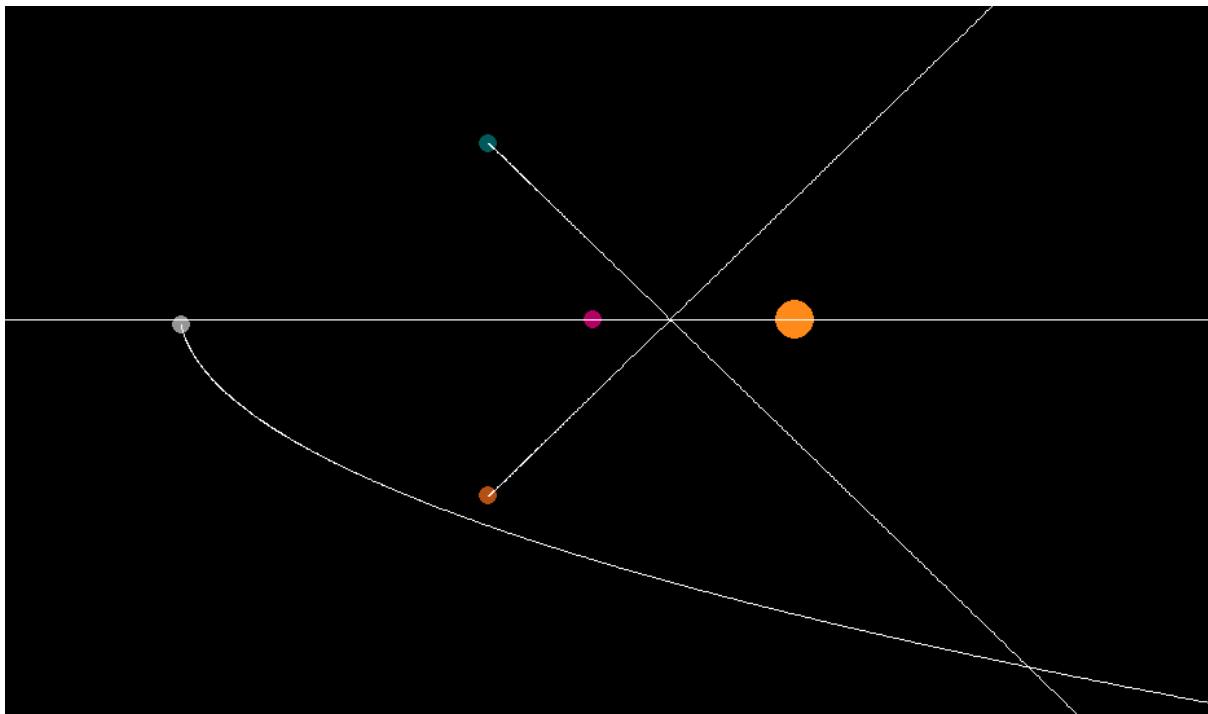
To check that the velocities in the `velocityArray` were being passed by value and not by reference, I used the line `print(velocityArray[0] is sprites[0].velocity)` which returned false for all objects, so that was not the issue. The only other thing that could be causing this would be if the physics calculations that changed the velocities were still being run in the game's paused state. To check this, I printed the velocities of the planets as the game ran.

```
[1.23985, -1.21368]  
[1.23985, 1.21368]  
[-0.678538, 0]  
[0.779005, 0.5]  
[3.51634, 0]  
[1.24502, -1.21874]  
[1.24502, 1.21874]  
PS E:\NEA (USB)\proj> []
```

When the program is started, the velocities of all the objects change, even when nothing has been pressed. This is telling me that even though the physics engine calculation step should not be being called, something is calling it. The only reference to the physics engine update step in the main file is in the main running loop, which checks if the pause button is clicked or not. Writing a debug line to print a flag when this is active showed that it was still being run, even if the pause button was active. The pause button state was not showing as

paused; I checked that I was referencing the right button in the group, which I was not. Changing this solved the issue.

I then increased the number of points being calculated from 10 to 100, as the previous change stopped the orbit lines from changing while the simulation was paused and so they became very small, which produced this effect (while also creating a lot of lag):



The issue with this is that the orbit line prediction is still not perfect in its current iteration; the lines still swing around when passing close to a planet instead of showing a perfect orbit.

Checking my code again, the `calcGravBetweenObjects()` method uses the distance between the two objects as they appear in the simulation in the calculation; this means that future points do not consider the distance between the two objects when calculating the orbit lines.

To fix this, I will have to pass through the radius as a parameter into the method instead of calculating it within it. This necessitates a change in the physics engine `update()` method as well as the `generatePointSet()` method but should not impact performance.

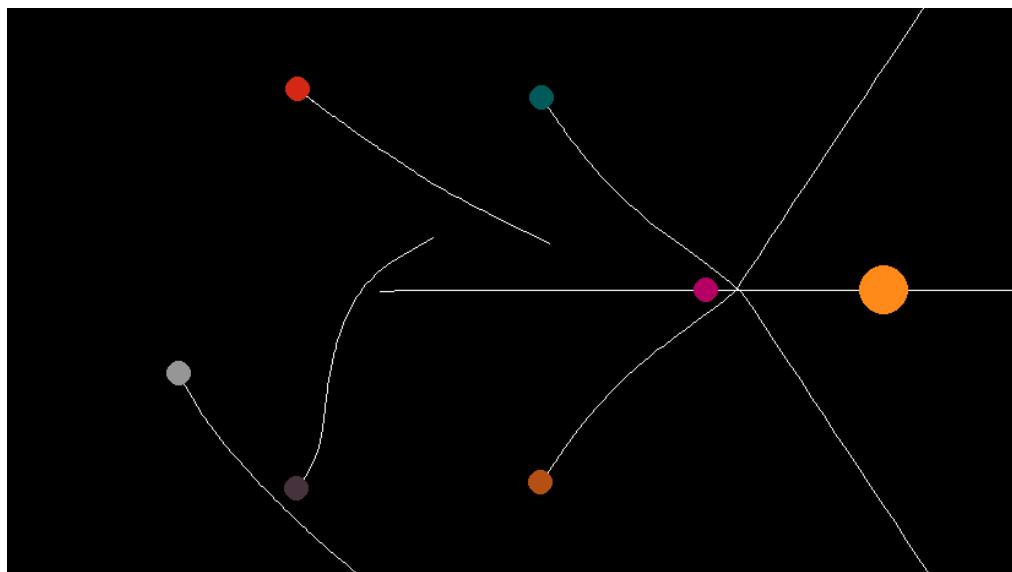
I then realised that the vector returned from the `calcGravBetweenObjects()` method was also created using the `gamePos` of each object; the only way to fix this is to return the force instead of the velocity vector, and instead move the velocity calculation outside into the `update()` method for the main movement, and a modified version into `generatePointSet()` for the orbit prediction.

```
for i in sprites: #for each sprite in group:  
    j = 0  
    while j < len(sprites): #for each other sprite:  
        i.velocity += Vector2((sprites[j].gamePos[0]-i.gamePos[0])/math.sqrt((sprites[j].gamePos[0]-i.gamePos[0])**2+(sprites[j].gamePos[1]-i.gamePos[1])**2),  
                               (sprites[j].gamePos[1]-i.gamePos[1])/math.sqrt((sprites[j].gamePos[0]-i.gamePos[0])**2+(sprites[j].gamePos[1]-i.gamePos[1])**2)) * self.calcgravBetweenObjects(sprites[j], i.gamePos.distance_to(sprites[j].gamePos) * 1e6)  
        j += 1
```

Yes, I am aware of how horrible this line is. I will probably optimise it later if there is time, but for now it works fine.

```
Traceback (most recent call last):  
  File "E:\NEA (USB)\proj\main.py", line 428, in <module>  
    physEng.update(physObjects, screen) #run physics calculation step  
  File "E:\NEA (USB)\proj\engine.py", line 33, in update  
    i.velocity += Vector2((sprites[j].gamePos[0]-i.gamePos[0])/math.sqrt((sprites[j].gamePos[0]-i.gamePos[0])**2+(sprites[j].gamePos[1]-i.gamePos[1])**2),  
                          es[j].gamePos) * 1e6) #calculate gravitational force  
ZeroDivisionError: float division by zero  
PS E:\NEA (USB)\proj\]
```

It doesn't work fine. This is because the check to see if the distance between the two sprites is no longer present, so I will add that into both instances of this calculation. Running again, it actually does work fine.



In fact, it works so well that I am happy the orbit lines are working 100% as intended. It would be nice to have these extend further, but the more points that are calculated for, the higher the impact on performance.

There are multiple optimisations that I can do in order to make these calculations more efficient, and the program run faster: the most major one is only calculating these orbit lines after certain points have been passed in the orbit, such as apoapsis or periapsis points, or parameters have changed. I could also make it so that only planets that are within the bounds of the screen and can be seen by the user have their orbit lines drawn. However, I decided to instead continue with work on the apoapsis and periapsis markers, as I could always come back to optimising the orbit lines; there are still many more features to implement.

The apoapsis and periapsis markers for each orbit line will be calculated using a new `calcApoapsisAndPeriapsis()` method in the physics engine; these markers will be relative to the object with the largest gravitational force acting on the object that has the orbit lines drawn for it.

```
def calcApoapsisAndPeriapsis(self, pointArray):
    #for each object:
    #get sublist from pointArray
    #calculate object of greatest acting force on that object
    #get point closest to object = periapsis
    #get point furthest = apoapsis
    pass
```

To get the points to display the markers, I will have to recalculate all of the gravitational forces between the objects – something which I have done already in the `generatePointSet()` method.

Consequently, it would be more efficient to just get the object from that calculation in that method (referred to as `masterObject`) and use the `calcApoapsisAndPeriapsis()` method to check which point is the closest and furthest to return apoapsis and periapsis. I can then append two new sublists to the `pointArray` for apoapsis and periapsis respectively, which will then be used to draw the markers in the main running loop.

In the `generatePointSet()` method, I created a new array to keep track of the master objects for each object. The force being calculated each time is now being compared against the maximum value calculated for that object previously, and the index of the object with the maximum value is stored in this new array.

Orbital Mechanics Simulator – Sam Prince
Centre number 64859 | Candidate number 6103

```
def generatePointSet(self, pointLevel, array, sprites, velocities): #generate points of objects for pointLevel steps in the future
    masterObjects = [0 for i in range(len(sprites))] #create array of master sprites for each object for apoapsis/periapsis calculations

    for i in range(len(sprites)): #for each sprite:
        greatestForce = 0 #set greatest force acting on that object to 0 initially

        for j in range(len(sprites)): #for each other sprite in group:
            if i != j:
                force = self.calcGravBetweenObjects(sprites[j], array[i][pointLevel - 1].distance_to(array[j][pointLevel - 1]) * 1e6) #calculate force between objects

                if force > greatestForce: #if this is new greatest force:

                    greatestForce = force #set new greatest force
                    masterObjects[i] = j #set master object as index in sprites[] of object force was calculated from

            velocities[i] += Vector2((sprites[j].gamePos[0]-sprites[i].gamePos[0])/math.sqrt((sprites[j].gamePos[0]-sprites[i].gamePos[0])**2+(sprites[j].gamePos[1]-sprites[i].gamePos[1])**2)) * self.timeScale #set point at Level in array as predicted point

        array[i][pointLevel] = array[i][pointLevel - 1] + velocities[i] * self.timeScale #set point at Level in array as predicted point

    return masterObjects #return array of master objects for apoapsis/periapsis calculations
```

The calcApoapsisAndPeriapsis() method creates an array for each periapsis and each apoapsis, and then for each object calculates the distance between each point and the master object for that object. The maximum distance point is the apoapsis, and is appended to the array as such, with the minimum point similarly being appended to the periapsis array. The apoapsis and periapsis arrays are finally appended to the pointArray to be returned to the main file to be drawn.

```
def calcApoapsisAndPeriapsis(self, pointArray, sprites, masters):
    periapsisArray = []
    apoapsisArray = []

    for i in range(len(sprites)): #for each object:
        minDist = 9999999999999999 #set initial min distance as a large number (can't do -1 so this is the best we are getting)
        maxDist = -1 #set initial max distance as -1

        for j in pointArray[i]: #for each point in the object's set of points:
            pointDist = sprites[masters[i]].screenPos.distance_to(j) #calculate distance between point and master object

            if pointDist < minDist: #if master object of sprite's distance to point is less than the minimum:
                minDist = j #set min distance as this point

            if pointDist > maxDist: #if master object of sprite's distance to point is greater than the maximum:
                maxDist = j #set max distance as this point

        apoapsisArray.append(maxDist)
        periapsisArray.append(minDist)

    pointArray.append(apoapsisArray)
    pointArray.append(periapsisArray)
```

The predictOrbit() method has been changed to accommodate the masters array, to pass it to the calcApoapsisAndPeriapsis methods for calculation.

```
def predictOrbit(self, sprites, points, screen):
    pointArray = [[Vector2(0,0) for i in range(points + 1)] for j in range(len(sprites))] #create array of points for number of points

    velocityArray = [] #create array to store velocity of objects at future points

    for i in range(len(sprites)):
        pointArray[i][0] = sprites[i].gamePos #set the initial line to be drawn as the centre of the planet
        velocityArray.append(Vector2(sprites[i].velocity)) #get velocities of all sprites in object group

    for i in range(1, points + 1): #for the requested number of points:
        masters = self.generatePointSet(i, pointArray, sprites, velocityArray) #generate a set of points for i steps in the future

    for i in pointArray: #for each object:
        for j in range(0, points + 1): #for each vector2 coordinate:
            i[j] = self.getScreenPos(i[j], screen) #convert to screenPos (pass through 0 for object)

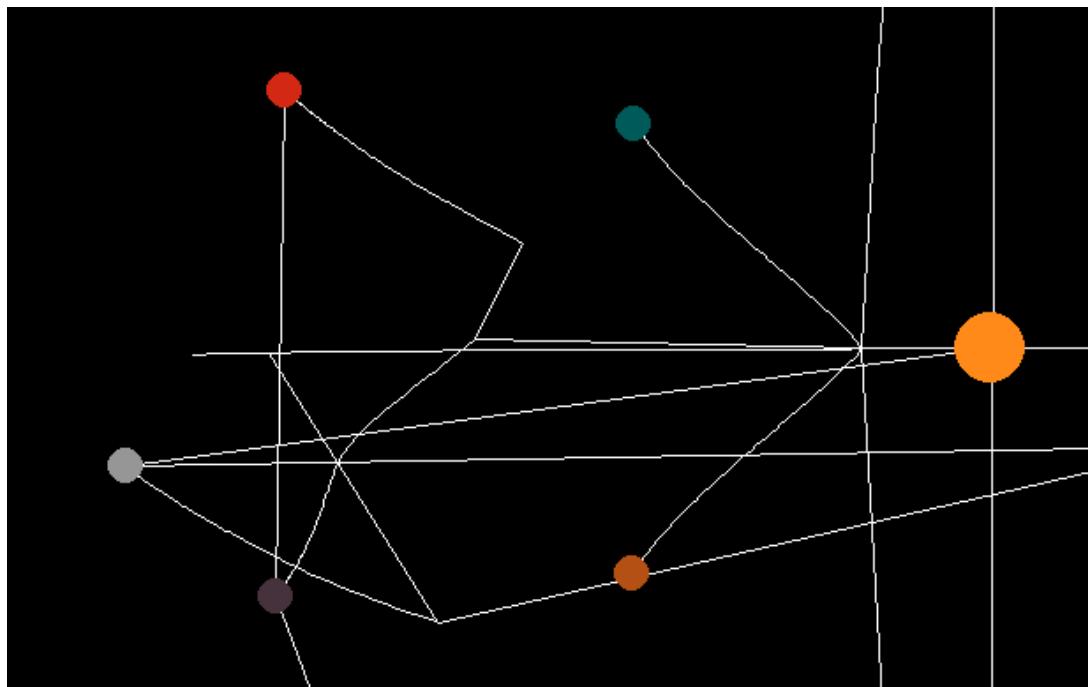
    #apoapsis and periapsis markers
    self.calcApoapsisAndPeriapsis(pointArray, sprites, masters)

    return pointArray
```

Running for the first time:

```
Traceback (most recent call last):
  File "e:\NEA (USB)\proj\main.py", line 433, in <module>
    pointsToDraw = physEng.predictOrbit(physObjects.sprites(), 100, screen) #calculate orbit lines
                                         ^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "e:\NEA (USB)\proj\engine.py", line 102, in predictOrbit
    self.calcApoapsisAndPeriapsis(pointArray, sprites, masters)
  File "e:\NEA (USB)\proj\engine.py", line 146, in calcApoapsisAndPeriapsis
    if pointDist < minDist: #if master object of sprite's distance to point is less than the minimum:
                                         ^^^^^^^^^^
TypeError: '<' not supported between instances of 'float' and 'pygame.math.Vector2'
PS E:\NEA (USB)\proj> 
```

This is due to me setting the point as the minimum distance if it is shorter, instead of the actual minimum distance itself. After adding two new variables for the actual points and fixing this, I got some cool patterns in the simulation from joining up all the apoapsis and periapsis lines, as I had not redone the draw function yet.



I then completed the draw function:

```
if buttonGroup.sprites()[0].isClicked: #if orbit lines are being shown (orbit lines are first in buttonGroup list)

    pointsToDraw = physEng.predictOrbit(physObjects.sprites(), 100, screen) #calculate orbit lines

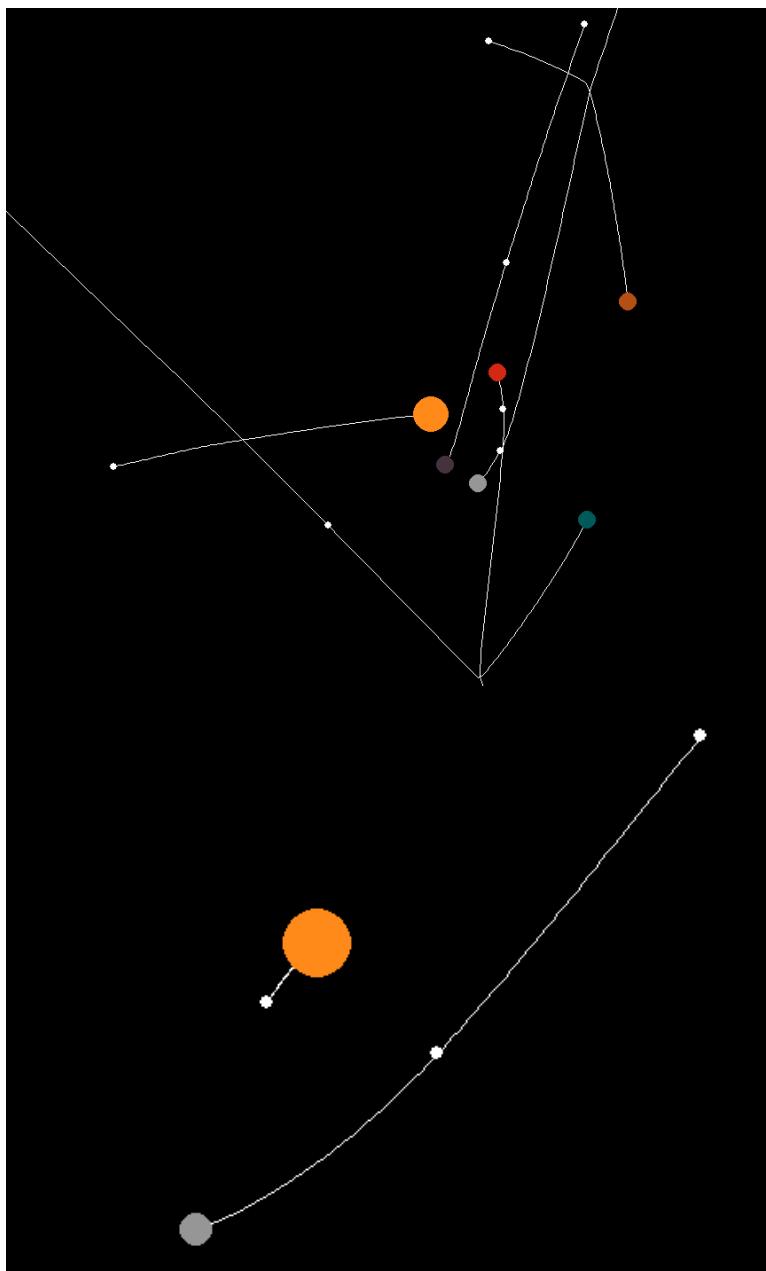
    for i in range(len(pointsToDraw) - 2): #for each object's orbit line (-2 for apoapsis and periapsis sections):

        pygame.draw.lines(screen, (255,255,255), False, pointsToDraw[i]) #draw orbit line

        #draw apoapsis markers
        for i in pointsToDraw[len(pointsToDraw) - 2]:
            pygame.draw.circle(screen, (255,255,255), i, 4) #draw marker

        #draw periapsis markers
        for i in pointsToDraw[len(pointsToDraw) - 1]:
            pygame.draw.circle(screen, (255,255,255), i, 4) #draw marker
```

And ran the program:



As shown, the objects have a point of closest approach to the object of largest gravitational force (the star in most cases), and a periapsis marker at point of furthest approach (as they are not in stable orbits, all are at the end of their respective orbit lines).

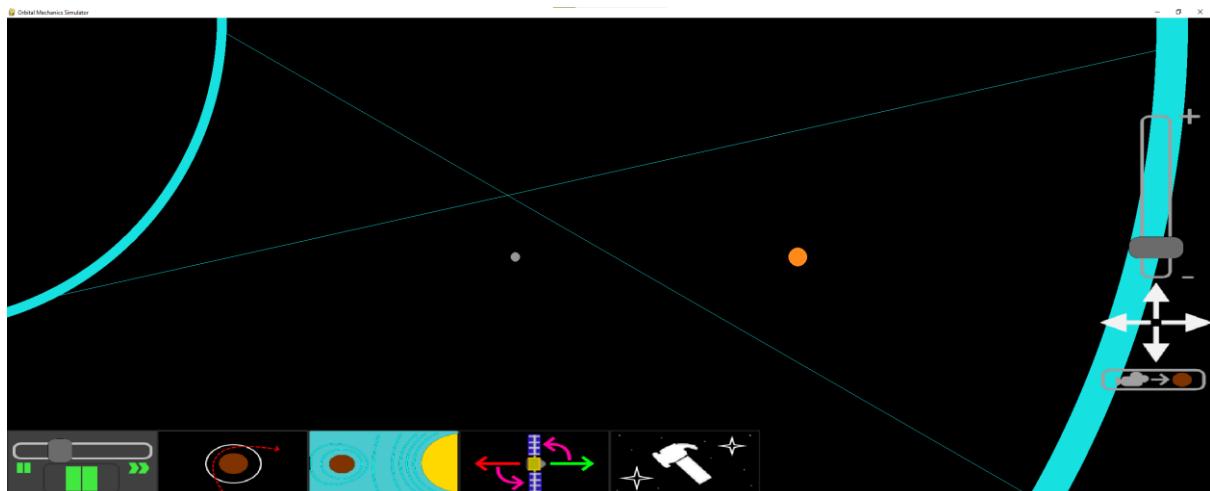
I removed all objects but the test planet and the star and ran it again to see it more clearly when the planet is in a stable state. At the point of closest approach, shown here, the apoapsis marker is present. At the point of furthest distance later on in the orbit, the periapsis marker is displayed in the correct place.

I changed the colour of the apoapsis and periapsis markers to blue and green respectively in order to better distinguish them.

Now for the equipotential lines. I will write the new `calcEquipotential()` method in the physics engine, which looks at every single integer point on the screen and calculates the force at that point from all physics objects. If the force is a certain value, the point is added to an array with other points with that force value to create the equipotential lines to be drawn. As an initial test, I will run the calculations and compare against the forces given in a temporary array, without the draw function having been implemented yet. This method is called in the main running loop if the equipotential button has been pressed, in a similar way to the orbit lines.

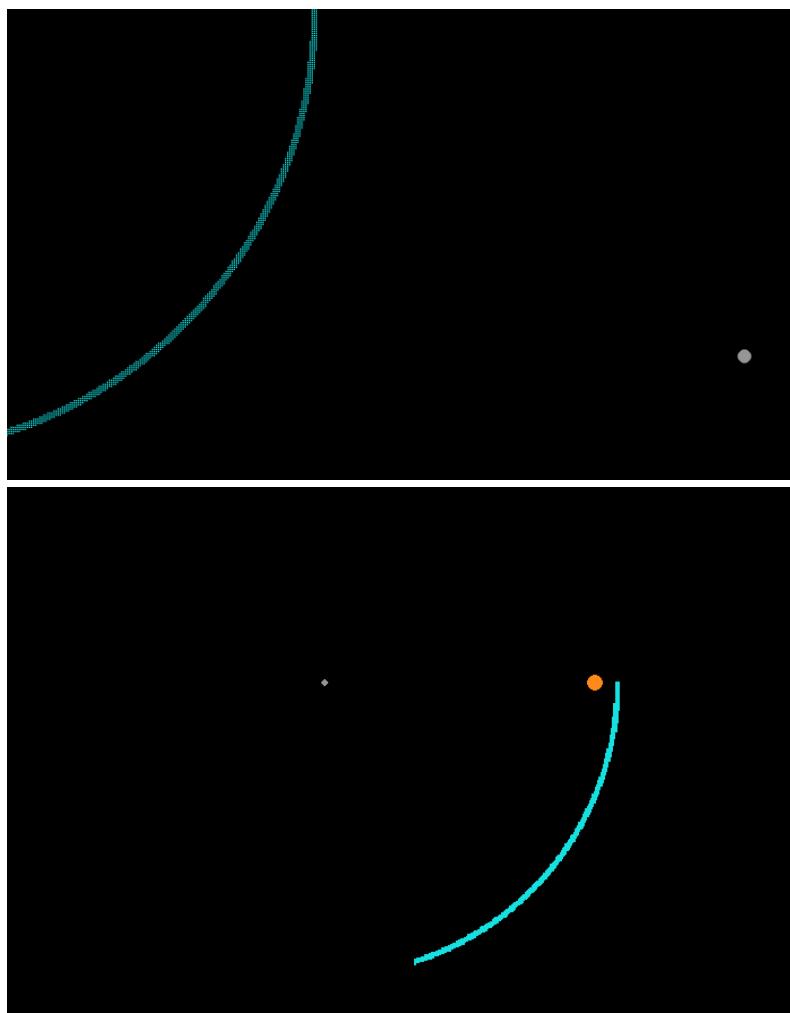
```
def calcEquipotential(self, objects, screen):  
  
    forcesToDraw = [5,10,15,20]  
  
    #for each point on the visible screen:  
    for i in range(screen.get_width()):  
        for j in range(screen.get_height()):  
  
            for k in objects: #for each physics object:  
                if self.calcGravBetweenObjects(k, k.gamePos.distance_to(Vector2(i,j))) in forcesToDraw:  
                    print(Vector2(i,j))  
  
    #add to 2d array based on force it meets, if is the same as one of the forces  
    #draw lines on screen  
    pass
```

When run, this did not print any points that met this level of force and also produced a noticeable level of lag. As the points being examined are only the integer points and not any points in between that may be floats, it is therefore unlikely that I will calculate any points that are exactly 15. To account for this, I will instead check if the calculated force is within a range of values. I printed the force calculated on each point, which returned a huge value; consequently, the force to be checked against is also increased to that range. I will also draw this one line of force to see how it looks initially and make changes accordingly.



This was the result; the blue lines should show the points where the gravitational force from all the objects are equal. The lines joining them across the screen are a result of the pygame.draw.lines method connecting all the points. While this has somewhat worked, it is an extremely slow process as it has to look at and calculate the force for every single pixel on the screen; on a 720p resolution screen, this is 921,600 calculations that must be completed per tick, which is horribly inefficient.

To account for this, I will decrease the number of calculations required by two thirds as I change the ‘for i in range’ line to skip every second and third value. Instead of using the Pygame line drawing method, I will instead try drawing a small point at the pixel where the equipotential has been calculated.



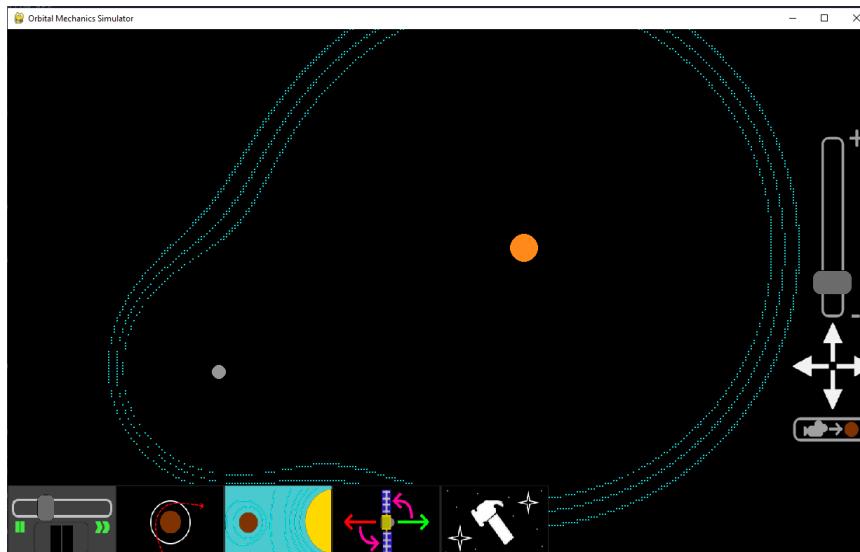
This increased performance by a fair amount and removed the lines crossing the screen, at the expense of the solidness of the lines. However, this is less of an issue; the lines are still easily made out.

One other issue is that the lines do not move with the camera, which can be easily fixed by passing the points through the `getScreenPos()` method.

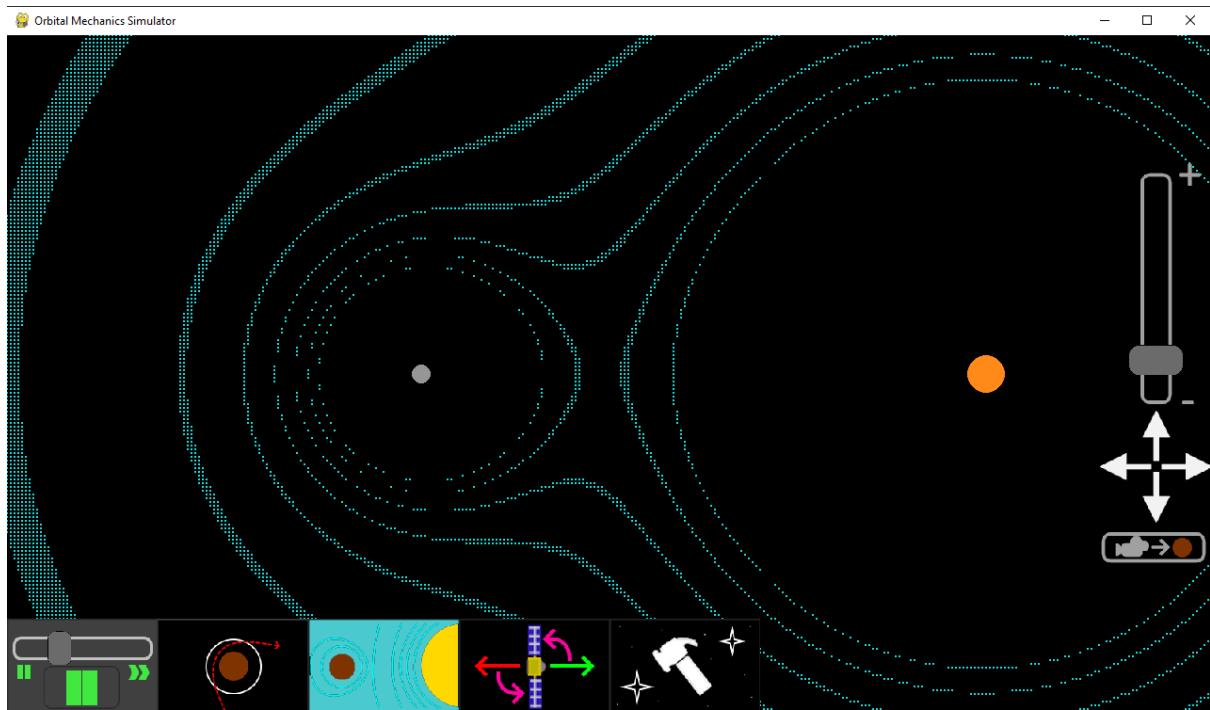
Now, however, the points are only present in the section that the camera looks at initially, and the rest of the lines are not recalculated when the camera moves into their view. This is because I am using the `gamePos`, not the `screenPos`, of each point and then converting it – I can instead just use the `screenPos`.

I then realized that I was not considering the force of all the

objects due to an indentation error in the `calcEquipotential()` method, which I promptly fixed. This was the next result:



After adding a few more regions of forces for which to draw the points, I ended up with this result:



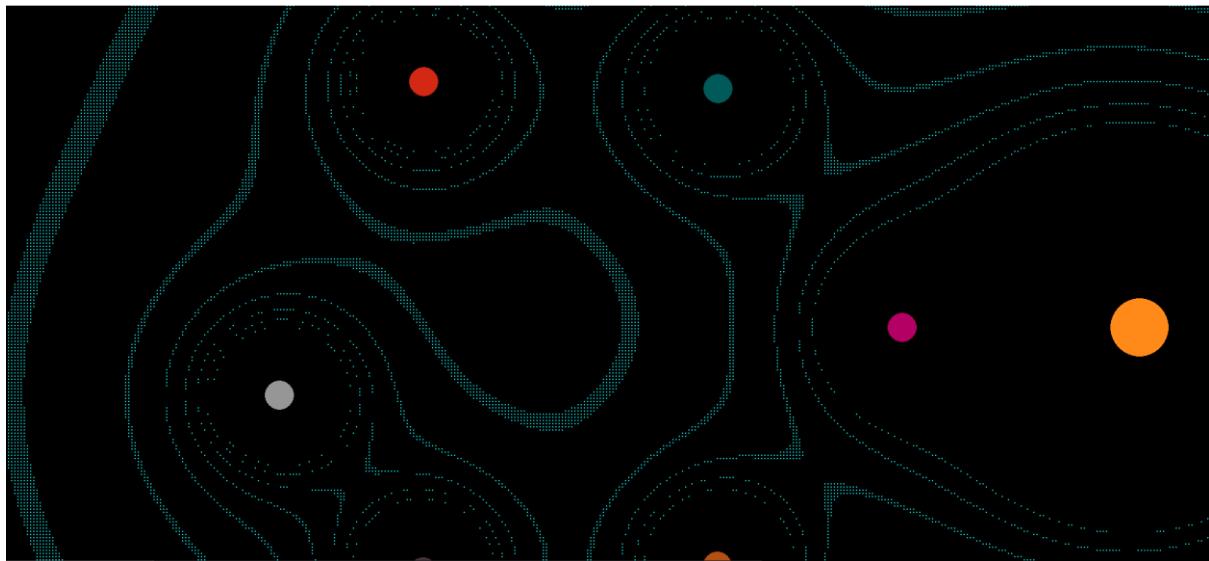
This was exactly what I wanted to achieve with these lines, and so I can consider it implemented.

Testing + Bug fixing

For many of the tests below, I used print statements to flag when certain methods were being called. To determine when a new tick started, I used the `clock.tick` line.

<i>Test #</i>	<i>Test description</i>	<i>Expected outcome</i>	<i>Actual outcome</i>	<i>Pass/fail</i>
<i>1a</i>	Orbit line drawing: show orbit lines, then unpause simulation	Orbit lines are calculated correctly and drawn on screen.	As expected.	Pass
<i>1b</i>	Orbit lines aren't calculated when not shown	<code>predictOrbit()</code> is never called when button untoggled.	Flag not triggered.	Pass
<i>1c</i>	Single call for orbit line calculations per tick	Method is called only once.	Flag triggered once per tick.	Pass
<i>1d</i>	Correct number of points calculated in future	<code>generatePointSet()</code> called the correct number of times.	Flag triggered the correct number of times per tick.	Pass
<i>2a</i>	Apoapsis and periapsis calculations: master sprite determined correctly for each object (7-object system)	Master sprite correctly determined for all objects, and orbit markers displayed as such.	As expected.	Pass
<i>2b</i>	Correct positions determined for apoapsis and periapsis markers	Apoapsis markers at closest points, periapsis markers at furthest points from master sprites	As expected.	Pass
<i>3a</i>	Equipotential lines: all coordinates on screen are examined when calculating lines	First point examined is (0,0), last point is	As expected.	Pass

		(screen width, screen height)		
3b	Equipotential lines aren't calculated when not shown	calcEquipotential() never called when button untoggled.	Flag not triggered.	Pass
3c	Single call for equipotential calculations per tick	Method is called only once.	Flag triggered once per tick.	Pass



Above are the equipotential lines for the 7-planet test system, which worked as intended.

Due to changes to the camera system, I also re-ran tests 2 and 3 from the previous development stage.

Test #	Test description	Expected outcome	Actual outcome	Pass/fail
2a	Zoom function: slow zoom in and then out to maximum. 6 objects in ring around centre, all keep consistent physics and spacing	All objects on screen move into the centre with zoom keeping consistency.	As expected.	Pass
2b	Move slider to maximum value	Value returned between 4.9 and 5.	4.9603174603174605	Pass
2c	Move slider to minimum value	Value returned between 0.28 and 0.32.	0.2976190476190487	Pass
3	Zoom/focus functions: Focus on each planet in turn while running. Zoom in and out	Planets and physics behave as predicted and consistently.	When an object is focused on and zoomed into past ~3.1x when simulation unpause, camera oscillates and is flung off to the side and software crashes	Fail

```
Traceback (most recent call last):
  File "e:\NEA (USB)\proj\main.py", line 480, in <module>
    physObjects.update() #apply results to objects + draw to screen
    ~~~~~~
  File "C:\Users\chick\AppData\Roaming\Python\Python311\site-packages\pygame\sprite.py", line 555, in update
    sprite.update(*args, **kwargs)
  File "e:\NEA (USB)\proj\main.py", line 274, in update
    self.rect = self.image.get_rect(center=self.screenPos) #update rect, centred at the planet's position
    ~~~~~~
TypeError: invalid rect assignment, expected value between -2147483648 < x < 2147483647
PS E:\NEA (USB)\proj>
```

Having the program crash is not acceptable, so to quickly fix the cause of the crash I will cap the screen position value to the boundary of these limits. However, this is more treating the symptoms of the error in the zoom function. After I complete more of the main features of the program, I will revisit this, but for now I will leave it as a known bug.

Stakeholder Feedback

Oli thought that it would be best to display a warning on the screen when the equipotential lines were active to advise users that it was very resource-intensive to calculate, and as such recommend keeping the simulation in a paused state. He also said there should be a setting to change the resolution of the equipotential lines, so that users would be able to trade between performance and line definition.

Mr Douglas-Scott was very impressed with the equipotential lines function, and said that it would be perfect for teaching an A level class.

Kai was otherwise happy with the overall solution.

Stage 4 – Object manipulation

Success criteria

REQUIREMENT	DESCRIPTION	SUCCESS CRITERIA	CHECK
1A	Mass manipulation	Ability to change the mass of objects in-game, and the effects of the change reflecting on the physics.	✓
1B	Speed manipulation	Ability to change the speed of objects in-game, and the effects of the change reflecting on the physics.	✓
1C	Direction of travel manipulation	Ability to change the direction of travel of objects in-game, and the effects of the change reflecting on the physics.	✓
1D	Time period manipulation (E)	Ability to change the time period of objects in-game, and the effects of the change reflecting on the physics.	✓
1F	Show information on objects	Labels on objects that give the user the basic information of it when selected.	✓
4D	Keep selected parameters from changing as the simulation runs	Ability to 'lock' variables of choice so that they do not change as the simulation runs.	✓

Testing plan

<i>Test #</i>	<i>Test description</i>	<i>Expected outcome</i>
1a	Click each object in turn.	selectedObject variable changes to become each object in turn.
2a	Single object in simulation: Open info panel.	Correct data displayed.
2b	Two objects in simulation: Open info panel for each object in turn.	Correct data displayed for each respective object.
3a	Mass manipulation: change value for mass of an object to 5 (normal).	Object's mass changes in panel, and change is reflected in physics.
3b	Change value to -5 (erroneous).	Object's mass does not change.
3c	Change value to 0 (boundary).	Object's mass is set to 0, and change is reflected in physics. (I am doing this test to check for any divide by 0 errors.)
3d	Change value to 1000 (extreme).	Object's mass changes in panel, and change is reflected in physics.
4a	Speed manipulation: change value for speed of an object to 5 (normal).	Object's speed and velocity change in panel, and change is reflected in physics.
4b	Change value to -5 (erroneous).	Object's speed and velocity do not change.
4c	Change value to 0 (boundary).	Object's speed and velocity become 0 in panel, and object stops moving in simulation.
4d	Change value to 1000 (extreme).	Object's speed and velocity change in panel, and change is reflected in physics.
5a	Velocity manipulation: change value for velocity of an object to 5 (normal).	Object's speed and velocity change in panel, and change is reflected in physics.
5b	Change value to -5 (erroneous).	Object's speed and velocity do not change.
5c	Change value to 0 (boundary).	Object's speed and velocity become 0 in panel, and object stops moving in simulation.
5d	Change value to 1000 (extreme).	Object's speed and velocity change in panel, and change is reflected in physics.
6a	Time period manipulation: change value for time period of an object to 5 (normal).	Object's time period changes in panel, and change is reflected in physics.
6b	Change value to -5 (erroneous).	Object's time period does not change.
6c	Change value to 0 (boundary).	Object's time period becomes 0 in panel.
6c	Change value to 1000 (extreme).	Object's time period changes in panel, and change is reflected in physics.
7a	Dual planet system: lock velocity of one planet.	Velocity of locked object does not change, whereas velocity of unlocked object does.
7b	Dual planet system: lock speed of one planet.	Speed of locked object does not change, whereas speed of unlocked object does.
7c	Surface gravity manipulation: change value to 5 (normal).	Mass and radius change accordingly, with equal weighting on both.
7d	Lock mass and change value to 5.	Radius changes accordingly, with mass remaining the same.
7e	Lock radius and change value to 5.	Mass changes accordingly, with radius remaining the same.
7f	Lock both mass and radius and change value to 5.	All values do not change.

- | | | |
|----|---|---|
| 8a | Increase equipotential line resolution. | Resolution increases by one step per click. |
| 8b | Decrease equipotential line resolution. | Resolution decreases by one step per click. |

Development

To create a way to increase the resolution of the equipotential lines as per stakeholder feedback, I will need to create two buttons that will appear above the equipotential toggle button when it is activated; these will be designations 10 and 11. I will also need to create another sprite group, equipotentialResolutionButtonGroup, to contain these two buttons so that I can draw them only when the equipotential lines are toggled on.

```
equipotentialResolutionButtonGroup.add(Button(10)) #create equipotential resolution buttons
equipotentialResolutionButtonGroup.add(Button(11))
```

I then wrote the code to draw the lines if the equipotential button was toggled on:

```
if buttonGroup.sprites()[1].isClicked: #if equipotential lines are being shown:
    equipotentialResolutionButtonGroup.draw(screen)

    equipotentialLines = physEng.calcEquipotential(physObjects, screen) #get array
    for i in equipotentialLines: #for each point on equipotential line:

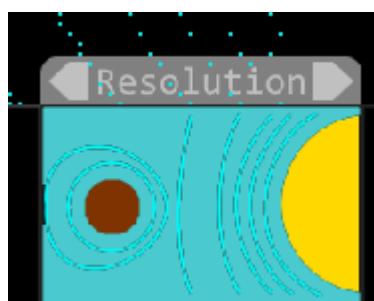
elif event.type == pygame.MOUSEBUTTONDOWN: #if the mouse has been clicked (or interactive whiteboard tapped)
    buttonGroup.update() #run checks for buttons

    if selectedObject != 0: #if an object is currently selected:
        sidePanelButtonGroup.update()

    if buttonGroup.sprites()[1].isClicked: #if equipotential lines are being shown:
        equipotentialResolutionButtonGroup.update()

    for i in physObjects: #for each physics object:
        i.checkIfSelected() #run checks to see if selected

    #print(pygame.mouse.get_pos()) #debug
```



This worked apart from the fact that the equipotential lines were being drawn on top of the button; easily fixed by moving the .draw method to after the lines are drawn.

I then added the buttons to the rescale() method and created their collisions.

To add functionality, I will create a new global variable called resolution which will be incremented or decremented each time

one the corresponding resolution button is clicked. The maximum resolution is 1, whereas the minimum resolution will be 10. In the effect() method for each of the buttons, a check will be

```
case 10:
    if resolution != 10: #if resolution not minimum:
        resolution += 1 #increment resolution

case 11:
    if resolution != 1: #if resolution not maximum:
        resolution -= 1 #decrement resolution
```

Made to see whether the value is at its maximum/minimum, and if not will change the value accordingly.

This resolution value is then passed as a parameter through the calcEquipotential() method, to

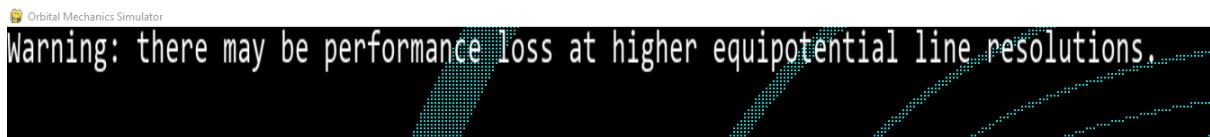
determine how many pixels on the screen are skipped to draw the equipotential lines.



I now have the two buttons implemented, and I now needed a way to display to the user what the current resolution is, while also warning them of the performance loss at higher resolutions. To do this, I will write text in the top left of the screen when the equipotential lines are active and below 5 resolution.

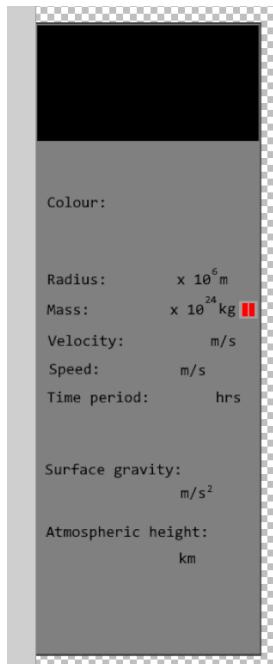
```
if resolution < 5:  
    screen.blit(font.render("Warning: there may be performance loss at higher equipotential line resolutions.", True, (255,255,255)), (0, 0))
```

When this is blit to the screen, it appears large and cut off; to fix this, I will need to use the `Pygame.transform.smoothscale()` method to scale the text to the size of the screen.



This is now working as intended.

One of the main features of this program is the ability to change the parameters of objects as the simulation is running. To do this, I will create an info box that appears along the left side of the screen when a certain object is clicked, as planned in the design.



I originally planned to create the background for this in the same way as the menu background from stage 1, as a separate image; however, after my experience with creating the slider background as part of the focus button, I decided that would be the best option. The buttons present in the info box consist of lock buttons to freeze variables (for requirement 4D), and so I will just create the background as part of the image of one of those. This is shown in the image on the left.

The actual button to freeze the variables (only the mass one shown here, but I will require one each for velocity, speed and time period) is the small square to the right of the variable. The example on the left is one such button in its clicked state.

In order to show and hide the buttons, I will add them to a separate `Pygame.sprite.group` (called `sidePanelButtonGroup`) for easy control over when they are drawn. This group's draw method will be called in the main running loop if there is an object currently selected, and I will make sure that the group rescales at the same time as the other button group.

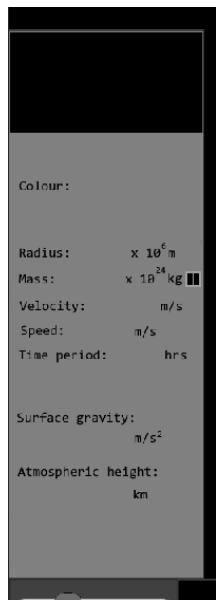
To tell whether or not any object is selected, I will use a global `selectedObject` variable in the main program; if this is not 0, then the side panel and the respective buttons will be drawn. When the mouse button is clicked, I will call a method in each planet class to check the position of the mouse cursor against the rect of the object; if they intersect, then the object will set itself as the `selectedObject`.

While I was originally intending for this to be part of the planet update() method, it would be more efficient to make all of the checks for intersection after a mouse press happen in a separate method called by the Pygame events system. This will be called checkIfSelected().

```
def checkIfSelected(self): #if mouse button has been pressed:  
  
    global selectedObject  
  
    if self.rect.collidepoint(pygame.mouse.get_pos()): #if mouse is over object:  
        print("flag")  
        selectedObject = self #set self as the selected object
```

This is called in the Pygame event for mouse button down, which loops over each physics object.

I ran the program to test whether the panel appeared when any physics object was clicked, and it displayed and resized correctly. However, I was not so happy with the readability of the text on the panel.



This was what was displayed when the program was run in its default 720p resolution. I decided that pasting the text as part of the background was probably not the best idea, especially for scaling, so I removed this from the background and instead decided to write the text using the Pygame font system.

To scale the font with the screen size, I will call the Pygame.transform.smoothscale() method once the font has been rendered into a surface to scale it before it is drawn to the screen.

Each variable displayed in the info box will require its own separate text surface; consequently, in order to avoid repeating the same process while scaling them all to the same size, I will add each surface to a list that will loop over them all in order to scale and draw them.

I then wrote the code to display the mass and the colour of the object to test inside the main running loop:

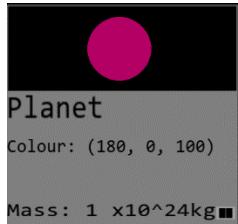
```
if selectedObject != 0: #if an object is currently selected:  
  
    sidePanelButtonGroup.draw(screen) #draw info panel to screen  
  
    textBuffer = [0 for i in range(2)] #generate array for text surfaces  
  
    textBuffer[0] = font.render(f"Colour: {selectedObject.colour}", True, (0,0,0)) #render font for colour  
    textBuffer[1] = font.render(f"Mass: {selectedObject.mass} x 10^24", True, (0,0,0)) #render font for mass  
  
    for i in textBuffer: #for each rendered text surface:  
        i = pygame.transform.smoothscale(i, (screen.get_width()*0.13, screen.get_height()*0.03)) #rescale  
  
    screen.blit(textBuffer[0], (0,screen.get_height()*0.3)) #blit colour to screen  
    screen.blit(textBuffer[1], (0,screen.get_height()*0.5)) #blit mass to screen
```

I use the textBuffer array to hold the surfaces, as it is more straightforward than using many separate variables. I rescale all of the surfaces in the same way before drawing them at the correct positions.



This displays the text with the correct values, but it seems like the scaling method is not working as intended. When the window is resized, the text does not change in size to fit it and is instead cut off.

I suspect that this is due to the *i* in the scaling for loop is not changing the variable inside the array as it is rescaled. To get around this, I rewrote *i* to become the index of the array instead of the surface itself, so that the surface at the index value would be changed instead of *i*. This solved the issue, and the text scaled correctly.



I then added the other variables, and an image of the selected object in the window at the top of the info box. To differentiate between stars and planets, I also added the line to state the class of the object. I also needed to make the planet in the box scale with the resizing of the window; if resized too large, the planet became too small and vice versa. This was fixed by making the radius command a multiple of the screen width. This should be a satisfactory solution as it is very unlikely that the window will be resized such that there is a large difference between the window's width and height. To further confirm this, I tested the scaling on a 2560x1080 ultrawide display, during which the object in the info box scaled correctly.

'Speed' is used only for the info box, with the purpose of illustrating the speed of an object in a scalar form. As such, I have not implemented it until now, where I will calculate it using the magnitude feature of the Pygame Vector2 data structure.

Initially, my code looked like this, but it was optimised easily:

```
textBuffer = [0 for i in range(6)] #generate array for text surfaces

textBuffer[0] = font.render(str(selectedObject.__class__.__name__), True, (0,0,0)) #render font for class name (planet/star)
textBuffer[1] = font.render(f"Colour: {selectedObject.colour}", True, (0,0,0)) #render font for colour
textBuffer[2] = font.render(f"Mass: {selectedObject.mass} x10^24kg", True, (0,0,0)) #render font for mass
textBuffer[3] = font.render(f"Radius: {selectedObject.radius} x10^6m", True, (0,0,0)) #render font for radius
textBuffer[4] = font.render(f"Velocity: {selectedObject.velocity} m/s", True, (0,0,0)) #render font for velocity
textBuffer[5] = font.render(f"Speed: {round(selectedObject.velocity.magnitude(), 2)} m/s", True, (0,0,0)) #render font for speed

for i in range(1, len(textBuffer)): #for each rendered text surface (excludes class name):
    textBuffer[i] = pygame.transform.smoothscale(textBuffer[i], (screen.get_width()*0.13, screen.get_height()*0.03)) #rescale
textBuffer[0] = pygame.transform.smoothscale(textBuffer[0], (screen.get_width()*0.06, screen.get_height()*0.05)) #rescale

screen.blit(textBuffer[0], (0, screen.get_height()*0.28)) #blit class name to screen
screen.blit(textBuffer[1], (0, screen.get_height()*0.35)) #blit colour to screen
screen.blit(textBuffer[2], (0, screen.get_height()*0.45)) #blit mass to screen
screen.blit(textBuffer[3], (0, screen.get_height()*0.48)) #blit radius to screen
screen.blit(textBuffer[4], (0, screen.get_height()*0.51)) #blit velocity to screen
screen.blit(textBuffer[5], (0, screen.get_height()*0.54)) #blit speed to screen
```

The optimised code:

```
textBuffer = [0,0,0,0,0,0] #generate array for text surfaces

textBuffer[0] = font.render(str(selectedObject.__class__.__name__), True, (0,0,0)) #render font for class name (planet/star)
textBuffer[1] = font.render(f"Colour: {selectedObject.colour}", True, (0,0,0)) #render font for colour
textBuffer[2] = font.render(f"Mass: {selectedObject.mass} x10^24kg", True, (0,0,0)) #render font for mass
textBuffer[3] = font.render(f"Radius: {selectedObject.radius} x10^6m", True, (0,0,0)) #render font for radius
textBuffer[4] = font.render(f"Velocity: {selectedObject.velocity} m/s", True, (0,0,0)) #render font for velocity
textBuffer[5] = font.render(f"Speed: {round(selectedObject.velocity.magnitude(), 2)} m/s", True, (0,0,0)) #render font for speed

for i in range(1, 6): #for each rendered text surface (excludes class name):
    textBuffer[i] = pygame.transform.smoothscale(textBuffer[i], (screen.get_width()*0.13, screen.get_height()*0.03)) #rescale

    if i == 1: #if colour text (separate to add spacing):
        screen.blit(textBuffer[1], (0, screen.get_height()*0.35)) #blit colour to screen
    else:
        screen.blit(textBuffer[1], (0, screen.get_height()*(0.425 + 0.03*(i - 1)))) #blit text to screen

textBuffer[0] = pygame.transform.smoothscale(textBuffer[0], (screen.get_width()*0.06, screen.get_height()*0.05)) #rescale class name text
screen.blit(textBuffer[0], (0, screen.get_height()*0.28)) #blit class name to screen
```

I have also added a line to set the selected object to an object if the focus switches to that object.

Another value I would like to display is the surface gravity on the object, calculated using the `calcGravAtSurface()` method in the planet class itself that I coded back in stage 1. This is added in the same way.

After these variables are implemented, I can now position the buttons for freezing the variables next to them correctly. To do this, I will get a screenshot of the program in its default resolution before pasting it into a new paint.net file as the background; I will then create a new layer and draw the button on top of that to ensure my positioning is exactly where I need it to be. I have also decided to merge the speed and velocity lock buttons, as they are calculated based on each other; if one is locked, then the other must also be.

Another button I need is one to close the info panel if clicked; if the user wants to focus on an object and not have the info panel taking up part of the screen, this is vital. It will be positioned in the top right corner of the info panel.

I then realised that mass and radius do not need a lock button as they are not affected by any other factors. I also realised this in my design, so I'm not sure what I was doing here. For now, then, the only locking button present will be the merged speed/velocity button.

To add functionality to the buttons, I will set up their collision rects in the button `rescale()` method. In the `effect()` method, the close info box button will simply set the `selectedObject` variable to 0, while the freeze speed/velocity button will set a flag inside the relevant object (an internal Boolean variable called `velocityFreezeFlag`) that will be checked by the physics engine every update tick; if this flag is present, it will not calculate a change in velocity for the object.

```
case 25: #freeze velocity/speed button
    selectedObject.velocityFreezeFlag = self.clicked #set flag in object
```

Changes had to be made to the physics engine `update()` method to accommodate this:

```
if not self.isPaused: #physics calculations: do not run when simulation is paused

    for i in sprites: #for each sprite in group:

        if not i.velocityFreezeFlag: #if velocity of object has not been frozen:
            j = 0
            while j < len(sprites): #for each other sprite:

                if not sprites[j] is i:

                    i.velocity += Vector2((sprites[j].gamePos[0]-i.gamePos[0])/math.sqrt((sp
                    j += 1
```

Running to test, both buttons work as intended after some tweaking of the collision rects. However, if the selected object is changed while another one is frozen, the state of the freeze button is kept, and has to be clicked twice to change the state of the freeze flag for that new object. To fix this, I will try setting the state of the freeze button to the state of the freeze flag in an object when it is selected.

```
def checkIfSelected(self): #if mouse button has been pressed:

    global selectedObject #summon the great and powerful global variable

    if self.rect.collidepoint(pygame.mouse.get_pos()): #if mouse is over object:

        selectedObject = self #set self as the selected object
        buttons = sidePanelButtonGroup.sprites()

        print(self.velocityFreezeFlag)

        buttons[1].isClicked = self.velocityFreezeFlag #match velocity freeze button to object flag

        if self.velocityFreezeFlag:
            buttons[1].imageUnscaled = buttons[1].imageClicked
        else:
            buttons[1].imageUnscaled = buttons[1].imageUnclicked
```

However, this did not display the correct state of the button. I knew that I was making the changes to the correct button as sometimes the changes did happen, but the button would switch to the wrong state. I also need to consider the orbit line prediction of an object when its velocity has been frozen. These were fixed by implementing a check similar to the one in the physics engine update() method to the generatePointSet() method, and by rescaling the button after showing it in the side panel. This fixed the issues, and the feature now works.

Another feature that I would like to add is an indication of which object the selected object is currently orbiting; this will allow for the time period to be calculated using that object, and the orbit speed can also be found.

The object that is being orbited can be found from the master sprite list created within the predictOrbit() method; the issue with this is that this method is not called if the orbit lines are not being shown.

The simplest way that I can achieve this is by calling the generatePointSet() method from the physics engine separately in the main running loop, passing through a point level of just 1, which will return the master sprite list for any object's current position. However, since I only need the master sprite of the selected object, I can run these calculations directly in the main running loop with the calcGravBetweenObjects() method to find which exerts the most force on the selected object.

```
if selectedObject != 0: #if an object is currently selected:

    #determine master object
    greatestForce = 0 #set initial greatest force as 0
    masterObject = selectedObject #set master object as itself for time being
    for j in range(len(physicsSprites)): #for each other sprite in group:

        if selectedObject is not j: #skip if object is the same as the object calculating force for

            force = physEng.calcGravBetweenObjects(physicsSprites[j], physicsSprites[j].distance_to(selectedObject) * 1e6)

            if force > greatestForce: #if this is new greatest force:

                greatestForce = force #set new greatest force
                masterObject = physicsSprites[j] #set master object as object force was calculated from
```

Now that I have the master sprite, I can calculate the time period of the selected object as if it were in an orbit around the master object. This is done using the equation $T = \frac{2\pi r^{\frac{3}{2}}}{\sqrt{GM}}$, where G is the gravitational constant and M is the mass of the master object. This gives T, time period, in seconds,

which can be divided by 3600 to get it in hours. I decided to both perform the calculation and render the font in the same line to save using another variable to split it.

```
textBuffer[7] = font.render(f"Time period:  
{(2*math.pi*(selectedObject.gamePos.distance_to(masterObject.gamePos)**(3/2)))/(3600*math.sqrt(6.67e-11*masterObject.correctedMass))} hrs",  
True, (0,0,0)) #calculate time period and render to font
```

I also created and positioned a new variable freezing button for time period, designated 26. I will add the functionality for it later.

Another two parameters that I can display are the distance to the master sprite, aka the orbital radius, and the orbital velocity. However, the info panel was now getting very cluttered with all of these variables. I had two ideas to fix this: one was to create a new info box to the side to hold the orbit information, and the second was to create two new buttons on the info box to swap ‘pages’; one would be for the selected object’s information, another would be for the information about the orbit the selected object was in. I decided to go with the second option, as a new box would only make the program more cluttered.

I did not like my first iteration of these buttons (the images on the left), so I changed them to the style of the image on the right.



These buttons have been saved as designations 27 and 28 respectively.

To create the second page, I need to create another group to control when the variable frozen buttons are drawn on the second page, which I will call sidePanelOrbitButtons. To match, I renamed the sidePanelButtonGroup to sidePanelObjectButtons.

To keep track of which page is being displayed, I will create yet another global variable as an int called sidePanelPage. I am using an int here so that if I do need to add more pages in the future, it will be made easier as there will be more than two states that the variable could be in.

All of the information related to the master object will be displayed on the orbit info page of the side panel; consequently, if the object page is selected, I will not need to determine the master object, and so I can move the calculations for it under the check for the page number, where I draw the sidePanelOrbitButtons.

```
if selectedObject != 0: #if an object is currently selected:  
  
    sidePanelCloseButton.draw(screen) #draw info panel to screen  
  
    if sidePanelPage == 0: #object info page  
  
        sidePanelObjectButtons.draw(screen)  
  
    elif sidePanelPage == 1: #orbit info page  
  
        sidePanelOrbitButtons.draw(screen)  
  
    #determine master object  
    greatestForce = 0 #set initial greatest force as 0  
    masterObject = selectedObject #set master object as itself for time being  
    for j in range(len(physicsSprites)): #for each other sprite in group:  
  
        if selectedObject is not j: #skip if object is the same as the object calculating force for  
  
            force = physEng.calcGravBetweenObjects(physicsSprites[j], physicsSprites[j].gamePos.distance_to(selectedObject.gamePos) * 1e6)  
  
            if force > greatestForce: #if this is new greatest force:  
  
                greatestForce = force #set new greatest force  
                masterObject = physicsSprites[j] #set master object as object force was calculated from
```

To keep the close panel button's functionality on both pages, I will add it to its own group (Pygame sprites need to be in groups to be drawn, otherwise I would draw it directly) and draw it when needed as it is the only sprite that is present on both pages.

I also split the rendering for the text to draw to the screen into their respective pages. I kept the rescaling code outside of the check loop and changed it to run for the length of the textBuffer array.

```
if sidePanelPage == 0: #object info page
    textBuffer = [0,0,0,0,0,0] #generate array for text surfaces
    sidePanelObjectButtons.draw(screen)

    textBuffer[2] = font.render(f"Mass: {selectedObject.mass} x10^24kg", True, (0,0,0)) #render font for mass
    textBuffer[3] = font.render(f"Radius: {selectedObject.radius} x10^6m", True, (0,0,0)) #render font for radius
    textBuffer[4] = font.render(f"Velocity: {selectedObject.velocity * 1000} km/h", True, (0,0,0)) #render font for velocity, converted to kilometres per second
    textBuffer[5] = font.render(f"Speed: {round(selectedObject.velocity.magnitude()) * 1000, 1}) km/h", True, (0,0,0)) #render font for speed, converted to kilometres per second
    textBuffer[6] = font.render(f"Surface gravity: {selectedObject.surfaceGrav} m/s^2", True, (0,0,0)) #render font for surface gravity

elif sidePanelPage == 1: #orbit info page
    sidePanelOrbitButtons.draw(screen)

#determine master object
greatestForce = 0 #set initial greatest force as 0
masterObject = selectedObject #set master object as itself for time being
for j in range(len(physicssprites)): #for each other sprite in group:
    if selectedObject is not j: #skip if object is the same as the object calculating force for
        force = physEng.calcGravBetweenObjects(physicssprites[j], physicssprites[j].gamePos.distance_to(selectedObject.gamePos) * 1e6) #calculate force between objects
        if force > greatestForce: #if this is new greatest force:
            greatestForce = force #set new greatest force
            masterObject = physicssprites[j] #set master object as object force was calculated from

    textBuffer = [0,0,0,0] #generate array for text surfaces

    textBuffer[2] = font.render(f"Time period: {(2*math.pi*(selectedObject.gamePos.distance_to(masterObject.gamePos)**(3/2)))/(math.sqrt(6.67e-11*masterObject.correctedMass))} hrs", True, (0,0,0))
    textBuffer[3] = font.render(f"Orbital radius: {round(selectedObject.gamePos.distance_to(masterObject.gamePos), 1)} x10^6m", True, (0,0,0))
```

```
case 27:
    sidePanelPage = 1
case 28:
    sidePanelPage = 0
```

Running to test, the information and sprites on the object info page are drawn correctly. To add functionality to the page movement buttons, I made their effects increment and decrement the sidePanelPage variable.

The two pages are now complete and work as intended. However, the value for time period in hours is very small and so squishes the text. To fix this, I will change the unit that the time period is measured in based on what its value is. This now all works as intended.

```
timePeriod = (2*math.pi*(selectedObject.gamePos.distance_to(masterObject.gamePos)**(3/2)))/(math.sqrt(6.67e-11*masterObject.correctedMass))

#determine units for time period
if timePeriod < 300: #if time period is less than 5 minutes:
    timePeriod = str(timePeriod) + " secs" #use seconds as units
elif timePeriod < 10800: #if time period is less than 3 hours:
    timePeriod = str(timePeriod/60) + " mins" #convert and use minutes as units
elif timePeriod < 259200: #if time period is less than 3 days:
    timePeriod = str(timePeriod/3600) + " hrs" #convert and use hours as units
else:
    timePeriod = str(timePeriod/86400) + " days" #convert and use days as units

textBuffer[2] = font.render(f"Time period: " + timePeriod, True, (0,0,0)) #calculate time period and render to font
textBuffer[3] = font.render(f"Orbital radius: {round(selectedObject.gamePos.distance_to(masterObject.gamePos), 1)} x10^6m", True, (0,0,0))
```

To calculate orbital velocity, I can use the equation $v = \sqrt{\frac{GM}{r}}$; this is the velocity perpendicular to the direction of the master object's gravitational field that an object must have in order to be in orbit around that object. To make this easier to calculate, I can just take the force returned by the calcGravBetweenObjects() method for the master object, multiply it by the radius once and then take the square root of that.

```
textBuffer[2] = font.render(f"Time period: " + timePeriod, True, (0,0,0)) #calculate time period and render to font
textBuffer[3] = font.render(f"Orbital radius: {round(selectedObject.gamePos.distance_to(masterObject.gamePos), 1)} x10^6m", True, (0,0,0)) #render font for
textBuffer[4] = font.render(f"Orbital velocity: {round(math.sqrt(force*selectedObject.gamePos.distance_to(masterObject.gamePos)), 2)} km/h", True, (0,0,0))
```

One thing I have to take into account is the difference between the star and planet objects; while they share many variables, the star object's colour is determined based on its own temperature

variable. As such, when displaying a star, its surface temperature should also be displayed on the object info page.

```
if selectedObject is Star:  
    textBuffer.append(font.render(f"Surface temperature: {selectedObject.temperature}K", True, (0,0,0))) #render font for surface temperature
```

I use append to do this as otherwise the scaling code will return an error for trying to rescale an integer (the default 0) instead of a rendered font surface. Running to test, the star object does not show the temperature variable when it is selected. I tested again with a flag, which showed me that the check was not registering that the object was a star. To get around this, I instead checked the `selectedObject.__class__.name__` to see if that was a star, as this is what is returned when rendering the font for the type of the object. The text now displayed correctly.

All of the variables I would like to display have been rendered; I will now finalise the buttons to freeze these variables while the simulation is running. In terms of the object parameters, the only variables to freeze are the velocity and speeds, which have been implemented. In terms of the selected object's orbit, the variables that should be able to be frozen are the time period, the orbital radius and the orbital velocity.

```
self.velocityFreezeFlag = False  
self.timePeriodFreezeFlag = False  
self.orbitalRadiusFreezeFlag = False  
self.orbitalVelocityFreezeFlag = False
```

First in the implementation of these is creating a flag for each in the planet class, called `timePeriodFreezeFlag`, `orbitalRadiusFreezeFlag` and `orbitalVelocityFreezeFlag` respectively.

I then need to create and position the freeze buttons for each of the variables. These will be placed on the orbit page of the side panel, and so will be part of the `sidePanelOrbitButtons` group. This will also mean that I will need to make button designation 26, the time period freeze button, part of this group. The issue with this is that it is in the middle of the sequence for the `sidePanelObjectButtons` group; to fix this, I will change its designation to 29 while changing the designation of the velocity freeze button to 26 to allow for the instantiation of the buttons to stay as a single for loop for each group.

```
objectButtons[0].isClicked = self.velocityFreezeFlag #match velocity freeze button to object flag  
orbitButtons[1].isClicked = self.timePeriodFreezeFlag #do the same for time period  
orbitButtons[2].isClicked = self.orbitalRadiusFreezeFlag #and orbital radius  
orbitButtons[3].isClicked = self.orbitalVelocityFreezeFlag #and orbital velocity  
  
if self.velocityFreezeFlag: #set image of the button as the correct state  
    objectButtons[0].imageUnscaled = objectButtons[0].imageClicked  
else:  
    objectButtons[0].imageUnscaled = objectButtons[0].imageUnclicked  
objectButtons[0].rescale()  
  
if self.timePeriodFreezeFlag: #set image of the button as the correct state  
    orbitButtons[1].imageUnscaled = orbitButtons[1].imageClicked  
else:  
    orbitButtons[1].imageUnscaled = orbitButtons[1].imageUnclicked  
orbitButtons[1].rescale()  
  
if self.orbitalRadiusFreezeFlag: #set image of the button as the correct state  
    orbitButtons[2].imageUnscaled = orbitButtons[2].imageClicked  
else:  
    orbitButtons[2].imageUnscaled = orbitButtons[2].imageUnclicked  
orbitButtons[2].rescale()  
  
if self.orbitalVelocityFreezeFlag: #set image of the button as the correct state  
    orbitButtons[3].imageUnscaled = orbitButtons[3].imageClicked  
else:  
    orbitButtons[3].imageUnscaled = orbitButtons[3].imageUnclicked  
orbitButtons[3].rescale()
```

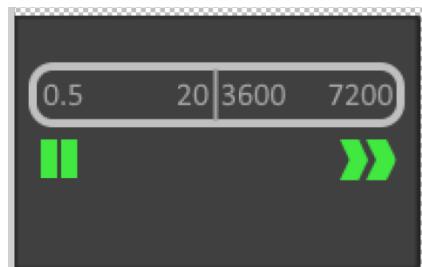
Now I need to implement the text boxes to change the values of certain variables in the objects. The first step to doing this is to finalise the scale for this simulation; currently, the units for speed and velocity are placeholders as I have not determined the actual unit of speed. Based on a 1x time scale, this turns out to be megametres per second (Mm/s). For simplicity I converted this just for the purpose of displaying it as kilometres per second, as it will be much more familiar to students.

However, now the objects have motion that is not reflective of real-life celestial movement; this will make it difficult to get a correct value for any time period that is realistic. To fix this, I need to divide each velocity value by 3600 when applying to objects. This turns the units from kilometres per second to kilometres per hour, which will make it much closer to reality. To accommodate for this, I also need to change the time scaling factor to allow for a much greater range, otherwise there will be no way to observe the movement of the objects in a feasible time.

I changed the maximum value of the time scale slider to be 10800x, which should be the equivalent to the previous 4x scale. This functioned correctly, but it was not possible to reset the scale back to 1x. It also broke the orbit prediction system slightly, which can be fixed by dividing the time scale in that calculation by 3600 as well.

To fix the issue of not being able to go back to 1x, I will split the time scale slider into two sections: one for real-time scales (1-20x), and one for celestial scales (3600-7200x). This will make the slider more sensitive to changes however, so I will need to test it and get feedback to see whether this is actually a feasible solution.

Implementing this and running to test, the 0-20x side of the slider worked fine, but when the mouse was moved over to the celestial time scale side, the time scale value was double what was expected, and when the mouse button was released the slider button moved to double the distance away from the centre of the slider than where it was meant to be.



I halved both the slider positioning and the zoom scale return values, which fixed the position of the slider but doubled the scale value again. To fix this, therefore, I will have to multiply the scale value by 2. After doing this, the slider now works as expected. To better show the difference between real and celestial scale, I will redraw parts of the time scale slider background to show the maximum and minimum values.

Now that the scales have been changed, I can finally create the Textbox class so that users can change the parameters of objects directly while the simulation is running. I will be using a similar designation system to the button class to position and differentiate between each text box object.

```
class Textbox(pygame.sprite.Sprite):

    def __init__(self, designation):
        pygame.sprite.Sprite.__init__(self) #initialise Pygame sprite

        self.text = "" #set initial empty text
        self.active = False #set initial state as inactive
        self.designation = designation

        self.clickedColour = (172,172,172) #grey
        self.unclickedColour = (255,255,255) #white

        self.image = pygame.Surface([screen.get_width()*0.13, screen.get_height()*0.03]) #create surface
        self.rect = self.image.get_rect() #set initial rect
```

I will then use two variables, clickedColour and unclickedColour, to store RGB values for the box when it is clicked and unclicked. As the box itself is just a single colour, I will be using this method instead of creating a different image for the clicked and unclicked states as with the buttons.

To display the textboxes, I will need to create two new sprite groups; one for the object section of the side information panel and one for the orbit section. These will be called textboxObjectGroup and textboxOrbitGroup respectively. Like with the buttons, I realised that I would also need a resize() method in order to resize the text boxes if the window was resized. The call for this will replace the image and rect setting lines in the above method.

```
def rescale(self):
    self.image = pygame.Surface([screen.get_width()*0.13, screen.get_height()*0.03]) #create surface
    self.rect = self.image.get_rect() #set initial rect
```

The data to be displayed on the textboxes will be that of the selected object; as such, I will need to use the update() method to set the text values of each box to that of the corresponding variables. To determine which value to take, I will use another match...case statement. I have moved the calculations for variables such as time period here, as I am replacing the text used previously to display these with the textboxes themselves.

```
def update(self, masterObject): #called per tick to update values to that of the selected object
    global selectedObject

    if self.active: #if textbox is active:
        return #do not update textbox while value is being changed

    match self.designation:
        case 1: #mass
            self.text = str(selectedObject.mass)

        case 2: #radius
            self.text = str(selectedObject.radius)

        case 3: #velocity
            self.text = str(selectedObject.velocity * 1000) #convert to km/h

        case 4: #speed
            self.text = str(round(selectedObject.velocity.magnitude() * 1000, 1)) #convert to km/h and round

        case 5: #surface gravity
            self.text = str(round(selectedObject.calcSurfaceGrav(), 3))

        case 6: #surface temperature (STAR OBJECTS ONLY)
            self.text = str(selectedObject.temperature)

        case 7: #time period
            timePeriod = round((2*math.pi*(selectedObject.gamePos.distance_to(masterObject.gamePos)**(3/2)))/(math.sqrt(6.67e-11*masterObject.correctedMass)), 2) #calculate time period, in seconds

            #determine units for time period
            if timePeriod < 300: #if time period is less than 5 minutes:
                self.text = str(timePeriod) + " secs" #use seconds as units
            elif timePeriod < 10800: #if time period is less than 3 hours:
                self.text = str(timePeriod/60) + " mins" #convert and use minutes as units
            elif timePeriod < 259200: #if time period is less than 3 days:
                self.text = str(timePeriod/3600) + " hrs" #convert and use hours as units
            else:
                self.text = str(timePeriod/86400) + " days" #convert and use days as units

        case 8: #orbital radius
            self.text = str(round(selectedObject.gamePos.distance_to(masterObject.gamePos), 1)) #get orbital radius as distance to master sprite

        case 9: #orbital velocity
            self.text = str(round(math.sqrt(force*selectedObject.gamePos.distance_to(masterObject.gamePos)))) #get orbital velocity as root(force * distance to sprite)

        case _: #no designation
            print("ERROR: Textbox has no designation.")

    
```

However, the issue I ran into with this was that the master sprite was not declared inside the method. I could put the calculation step in the method before each of the values is taken, but this is very inefficient. Instead, I opted to pass through the master object as a parameter. Another change I made was that if the textbox is active, the method will return before updating any values to prevent it updating while the user is entering data. Inside the main running loop, where the logic for the side info panel pages are processed, I called the update() method for each respective textbox group. I do not use the master sprite in any of the textboxes' calculations in the object group, so to save unnecessarily calculating it, I just pass through 0.

When the object is clicked, it enters its ‘active’ state where it listens for any text input; it should also change its colour according to the clickedColour and unclickedColour. To do this, I call a new `clicked()` method when the mouse button is pressed that checks if the mouse cursor is over the parent object, and changes its state accordingly. I realised that the colour variables are only used within this method and do not change, and so they can be removed in favour of directly setting the colours.

```
def clicked(self): #called on mouse click

    if self.rect.collidepoint(pygame.mouse.get_pos) and not self.active: #if this object is clicked:

        self.active = True
        self.image.fill((172,172,172)) #fill with grey

    elif self.active and not self.rect.collidepoint(pygame.mouse.get_pos): #if object is active, but click was elsewhere on screen

        self.active = False
        self.image.fill((255,255,255)) #fill with white
```

When a key has been pressed by the user, the Pygame events system is used to call the `textInput()` method of each active textbox object, which takes in the event itself as a parameter. If the pressed key is backspace, it removes the last character of any active textbox’s text. If it is a digit (the unicode of the event key pressed is compared to find this out), it is added to the text, and if it is the enter key then the changes are made to the selected object. To check which variable of the selected object to change, I will again use another match...case statement. I may in the future look at using the designation of the textbox as the index value for an array containing the names of the corresponding variables for easier maintenance.

```
def textInput(self, input):

    global selectedObject

    print(input) #DEBUG

    if input.key == pygame.K_BACKSPACE and len(self.text) > 0: #if backspace pressed, and there is text in the textbox:

        self.text = self.text[:-1] #set text as itself up to last character

    elif input.key == pygame.K_RETURN: #if enter key pressed:

        #TODO make changes to object based on text
        pass

    elif 40 > event.unicode > 29: #if input is a number:

        self.text += event.unicode #add to text
```

When a variable is changed that is calculated using other factors, such as the surface gravity, other variables must change in order to accommodate for this. For instance, if surface gravity is to change, either radius or mass or both of the selected planet must increase or decrease. Time period, orbital radius and orbital velocity are the other variables for which this is also the case. To do this, I will have to create a freeze button for mass and radius, among other things. As I had already implemented the framework around this, it took less than 5 minutes to create two more button

objects for this purpose. Below is the code for changing each variable:

```

elif input.key == pygame.K_RETURN: #if enter key pressed:

    #make changes to object based on text
    match self.designation:

        case 1: #mass
            selectedObject.correctedMass = int(self.text)
            selectedObject.mass = int(self.text) / (10**24) #convert back to base mass

        case 2: #radius
            selectedObject.correctedRadius = int(self.text)
            selectedObject.radius = int(self.text) / (10**6) #convert back to base radius

        case 3: #velocity
            selectedObject.velocity = pygame.Vector2(self.text) / 1000 #convert back from km/h

        case 4: #speed
            #similar to zoom: by increasing speed, I must increase both components of velocity equally by the same factor
            #factor = old speed / new speed
            selectedObject.velocity = selectedObject.velocity * (selectedObject.velocity.magnitude() / int(self.text)) / 1000 #convert back from km/h

case 5: #surface gravity

    #G = GM/(R**2)

    if selectedObject.massFreezeFlag and selectedObject.radiusFreezeFlag:
        return #if both are frozen, the gravity cannot change while keeping consistent physics

    elif selectedObject.massFreezeFlag: #if mass frozen, but radius is not:
        #G inversely proportional to R**2
        #if G*4, R*1/sqrt(4)
        #G increases from old to new => new R = old R * 1/(new G / old G)
        selectedObject.radius = selectedObject.radius * 1/(int(self.text) / selectedObject.calcSurfaceGrav()) #set radius
        selectedObject.correctedRadius = selectedObject.radius * (10**6) #use to calculate corrected radius

    elif selectedObject.radiusFreezeFlag: #if radius frozen, but mass is not:
        #G proportional to M
        #if G*4, M*4
        #new M = old M * (new G / old G)
        selectedObject.mass = selectedObject.mass * (int(self.text) / selectedObject.calcSurfaceGrav()) #set mass
        selectedObject.correctedMass = selectedObject.mass * (10**24) #use to calculate corrected mass

    else: #if nothing frozen
        #split effects between both factors
        #if G*4, M*2, R*1/sqrt(2)
        #basically halve the stuff from before
        selectedObject.radius = (selectedObject.radius * 1/(int(self.text) / selectedObject.calcSurfaceGrav()))/2 #set radius
        selectedObject.correctedRadius = selectedObject.radius * (10**6) #use to calculate corrected radius
        selectedObject.mass = (selectedObject.mass * (int(self.text) / selectedObject.calcSurfaceGrav()))/2 #set mass
        selectedObject.correctedMass = selectedObject.mass * (10**24) #use to calculate corrected mass

```

I have made it very difficult on myself by making the units for time period change depending on its value. At present, in my scenarios I never get a time period that is in any other unit than seconds, so I will assume for now that that is the unit that the user will enter (as it is labelled as such); I commented out the unit conversions in case I had the time to re-implement them later. To calculate the new orbital radius from the time period, I rearranged the equation $T = \frac{2\pi d^{3/2}}{\sqrt{GM}}$ to solve for d (representing orbital radius).

```

case 7: #time period

    if not selectedObject.orbitalRadiusFreezeFlag: #if orbital radius frozen, cannot change while keeping consistent physics

        #time period = 2 * pi * ((selectedObject distance to masterObject)**(3/2)) / (sqrt(6.67e-11 * masterObject mass))
        #=> new d = ((t*sqrt(6.67e-11*mass))/(2*pi))**(2/3)
        #factor = new d / old d

        factor = ((self.text*math.sqrt(6.67e-11*masterObject.correctedMass))/(2*math.pi))**(2/3) / selectedObject.gamePos.distance_to(masterObject.gamePos)

        #components: new x or y = factor*(distance to origin - distance to master) + distance from master to origin
        selectedObject.gamePos[0] = factor*(selectedObject.gamePos[0] - masterObject.gamePos[0]) + masterObject.gamePos[0]
        selectedObject.gamePos[1] = factor*(selectedObject.gamePos[1] - masterObject.gamePos[1]) + masterObject.gamePos[1]

```

For the change in orbital radius, I copied the calculation as time period and orbital velocity would just be recalculated in the textbox update() method the next tick.

```
case 8: #orbital radius

#orbital radius = selectedObject distance to masterObject

#factor = new d / old d
factor = self.text / selectedObject.gamePos.distance_to(masterObject.gamePos)

#components: new x or y = factor*(distance to origin - distance to master) + distance from master to origin
selectedObject.gamePos[0] = factor*(selectedObject.gamePos[0] - masterObject.gamePos[0]) + masterObject.gamePos[0]
selectedObject.gamePos[1] = factor*(selectedObject.gamePos[1] - masterObject.gamePos[1]) + masterObject.gamePos[1]
```

To calculate and pass through the master object for use in these cases, I set the master object to be an optional parameter for the `textInput()` method, and calculated and passed it through in the events loop only if it was for the relevant textboxes.

```
for i in textBoxOrbitGroup:

    if i.active:

        if i.designation == 7 | 8: #if time period or orbital radius box:
            #determine master object
            greatestForce = 0 #set initial greatest force as 0
            masterObject = selectedObject #set master object as itself for time being

            for j in range(len(physicsSprites)): #for each other sprite in group:

                if selectedObject is not j: #skip if object is the same as the object calculating force for

                    force = physEng.calcGravBetweenObjects(physicsSprites[j], physicsSprites[j].gamePos.distance_to(selectedObject.gamePos) * 1e6)

                    if force > greatestForce: #if this is new greatest force:

                        greatestForce = force #set new greatest force
                        masterObject = physicsSprites[j] #set master object as object force was calculated from

                i.textInput(event, masterObject) #pass master object as parameter

        else:
            i.textInput(event)
```

For the orbital velocity, I rearranged another equation to get $d = \frac{GM}{v^2}$ (with d again representing orbital radius).

```
case 9: #orbital velocity

if not selectedObject.orbitalRadiusFreezeFlag: #if orbital radius frozen, cannot change while keeping consistent physics

    #orbital velocity = sqrt(force on selected object by master object * selectedObject distance to masterObject)
    #=> new d = (6.67e-11*master object mass) / velocity**2
    #factor = new d / old d

    factor = ((6.67e-11*masterObject.correctedMass) / self.text**2) / selectedObject.gamePos.distance_to(masterObject.gamePos)

    #components: new x or y = factor*(distance to origin - distance to master) + distance from master to origin
    selectedObject.gamePos[0] = factor*(selectedObject.gamePos[0] - masterObject.gamePos[0]) + masterObject.gamePos[0]
    selectedObject.gamePos[1] = factor*(selectedObject.gamePos[1] - masterObject.gamePos[1]) + masterObject.gamePos[1]
```

Mass: 10 x10²⁴kg
Radius: 20 x10⁶m
Velocity: [0, 0] km/h
Speed: 0.0 km/h
Surface gravity: 1.6675 m/s²
Surface temperature: 2000K

To position the boxes on the screen, they will be drawn by looping over each item in the respective textbox group individually, instead of just using the `group.draw()` method. This is so that I do not have to store the coordinates of the object on the screen. I did not have to do this previously as the button objects were all positioned in the centre of the screen and drawn as such, while the slider objects took their positioning from the values that they represented.

For the surface temperature textbox, I had to place it separately to the other textboxes as it would only be shown if the selected object was a star. Creating another entire group for it was unnecessary this time as I was directly blitting the surface to the screen.

I then began to run into problems with the rects; they were being created at the top left of the screen, and as such all the collisions would be there. This was due to the `image.get_rect()` method only getting the size of the surface, and not its position on the screen. I then realized that using surfaces to represent the textboxes was inefficient; I could instead just draw a rectangle using the rect coordinates of each object without them having an image surface at all. To accommodate for the colour of the rectangle being changed, I changed the `self.image.fill` lines with the setting of a new `self.colour` variable. These variables are read by the main program when drawing the rectangle for the corresponding textbox to the screen.

```
#place textboxes on screen
for i in textBoxOrbitGroup.sprites():
    pygame.draw.rect(screen, i.colour, i.rect)
```

```
def rescale(self):
    if self.designation == 6: #if star temperature textbox:
        self.rect.update(0, screen.get_height()*0.005, screen.get_width()*0.13, screen.get_height()*0.03) #update rect for star temperature
    elif self.designation < 7: #if object screen textbox:
        self.rect.update(0, screen.get_height() * (0.425 + 0.03 * self.designation), screen.get_width()*0.13, screen.get_height()*0.03) #position rect according to designation
    else: #if orbit screen textbox:
        self.rect.update(0, screen.get_height() * (0.245 + 0.03 * self.designation), screen.get_width()*0.13, screen.get_height()*0.03) #position rect according to designation
```

To get the values on the text to update as they were being written, I simply replaced the variables that the text in the info panel referenced with the text value of the relevant textbox.

```
if sidePanelPage == 0: #object info page

    textBuffer = [0,0,0,0,0,0] #generate array for text surfaces

    textboxes = textBoxObjectGroup.sprites() #cache sprites for efficient call

    sidePanelObjectButtons.draw(screen) #draw buttons on side panel

    textBuffer[2] = font.render(f"Mass: {textboxes[0].text} x10^24kg", True, (0,0,0)) #render font for mass
    textBuffer[3] = font.render(f"Radius: {textboxes[1].text} x10^6m", True, (0,0,0)) #render font for radius
    textBuffer[4] = font.render(f"Velocity: {textboxes[2].text} km/h", True, (0,0,0)) #render font for velocity, converted to kilometres per hour
    textBuffer[5] = font.render(f"Speed: {textboxes[3].text} km/h", True, (0,0,0)) #render font for speed, converted to kilometres per hour
    textBuffer[6] = font.render(f"Surface gravity: {textboxes[4].text} m/s^2", True, (0,0,0)) #render font for surface gravity

    if selectedObject.__class__.__name__ == "Star": #if selected object is a star

        textBuffer.append(font.render(f"Surface temperature: {surfaceTempTextbox.text}K", True, (0,0,0))) #render font for surface temperature
        surfaceTempTextbox.update(0) #update star surface temperature textbox
```

Running to test, the textboxes worked correctly when numbers were input but crashed and returned an error when anything other than a digit was entered.

```
Traceback (most recent call last):
  File "e:\NEA (USB)\proj\main.py", line 880, in <module>
    surfaceTempTextbox.textInput(event)
  File "e:\NEA (USB)\proj\main.py", line 428, in textInput
    elif 10 > int(event.unicode) > -1: #if input is a single digit number:
                                         ~~~~~
ValueError: invalid literal for int() with base 10: '-'
PS E:\NEA (USB)\proj>
```

This was due to the `event.unicode` being the key that was pressed, instead of the unicode corresponding to that key as I previously assumed. This can be fixed by creating an array (`validInputs`) with each valid character, and checking the `event.unicode` against that array; if it is in the list, it is added to the text. This still allows for the input to be filtered to only numbers and a decimal point, but I also included square brackets and commas for the velocity variable specifically.

```
validInputs = ["0","1","2","3","4","5","6","7","8","9","[","]",",",".",","]
```

```
elif event.unicode in validInputs: #if input is a single digit number or square brackets:  
    self.text = str(self.text) + event.unicode #add to text
```

Now, the program would not crash when different inputs happened, but it would not accept any input for the velocity.

```
pygame 2.5.2 (SDL 2.28.3, Python 3.11.0)  
Hello from the pygame community. https://www.pygame.org/contribute.html  
Traceback (most recent call last):  
  File "e:\NEA (USB)\proj\main.py", line 876, in <module>  
    itextInput(event) #update active text boxes with event data  
    ~~~~~~  
  File "e:\NEA (USB)\proj\main.py", line 370, in textInput  
    selectedObject.velocity = pygame.Vector2(self.text) / 1000 #convert back from km/h  
    ~~~~~~  
ValueError: Vector2 must be set with 2 real numbers, a sequence of 2 real numbers, or another Vector2 instance  
PS E:\NEA (USB)\proj>
```

To fix this issue, I will need to use the `split()` method to split the text input into two values, separated by the comma. I will then use `strip()` to clean the result of the square brackets and the spaces.

```
case 3: #velocity  
    #split into two values (x,y)  
    values = self.text.split(",")  
    values[0].strip(" [,]") #strip values of spaces, commas, square brackets  
    values[1].strip(" [,]")  
    selectedObject.velocity = pygame.Vector2(values[0], values[1]) / 1000 #convert back from km/h
```

However, this produced the same result. I printed the values array to see what the splitted values were. This showed that the square brackets were not being stripped from the values, which was strange as they were definitely being passed into the `strip()` method. I then realised that I was not setting the values as their stripped versions, which was easily fixed. I then explicitly set the values as floats before the `Vector2` was declared. This worked as intended.

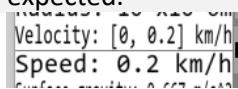
```
case 3: #velocity  
    #split into two values (x,y)  
    values = self.text.split(",")  
    values[0] = float(values[0].strip(" [,]")) #strip values of spaces, commas, square brackets  
    values[1] = float(values[1].strip(" [,]"))  
    selectedObject.velocity = pygame.Vector2(values[0], values[1]) / 1000 #convert back from km/h
```

I also changed a lot of the other `int()` conversions in this method to `float()` for more accuracy, and to avoid errors if decimal values were entered.

The textboxes are working correctly, and I now consider this stage complete.

Testing + Bug fixing

Test #	Test description	Expected outcome	Actual outcome	Pass/fail
1a	Click each object in turn.	selectedObject variable changes to become each object in turn.	As expected.	Pass
2a	Single object in simulation: Open info panel.	Correct data displayed.	As expected.	Pass

2b	Two objects in simulation: Open info panel for each object in turn.	Correct data displayed for each respective object.	As expected.	Pass
3a	Mass manipulation: change value for mass of an object to 5 (normal).	Object's mass changes in panel, and change is reflected in physics.	As expected.	Pass
3b	Change value to -5 (erroneous).	Object's mass does not change.	As the “-” symbol is not in the validInputs array, it is impossible to enter any negative value.	Pass
3c	Change value to 0 (boundary).	Object's mass is set to 0, and change is reflected in physics. (I am doing this test to check for any divide by 0 errors.)	Object exerts no gravitational force on any other object once it has been set to 0 mass.	Pass
3d	Change value to 1000 (extreme).	Object's mass changes in panel, and change is reflected in physics.	Object exerts an extreme force, as expected.	Pass
4a	Speed manipulation: change value for speed of an object to 5 (normal).	Object's speed and velocity change in panel, and change is reflected in physics.		Fail, fixed

When I change the speed of the object, I split it into the two components of velocity and add it by a factor to each one – however, if one or both of either the original or new values are 0, then the factor will either multiply or divide by 0, which will not produce the required result. Additionally, the direction of velocity that the planet is already travelling is not taken into account when the speed is changed.

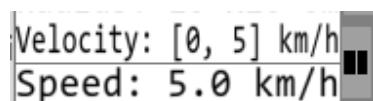
To fix this, instead of using a factor of the increase of speed, I will use the ratio of both initial velocities to add to them in a proportional way. After deriving some equations, I came up with these for the final velocities:

$$v_x = \frac{\Delta Speed + \sqrt{u_x^2 + u_y^2}}{\sqrt{(\frac{u_y}{u_x})^2 + 1}}, v_y = \frac{u_y v_x}{u_x} \quad \text{where } u \text{ is the initial velocity and } v \text{ is the final velocity. There is}$$

an issue with dividing by 0 when the horizontal component of the initial velocity is 0, so I will check for this before and if so just add to the vertical component.

```
case 4: #speed
    if selectedObject.velocity[0] == 0: #check for 0 in horizontal component to prevent div by 0 error
        selectedObject.velocity[1] = float(self.text) / 1000 #speed change affects only vertical velocity, convert to km/h
    else:
        selectedObject.velocity[0] = (abs(selectedObject.velocity.magnitude() - float(self.text)) + selectedObject.velocity.magnitude()) / (math.sqrt((selectedObject.velocity[0]**2 + 1) * 1000))
        selectedObject.velocity[1] = selectedObject.velocity[0] * (selectedObject.velocity[1]/selectedObject.velocity[0]) / 1000 #calculate, convert and set vertical velocity
```

Running to test:



When the initial velocity is [0,5], the first part of the if statement is run and the velocity changes correctly.

```
Velocity: [5, 5] km/h
Speed: 7.1 km/h
Velocity: [353.553, 0.3] km/h
Speed: 353.6 km/h
```

I then changed the velocity to [5,5] which also updated the speed correctly.

However, when I changed the speed to 350, the velocity was incorrectly changed, which then in turn updated the speed incorrectly.

Instead of calculating the new velocity by using the ratio of each component, I decided to do it by using the constant angle between them instead. This gave me these equations:

$$v_x = (m_u + |m_u - I|) \cos(\tan^{-1}(\frac{u_y}{u_x}))$$

$$v_y = (m_u + |m_u - I|) \sin(\tan^{-1}(\frac{u_y}{u_x}))$$

Where I is the new speed input and m_u is the magnitude of the initial velocity. The brackets on the left side of each equation is to find the magnitude of the new velocity, while the \tan^{-1} expression on the left is to find the angle between the two components of velocity. I then use sin or cos to solve for the two components of the final velocity.

The only difference between these two equations is the trig function used with the angle, so I will calculate the new magnitude and the angle beforehand. I also convert the final values back to km/h by dividing by 1000.

```
else:
    magnitude = selectedObject.velocity.magnitude() + abs(selectedObject.velocity.magnitude() - float(self.text))
    angle = math.atan(selectedObject.velocity[1]/selectedObject.velocity[0])

    selectedObject.velocity[0] = (magnitude * math.cos(angle)) / 1000
    selectedObject.velocity[1] = (magnitude * math.sin(angle)) / 1000
```

```
Velocity: [247.487, 247.487] km/h
Speed: 350.0 km/h
```

```
Velocity: [47.1495, 70.7243] km/h
Speed: 85.0 km/h
```

Now, the velocity correctly updates when a value is entered.

<i>Test #</i>	<i>Test description</i>	<i>Expected outcome</i>	<i>Actual outcome</i>	<i>Pass/fail</i>
4b	Change value to -5 (erroneous).	Object's speed and velocity do not change.	As the “-” symbol is not in the validInputs array, it is impossible to enter any negative value.	Pass
4b (ii)	Made after change to allow “-” symbol. Same test as above.	Object's speed and velocity do not change.	As expected.	Pass
4c	Change value to 0 (boundary).	Object's speed and velocity become 0 in panel, and object stops moving in simulation.	As expected.	Pass
4d	Change value to 1000 (extreme).	Object's speed and velocity change in panel, and change is reflected in physics.	As expected.	Pass

4e	Change value to 10 with velocity at [-100, 50]			
5a	Velocity manipulation: change value for velocity of an object to [5,5] (normal).	Object's speed and velocity change in panel, and change is reflected in physics.	As expected.	Pass
5b	Change value to [-5,0] (normal).	Object's speed and velocity change in panel, and change is reflected in physics.	As the “-” symbol is not in the validInputs array, it is impossible to enter any negative value.	Fail, fixed

To allow for negative values, I have to add the “-” sign to the validInputs array. However, this will also allow this symbol to be entered into the other textboxes. Therefore, I will need to add an abs() method to the text input in all non-negative variable calucations to ensure that only positive numbers are returned for them.

```
if self.designation != 3: #if not velocity:  

    self.text = abs(float(self.text)) #ensure positive value for non-negative variables
```

I tested this by entering a negative value for speed, which returned the change as if it were a positive value. However, when I entered the input “1-”, the program crashed. To solve this issue, I will instead just strip the input for these variables of any of the validInput characters that would cause this.

```
if self.designation != 3: #if not velocity:  

    self.text = float(self.text.strip(" ,[]-")) #ensure positive and valid value for non-negative variables
```

The only way that this will now crash is if two decimal points are input. To account for this, I will have to examine each character in the input string, and if there is a decimal point in a string where another decimal point has already been found, it will remove that point. This solution uses an array to store the characters that have been validated, that are joined together at the end; this was the simplest solution I could come up with.

```
if self.designation != 3: #if not velocity:  

    self.text = self.text.strip(" ,[]-") #ensure positive and valid value for non-negative variables  

    decimal = False  

    tempArray = []  

    for char in self.text: #for each character in input:  

        if char == ".": #if decimal point:  

            if not decimal: #if there has already been a decimal point, do not add this character to the array. otherwise, add it and set flag:  

                decimal = True  

                tempArray.append(char)  

            else:  

                tempArray.append(char)  

    self.text = float("".join(tempArray)) #join chars in array, and convert to float
```

The program now does not crash with any input to the textboxes.

Testing the velocity with a negative value, it works as expected.

Test #	Test description	Expected outcome	Actual outcome	Pass/fail
5c	Change value to [0,0] (normal).	Object's speed and velocity become 0 in panel, and object stops moving in simulation.	As expected.	Pass
5d	Change value to [1000, 1000] (extreme).	Object's speed and velocity change in panel, and change is reflected in physics.	As expected.	Pass
6a	Time period manipulation: change value for time period of an object to 5 (normal).	Object's time period changes in panel, and change is reflected in physics.	Time period: 5.0 secs Orbital radius: $1.61647.4 \times 10^6$ m Orbital velocity: 0.2 km/h	Fail? Fixed

It looks like the scale for the units is off; the orbital radius does actually seem to be relatively realistic, so I will change the units for time period to be hours instead, which should be more consistent.

Test #	Test description	Expected outcome	Actual outcome	Pass/fail
6b	Change value to -5 (erroneous).	Object's time period does not change.	As expected.	Pass
6c	Change value to 0 (boundary).	Object's time period becomes 0 in panel, and change is reflected in physics.	Time period: 0.0 hrs Orbital radius: 0.0 x10^6m Orbital velocity: 0.0 km/h Object moves to centre of second object (star)	Pass
6c	Change value to 1000 (extreme).	Object's time period changes in panel, and change is reflected in physics.	Program crash	Fail, fixed

This test was done immediately after the previous, and so the calculation for the factor to increase the time period by included a division by 0 in the old factor for orbital radius. To fix this, I will include a check for this specific case in each of the three relevant calculations. If the orbital radius is 0 then the value used will be a very small number (0.000001) to allow for the calculation to proceed. I am aware that this will produce a result that is very slightly off, but as the replacement number tends to but does not reach 0 it gets closer to the actual result.

```

case 7: #time period

if not selectedObject.orbitalRadiusFreezeFlag: #if orbital radius frozen, cannot change while keeping consistent physics

    #time period = 2 * pi * ((selectedObject.distance_to(masterObject)**(3/2)) / (sqrt(6.67e-11 * masterObject.mass)
    #=> new d = ((t**sqrt(6.67e-11*mass))/(2*pi))**(2/3)
    #factor = new d / old d
    if selectedObject.gamePos.distance_to(masterObject.gamePos) == 0: #prevent div by 0 error if objects are at same position
        #print("flag") debug
        #print(selectedObject.gamePos[0])
        selectedObject.gamePos[0] += 1 #move object very slightly in order to prevent div by 0
        #print(selectedObject.gamePos[0])
        #print(selectedObject.gamePos.distance_to(masterObject.gamePos))

    factor = ((self.text*math.sqrt(6.67e-11*masterObject.correctedMass))/(2*math.pi))**(2/3) / selectedObject.gamePos.distance_to(masterObject.gamePos)

    #components: new x or y = factor*(distance to origin - distance to master) + distance from master to origin
    selectedObject.gamePos[0] = factor*(selectedObject.gamePos[0] - masterObject.gamePos[0]) + masterObject.gamePos[0]
    selectedObject.gamePos[1] = factor*(selectedObject.gamePos[1] - masterObject.gamePos[1]) + masterObject.gamePos[1]

```

However, this still produced the same divide by 0 error. Printing the gamePos of the selected object just before and after it is moved by 1, the change is made, but when the distance to the masterObject gamePos is printed it says there is no difference between the two vectors. Printing the masterObject gamePos before and after the change as well, it shows that it has also been changed by the same amount. This confused me as there is no reason for this to be happening unless the selected object was setting itself as the master object.

After looking at the code that determined the master object, I found that I had made a logical error.

```
if selectedObject is not j: #skip if object is the same as the object calculating force for
```

In the above line, j is not an object; it is the index of that object in the list. This is fixed in the changed line below:

```
if selectedObject is not physicsSprites[j]: #skip if object is the same as the object calculating force for
```

I ran the program again after this change to get the same result. I then looked at the initial conditions of this loop:

```
greatestForce = 0 #set initial greatest force as 0
masterObject = selectedObject #set master object as itself for time being
```

The issue with these conditions is that the other star object in the scenario will return a force of 0 as it is at the same position as the selected object. To set the master object, the force applied to the selected object has to be greater than the greatestForce experienced as a result of any other object.

Time period: 10.0 hrs
Orbital radius: 256599.2 x10⁶m
Orbital velocity: 0.16 km/h

To fix this and allow the star to be selected, I set the initial greatestForce to -1. I also set the initial masterObject to 0. Running to test, this produced the desired result.

However, this made me wonder what would happen if the selected object was the only object present in the simulation. Running with only one object this time to test, the program crashed when any of the variables to do with a master object were changed. To fix this, I will run a simple check that will not call the `textInput()` method of the relevant textboxes when the master object is not set by the loop.

I will not need to run this check or make the change to the initial master object set before the `textbox update()` call in the main running loop, as these values will just be 0 if there is a single object in the simulation anyway.

Test #	Test description	Expected outcome	Actual outcome	Pass/fail
7a	Dual planet system: lock velocity of one planet.	Velocity of locked object does not change, whereas velocity of unlocked object does.	As expected.	Pass
7b	Dual planet system: lock speed of one planet.	Speed of locked object does not change, whereas speed of unlocked object does.	As expected. Unified button for speed/velocity, so exact same process as above test.	Pass

7c	Surface gravity manipulation: change value to 5 (normal).	Mass and radius change accordingly, with equal weighting on both.	Mass: 0.016675 x10^24kg Radius: 0.667 x10^6m Velocity: [0, 1000] km/h Speed: 1000.0 km/h Surface gravity: 2.5 m/s^2	Fail
----	--	---	---	------

It is time to do some more maths. My current method for changing the surface gravity is by using factors, i.e. finding the multiple of the new surface gravity to get the old surface gravity and multiplying the other values by this. However, if the mass of an object is set to 0, then there is a division by 0 error.

I made a copy of the main.py file in order to try out a new method of calculation. These are the equations I came up with, derived from the gravitational force equation $g = \frac{GM}{r^2}$:

Radius variable locked (change in mass)	$M_{new} = \left \frac{g_{new} r^2}{G} \right $
Mass variable locked (change in radius)	$r_{new} = \sqrt{\frac{GM}{g_{new}}}$
Neither variable locked (change in both) This should produce an equal weighting for both mass and radius.	$M_{new} = \left \frac{g_{new} r_{old}^2}{2G} \right $ $r_{new} = \sqrt{\frac{2GM_{old}}{g_{new}}}$

```
case 5: #surface gravity

#G = GM/(R**2)

if selectedObject.massFreezeFlag and selectedObject.radiusFreezeFlag: #if both are frozen, the gravity cannot change while keeping consistent physics
    self.text = str(round(selectedObject.calcSurfaceGrav(), 3))#reset gravitational force to what it was before
    return

elif selectedObject.massFreezeFlag: #if mass frozen, but radius is not:
    selectedObject.correctedRadius = math.sqrt((6.67e-11*selectedObject.correctedMass)/self.text) #calculate radius
    selectedObject.radius = selectedObject.correctedRadius / (10**6) #convert to corrected radius

elif selectedObject.radiusFreezeFlag: #if radius frozen, but mass is not:
    selectedObject.correctedMass = ((self.text * selectedObject.correctedRadius**2)/6.67e-11) #calculate mass
    selectedObject.mass = selectedObject.correctedMass / (10**24) #use to calculate corrected mass

else: #if nothing frozen

    oldRadius = selectedObject.correctedRadius #get old radius for mass calc

    selectedObject.correctedRadius = math.sqrt((2 * selectedObject.correctedMass * 6.67e-11)/self.text) #calculate new radius
    selectedObject.radius = selectedObject.correctedRadius / (10**6) #calculate corrected radius

    selectedObject.correctedMass = (oldRadius**2 * self.text)/(2 * 6.67e-11) #calculate new mass
    selectedObject.mass = selectedObject.correctedMass / (10**24) #calculate corrected mass
```

Running to test, the mass freeze and radius freeze calculations worked perfectly on their own, but with no values frozen the result was again wrong. I decided to just quickly fix this by making it so that the calculation was done as if the radius of the object were frozen in this case, but I will go back after the implementation of the other desired features in another attempt to fix it.

Test #	Test description	Expected outcome	Actual outcome	Pass/fail
7d	Lock mass and change value to 5.	Radius changes accordingly, with mass remaining the same.	As expected.	Pass

7e	Lock radius and change value to 5.	Mass changes accordingly, with radius remaining the same.	As expected.	Pass
7f	Lock both mass and radius and change value to 5.	All values do not change.	As expected.	Pass
8a	Increase equipotential line resolution.	Resolution increases by one step per click.	As expected.	Pass
8b	Decrease equipotential line resolution.	Resolution decreases by one step per click.	As expected.	Pass

At the end of testing, I re-ran all tests from this section in order to ensure that none of my bugfixing had made any other undesired changes.

Stakeholder Feedback

Almost all of the stakeholders have said that a button to hide/show the entire UI would be nice, as it would provide a much greater field of vision for observing the simulation if they are done with changing any settings to set it up.

Kai and Oli both wanted it so that when you clicked on a planet object that is selected, it becomes the focused object. This should be very simple to implement.

Stage 5 – Build menu and spacecraft

Success criteria

REQUIREMENT	DESCRIPTION	SUCCESS CRITERIA	CHECK
2	Controllable spacecraft	Object that the player can directly control via a separate menu.	✓
5B	Create + destroy planets	Users can place at least two celestial bodies at variable locations.	✓

Testing plan

Test #	Test description	Expected outcome
1a	Spawn spacecraft while camera is at origin.	Spacecraft is created in correct location at the centre of the screen.
1b	Spawn spacecraft while camera is at point other than origin.	As above.
1c	Spawn spacecraft while camera is zoomed out, not at origin.	As above.
1d	Deactivate spacecraft button.	Spacecraft object removed from physObjects group.
1e	Respawn spacecraft at new position after despawn.	Spacecraft object added to physObjects group. Spacecraft appears in correct location.
2a	Spacecraft movement: accelerate spacecraft in default (facing X-positive) direction.	Spacecraft's velocity X component increases, while Y component remains the same.
2b	Turn and accelerate facing top right direction.	Velocity X and Y components increase in correct proportion.

2c	Turn and accelerate facing top left direction.	As above.
2d	Turn and accelerate facing bottom right direction.	As above.
2e	Turn and accelerate facing bottom left direction.	As above.
3a	Spacecraft info panel: change radius value.	Change not accepted, and textbox reset as "N/A"
3b	Change surface gravity value.	As above.
4a	Build menu: select move object button, then close build menu	Move button deselected.
4b	As above with delete object button.	Delete button deselected.
4c	Select object with move tool.	Object is set as pickedObject.
4d	Delete object with deletion tool.	All references to object are removed.
4e	Place planet using planet tool.	New, unique planet object created and added to physObjects group.
4f	Place star using star tool.	New, unique star object created and added to physObjects group.

Development

First, as per stakeholder feedback, I made it so that if a selected object was clicked again it became the focused object, by using a check to see if the object was selected in the `checkIfSelected()` method; if so, the physics engine `updateCamera()` method would be called with the object itself as the argument to set itself as focused.

The next requested feature is the button to hide the UI. This will be designation 12, and I have decided to position it underneath the camera focus button for ease of reach.



These are the designs for the hide/show button.

To draw this button separately, I created a new Pygame sprite group, `hideUIButtonGroup`. This will be the only button group that is always drawn on the screen, regardless of settings.

In order to make the other buttons disappear, I will run a check against the `isClicked` property of the button in the main running loop where the buttons are drawn. Running to test, the button works as intended.



```
class Spacecraft(pygame.sprite.Sprite):
    def __init__(self, mass, pos, velocity):
        pygame.sprite.Sprite.__init__(self) #initialise Pygame sprite
        self.mass = mass
        self.image = pygame.image.load("spacecraft.png").convert_alpha()
        self.rect = self.image.get_rect()

        #set initial variable freeze flags
        self.velocityFreezeFlag = False
        self.timePeriodFreezeFlag = False
        self.orbitalRadiusFreezeFlag = False
        self.orbitalVelocityFreezeFlag = False

        self.gamePos = pos
        self.screenPos = pos #set position on screen initially
        self.velocity = velocity

    def turnLeft(self):
        pass

    def turnRight(self):
        pass

    def addVelocity(self):
        pass
```

The spacecraft is an integral feature of this program, as it provides an easier to control object that moves at a human scale, instead of a celestial one. I made a simple design in paint.net and began to work on the spacecraft class.

I first coded the skeleton of the object class before creating a sprite group for drawing the spacecraft. I have included flags here as well so that the spacecraft is able to be selected for the info panel, and values can be manipulated. However, if I am to make the spacecraft selectable, it will need its own `checkIfSelected()` method. I reexamined this method in the planet class, and resolved to remove it in favour of doing the checks by looping through each physics object directly in

the events loop. I moved the code and commented out the method in the planet class to test.

```
if buttonGroup.sprites()[2].isClicked: #if spacecraft controls active:
    spacecraftControlGroup.update()

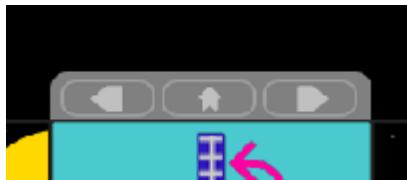
for i in physObjects: #for each physics object:
    if i.rect.collidepoint(pygame.mouse.get_pos()): #if mouse is over object:
        if i is selectedObject: #if object is already set as selected object:
            physEng.updateCamera(i) #set object as focused object
        else: #if not selected object:
            selectedObject = i #set object as the selected object
```

The lines up to the collision check were always part of the event loop, and would have followed with calls to each object's `checkIfSelected` method. Now, it does these checks directly, allowing me to use the spacecraft for selections as well. The only difference is that I will have to check if the class of the object is a spacecraft, in which

case the radius and mass freeze flags are not compared.

Another thing that I have to create is the three buttons for controlling the spacecraft, which will be designations 13, 14 and 15. I drew the images for these buttons in paint.net, as with all the others,

before creating a new group called `spacecraftControlGroup` for the purpose of only showing the buttons when the spacecraft controls toggle is active (as with the equipotential resolution buttons).



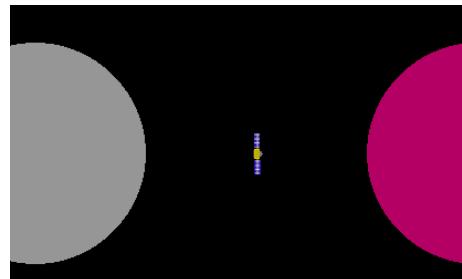
I then programmed in the draw method for this group in the main running loop, called when the spacecraft controls button is active. I also added the relevant case for each designation in the button `rescale()` method. When I first ran the program, I was confused as to why the buttons did not seem to register my mouse clicks; I then realised that I had not added their `update()` method calls to the event loop. I also added the draw call for the spacecraft itself through its own group.

To rescale and rotate the spacecraft object, I will need an `update()` method to perform rotations and rescales, as well as set and clamp its `screenPos` and `gamePos` from physics engine calculations. To make its size scale with the zoom, I use the Pygame `transform.scale_by()` method with the zoom scale as a parameter. Even after this, the spacecraft image will be too large, and so I apply a smaller multiplier to get it to an appropriate proportion compared with the planets.

```
def update(self):
    self.image = pygame.transform.rotate(pygame.transform.scale_by(self.imageUnscaled, physEng.zoomScale * 0.1), self.angle) #resize image by zoom factor, and rotate to angle
    self.gamePos += self.velocity * physEng.getTimeScale() #move sprite in terms of simulation
    self.screenPos = physEng.getScreenPos(self.gamePos, screen) #convert gamePos -> screenPos

    #check for out of bounds, and clamp to within bounds if so
    if self.screenPos[0] < -214748600:
        self.screenPos[0] = -214748600
    elif self.screenPos[0] > 214748600:
        self.screenPos[0] = 214748600
    if self.screenPos[1] < -214748600:
        self.screenPos[1] = -214748600
    elif self.screenPos[1] > 214748600:
        self.screenPos[1] = 214748600

    self.rect = self.image.get_rect(center=self.screenPos) #update rect
```



Running to test, the spacecraft would be created in the top left of the screen, and would not move from this position. I need to add this sprite to the `physObjects` group in order to process all physics logic for it. With this change made, the spacecraft is correctly positioned at the centre of the screen.

When I clicked the spacecraft to see the info panel, the program crashed. This was because it was trying to get the radius of the object to display, but the spacecraft has no radius. To quickly fix this, I wrote in a check to display "N/A" instead of a value if the selected object was the spacecraft object for both the radius and surface gravity variables. In the future, I will have to create a new sprite group for these two textboxes as well to hide them completely, but as I am running short on time this will be a good enough solution.

```
if selectedObject.__class__.__name__ != "Spacecraft": #spacecraft has no radius, so this will not be shown
    self.text = str(selectedObject.radius)
else:
    self.text = "N/A"
```

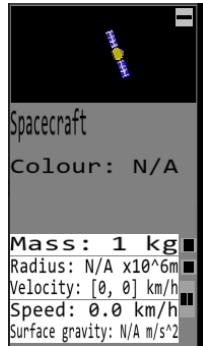
I also had to change the code that drew the object in the top of the info panel to draw the image of the spacecraft if it was the selected object. The difference with this is that the spacecraft object in the info panel will rotate to the same direction as the spacecraft in the scenario.

```
#render side info panel
if selectedObject.__class__.__name__ == "Spacecraft": #if spacecraft object:

    image = pygame.transform.rotate(pygame.transform.scale_by(spacecraftImage, screen.get_width()*0.0001), spacecraft.angle) #scale
    screen.blit(image, (screen.get_width()*0.07, screen.get_height()*0.17)) #paste to screen
    textBuffer[1] = font.render(f"Colour: N/A", True, (0,0,0)) #render font for colour

else:

    pygame.draw.circle(screen, selectedObject.colour, (screen.get_width()*0.07, screen.get_height()*0.21), screen.get_width()*0.02)
    textBuffer[1] = font.render(f"Colour: {selectedObject.colour}", True, (0,0,0)) #render font for colour
```



I also added some checks, mainly for the mass text rendering, so that the units were more consistent with the spacecraft. Running this now, the spacecraft's info correctly displayed, and the image of it was also correctly displayed.

To make the spacecraft rotate with the rotation buttons, I will pass through the spacecraft as a parameter to the button effects to directly change the angle variable; I realised that the `turnLeft()` and `turnRight()` methods were not needed because of this, and so they were removed. I can also make use of the `mouseHoldUpdate()` and `mouseHoldEnd()` methods on each of the control buttons to provide a continuous input.

```
def mouseHoldUpdate(self): #called every tick when the mouse button is down

    if ((self.designation >= 6 and self.designation <= 9) | self.designation > 19 | (self.designation >= 13 and self.designation <= 15)) and self.isClicked:
        self.effect() #apply button effect
```

```
if self.angle <= 90:

    self.velocity[0] += math.cos(self.angle) / 1000
    self.velocity[1] -= math.sin(self.angle) / 1000

elif self.angle <= 180:

    self.velocity[0] -= math.cos(180 - self.angle) / 1000
    self.velocity[1] -= math.sin(180 - self.angle) / 1000

elif self.angle <= 270:

    self.velocity[0] -= math.cos(self.angle - 180) / 1000
    self.velocity[1] += math.sin(self.angle - 180) / 1000

else:

    self.velocity[1] += math.cos(self.angle - 270) / 1000
    self.velocity[0] += math.sin(self.angle - 270) / 1000
```

Now the spacecraft turns. To make the velocity of the spacecraft increase when the add velocity button is clicked, I will use an equation very similar to the one used to manipulate the velocity of objects when the speed is changed to add speed only in the direction that the spacecraft is facing.

When I ran this to test, the movement still did not work properly; this was because the sin and cos functions worked in radians, not degrees, so I would need to convert the angle to radians before adding to the velocity. After this change, the angle of the spacecraft would also be changed, so to

avoid this I created a local variable called `radianAngle` for this method. After this change, the spacecraft controls worked correctly, and velocity was added in the correct direction. The velocity itself took a long time to increase for the spacecraft using the move forward button, so I removed the divide by 1000 in the above code.

One issue that I have found is that the program crashes when the “N/A” value in the radius of the spacecraft for its info panel is edited. To fix this, I will add the characters to the strip method and add a check so that the radius variable is never accessed if the selected object is the spacecraft.

```
~~~~~
File "d:\NEA (USB)\proj\main.py", line 386, in textInput
    self.text = float("".join(tempArray)) #join chars in array, and convert to float
~~~~~

ValueError: could not convert string to float: ''
PS D:\NEA (USB)\proj>
```

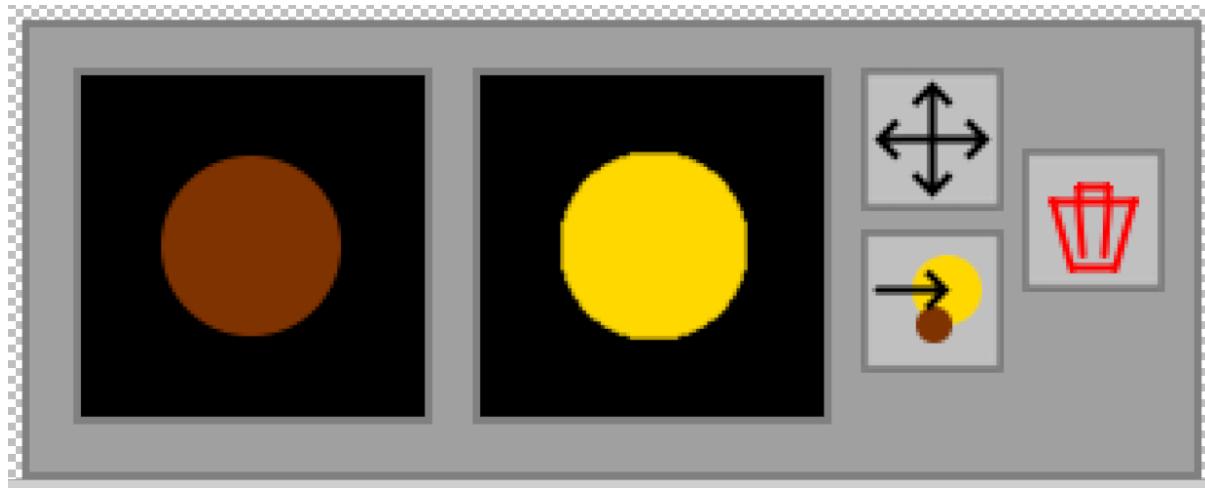
However, the float conversion would not accept an empty value; to fix this, I moved the check for an empty text input to after the strings were filtered. Now the program has no crashes due to the textboxes.

My next issue is that the spacecraft still is affected by physics even when it is not present in the simulation. To fix this, I will have to remove it from the physObjects group and re-add it when the spacecraft is placed in the simulation. Finally, I want the spacecraft to be placed in the simulation at the position where the camera is centred on the screen; to do this, I will have to reverse the screenPos calculation to position the spacecraft when the controls button is activated. I put both of these changes in the button effect() method for the spacecraft controls.

```
case 3: #spacecraft controls
    if self.isClicked: #if active:
        spacecraft.gamePos = pygame.Vector2((screen.get_width()/2)/(physEng.zoomScale + screen.get_width()/2)) - physEng.screenPos
        physObjects.add(spacecraft) #add spacecraft to physics object group for physics updates
    else:
        physObjects.remove(spacecraft)
```

Running to test, the spacecraft now correctly experiences physics at the right times, and correctly positions itself at the centre of the screen when the controls button is pressed. This is all of the functionality I would like to implement for the spacecraft, so I consider this completed.

For the build menu, I will need to create a new button group as I have done for the equipotential resolution and spacecraft control buttons, as well as the star placement, planet placement, deletion and movement buttons. I realised that I do not require the edit button that has been outlined in the design, as any changes to the object can be made in the info panel. The new group is called buildMenuGroup. I then drew the new buttons in paint.net, which will be designations 16-19. I will leave the save/load scenarios button for later, not only because I have run out of designations before the slider button at 20 but also as I have not implemented that system yet. I have left space for it on the actual menu.



I added the group's rescale() and update() methods to the events system, and created a new check in the main running loop for the draw() method. I also made clicking the build menu button pause the simulation – however the simulation can be unpause again while in the build menu if the user desires.

```
case 4: #build menu
    if not physEng.isPaused: #pause simulation if unpause
        buttonGroup.sprites()[9].isClicked = True #change pause button to pause state
        buttonGroup.sprites()[9].imageUnscaled = buttonGroup.sprites()[9].imageClicked
        buttonGroup.sprites()[9].rescale()
        physEng.isPaused = True #switch physics engine state
```

```
#####
if buttonGroup.sprites()[3].isClicked: #if build menu is active:
    for i in buildMenuGroup:
        i.mouseHoldUpdate() #update spacecraft control buttons
    buildMenuGroup.draw(screen)
```

I then repositioned the collision rects for the menu buttons. The first function of the buttons to implement will be the delete object button; this will loop over every physics object when the delete button is active and the mouse button has been pressed, and if the mouse cursor is over that object, it will be deleted. Python has no explicit delete method for objects, so I am relying on the automatic garbage data removal feature of the language to remove the object once all references to it have been cleared. I will also need another check to spare the spacecraft object from this deletion process. This all happens in the main running loop, on the mouse down event:

```
if buildMenuGroup.sprites()[3].isClicked: #if deletion button is active:
    #deletion checks
    for i in physObjects:
        if i.__class__.__name__ != "Spacecraft" and i.rect.collidepoint(pygame.mouse.get_pos()): #exclude spacecraft from deletion checks. if mouse is over object:
            if i is selectedObject:
                selectedObject = 0 #clear info panel
            if i is physEng.focusedObject:
                physEng.focusedObject = 0 #clear object focus
            physObjects.remove(i) #remove from physics objects group
```

The build menu buttons cannot be active at the same time as one another, so their initial activation should reset the states of all other active build menu buttons.

```
case 16|17|18|19: #place planet, place star, move object, delete object (build menu buttons)

    for i in buildMenuGroup: #for each other menu button:
        if i.isClicked and i is not self: #if any other button is active:
            i.isClicked = False #turn button off
            i.imageUnscaled = i.imageUnclicked #reset image
            i.rescale()
```

Another event that should reset all the build menu buttons is when the build menu is closed; to achieve this, I added in a check in the effect() method of the build menu button for when it is deactivated to perform this reset. Additionally, I had forgotten to make the check for the button being active when pausing the simulation after activating this button, so I also added this in.

```

case 4: #build menu
    if not physEng.isPaused and self.clicked: #pause simulation if unpause
        buttonGroup.sprites()[9].clicked = True #change pause button to pause state
        buttonGroup.sprites()[9].imageUnscaled = buttonGroup.sprites()[9].imageClicked
        buttonGroup.sprites()[9].rescale()
        physEng.isPaused = True #switch physics engine state

    if self.clicked == False: #build menu deactivation: reset build menu buttons

        for i in buildMenuGroup: #for each menu button:
            if i.clicked: #if any button is active:
                i.clicked = False #turn button off
                i.imageUnscaled = i.imageUnclicked #reset image
                i.rescale()

```

Running to test, the delete button worked as intended, and had no effect on the spacecraft when it was clicked. For the move button, I will have to create a variable specific to the move button object called pickedObject to keep track of the object that has been selected to be moved. In the events loop, where collisions for planets are being checked, I added a case if the move button was active which stopped any object from being selected or focused on. The clicked object would be stored in the pickedObject variable.

```

for i in physObjects: #for each physics object:

    if i.rect.collidepoint(pygame.mouse.get_pos()): #if mouse is over object:

        if buildMenuGroup.sprites()[2].clicked: #if build menu move button is active:

            if buildMenuGroup.sprites()[2].pickedObject != 0: #if no object has been picked up already:

                selectedObject = 0 #clear any selected object
                physEng.focusedObject = 0 #clear any focused object
                buildMenuGroup.sprites()[2].pickedObject = i #set object as picked object for movement

            elif i is selectedObject: #if object is already set as selected object:

```

I also need to make sure that no other buttons can be interacted with while the move button is active and an object has been picked up. To do this, I go to the start of the mouse down event and create a check. The else part of this if statement contains the code for placing the object back down again by setting the pickedObject variable back to 0 and the gamePos of the object to the position it has been placed at (using the same method as getting the spacecraft to the centre of the screen).

```

elif event.type == pygame.MOUSEBUTTONDOWN: #if the mouse

    if buildMenuGroup.sprites()[2].pickedObject == 0:

        buttonGroup.update() #run checks for buttons
        hideUIButtonGroup.update()

        if selectedObject != 0: #if an object is currently selected

```

```

else: #if an object is already picked up:
    buildMenuGroup.sprites()[2].pickedObject.gamePos = pygame.Vector2((pygame.mouse.get_pos()))
    buildMenuGroup.sprites()[2].pickedObject = 0 #place down picked object

```

To make the object move with the mouse cursor when it has been picked up, I set its screenPos to that of the mouse cursor in the main running loop.

```
#####BUILD MENU#####
if buttonGroup.sprites()[3].isClicked: #if build menu is active:

    for i in buildMenuGroup:
        i.mouseHoldUpdate() #update spacecraft control buttons

    if not hideUIButtonGroup.sprites()[0].isClicked: #if UI is not hidden:
        buildMenuGroup.draw(screen) #draw build menu to screen

    if buildMenuGroup.sprites()[2].pickedObject != 0: #if an object has been picked up:
        buildMenuGroup.sprites()[2].pickedObject.screenPos = pygame.mouse.get_pos()
```

Running to test, nothing happens when the move button is clicked; all of the other buttons are also still interactable. I put a few flag prints in for debugging, and found that I had made a logical error in the check for no objects already being picked up in the loop over each physics object on mouse press. This was an easy fix, with a != changed to a ==, and I ran the program again.

This time, the object was registered as being picked up, but would not move with the mouse cursor on the screen. When the object was placed down, it would move to the centre of the screen instead of the mouse cursor position. The first issue was due to the screenPos being recalculated and the object being drawn in the same method call when the object was updated; the only way I can change the screenPos is by including a check in this method. By doing this, I no longer need the position set in the main running loop.

```
def update(self):

    self.gamePos += self.velocity * physEng.getTimeScale() #move sprite in terms of simulation

    if self is buildMenuGroup.sprites()[2].pickedObject: #if picked object, go to mouse cursor position

        self.screenPos = pygame.mouse.get_pos()

    else: #otherwise:

        self.screenPos = physEng.getScreenPos(self.gamePos, screen) #convert gamePos -> screenPos
```

The object now follows the mouse cursor correctly. To make it stay in the correct position once it has been dropped, I tried using the position of the object when it is placed down instead of the position of the mouse cursor, even though they should be the same. This, however, was not successful, and produced the same effect.

To fix it, I had to account for the width and height of the screen when reversing the screenPos calculation. This solved the issue. However, if orbit lines were active when an object was moved, the program crashed and returned this error:

```
Hello from the pygame community. https://www.pygame.org/contribute.html
Traceback (most recent call last):
  File "d:\NEA (USB)\proj\main.py", line 1137, in <module>
    pointsToDraw = physEng.predictOrbit(physicsSprites, 300, screen) #calculate orbit lines
                                         ^^^^^^^^^^^^^^^^^^
  File "d:\NEA (USB)\proj\engine.py", line 123, in predictOrbit
    self.calcApoapsisAndPeriapsis(pointArray, sprites, masters) #calculate + append points for apoapsis and periapsis markers
                                         ^^^^^^^^^^
  File "d:\NEA (USB)\proj\engine.py", line 169, in calcApoapsisAndPeriapsis
    pointDist = sprites[masters[i]].screenPos.distance_to(j) #calculate distance between point and master object
                                         ^^^^^^^^^^
AttributeError: 'tuple' object has no attribute 'distance_to'
PS D:\NEA (USB)\proj>
```

At some point, the screenPos of the object is set as a tuple instead of a Vector2. In case it was the mouse cursor position, I converted it to a Vector2 when setting it as the picked object's screen position; this solved the issue. Another crash happened when the focus was set on the spacecraft, and the spacecraft was removed before the focus button was pressed again:

```
Traceback (most recent call last):
  File "d:\NEA (USB)\proj\main.py", line 969, in <module>
    buttonGroup.update() #run checks for buttons
    ~~~~~~
  File "C:\Users\Sam Prince\AppData\Roaming\Python\Python311\site-packages\pygame\sprite.py", line 538, in update
    sprite.update(*args, **kwargs)
  File "d:\NEA (USB)\proj\main.py", line 380, in update
    self.effect() #apply effect of button
    ~~~~~~
  File "d:\NEA (USB)\proj\main.py", line 244, in effect
    physEng.updateCamera(physObjects.sprites()[self.objectNumber]) #set object to focus on in physics engine
    ~~~~~~
IndexError: list index out of range
```

To fix this I will make the info panel close when the spacecraft is deactivated, and make the check for the focus button include values larger than the length of the physObjects list instead of just equal to.

To create the new planets and stars, I instantiate them in the effect() method for each respective button before setting them as the move button's picked object.

```
case 16|17|18|19: #place planet, place star, move object, delete object (build menu buttons)

    for i in buildMenuGroup: #for each other menu button:
        if i.clicked and i is not self: #if any other button is active:
            i.clicked = False #turn button off
            i.imageUnscaled = i.imageUnclicked #reset image
            i.rescale()

        if self.designation == 16: #place planet
            buildMenuGroup.sprites()[2].pickedObject = Planet(50, 10, pygame.Vector2(0,0), (180,0,100), pygame.Vector2(200,0))
            physObjects.add(buildMenuGroup.sprites()[2].pickedObject) #add to physics objects group

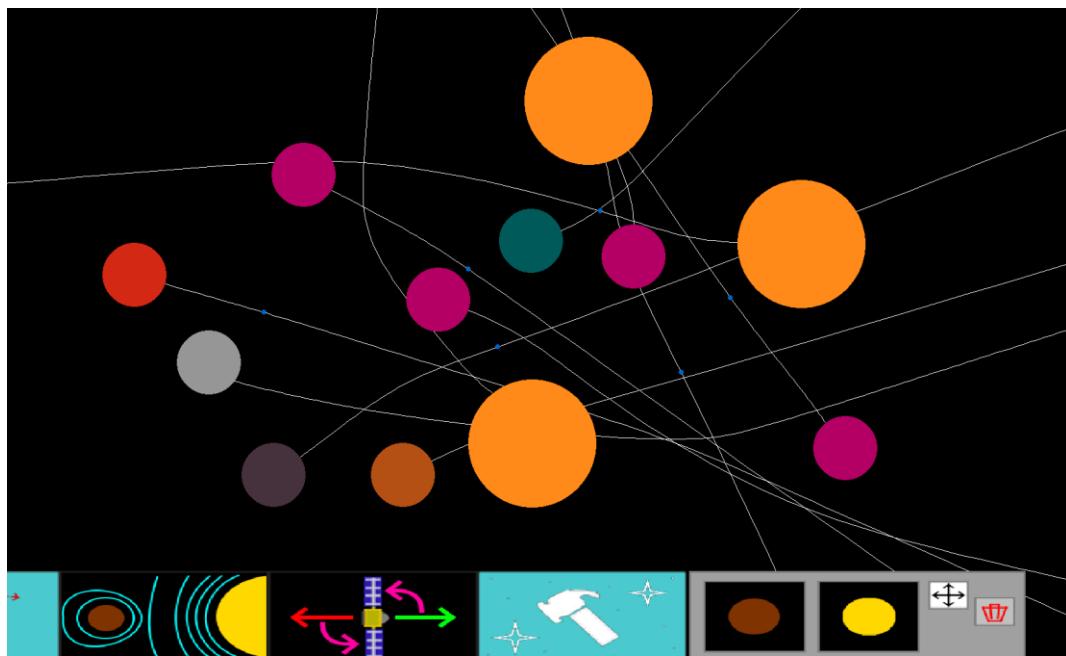
        elif self.designation == 17: #place star
            buildMenuGroup.sprites()[2].pickedObject = Star(100, 100, pygame.Vector2(0,0), 2000, pygame.Vector2(400,0)) #insta
            physObjects.add(buildMenuGroup.sprites()[2].pickedObject) #add to physics objects group
```

Running to test, the correct object is created when either the place star or place planet button has been pressed, and they are also correctly set and are able to be placed as if they were picked up by the move button. When the simulation is unpause, they act in line with the established physics. The final step for the build menu is making the place planet/star buttons go back to their unclicked state after the object is placed down, which is simple enough to do with a check when the object is placed down.

```
for i in range(0,2): #for planet and star placement buttons:

    if buildMenuSprites[i].clicked: #if button is active:

        buildMenuSprites[i].clicked = False #turn button off
        buildMenuSprites[i].imageUnscaled = buildMenuSprites[i].imageUnclicked #reset image
        buildMenuSprites[i].rescale()
```



This all works as intended, and I now consider the build menu (bar the save/load scenarios feature) fully implemented.

Testing + Bug fixing

<i>Test #</i>	<i>Test description</i>	<i>Expected outcome</i>	<i>Actual outcome</i>	<i>Pass/fail</i>
<i>1a</i>	Spawn spacecraft while camera is at origin.	Spacecraft is created in correct location at the centre of the screen.	As expected.	Pass
<i>1b</i>	Spawn spacecraft while camera is at point other than origin.	As above.	As expected.	Pass
<i>1c</i>	Spawn spacecraft while camera is zoomed out, not at origin.	As above.	As expected.	Pass
<i>1d</i>	Deactivate spacecraft button.	Spacecraft object removed from physObjects group.	As expected.	Pass
<i>1e</i>	Respawn spacecraft at new position after despawn.	Spacecraft object added to physObjects group. Spacecraft appears in correct location.	Spacecraft keeps velocity and angle from previous state, but otherwise works as expected.	Partial pass, fixed

I decided to change the code to reset the velocity and the angle of the spacecraft whenever it is spawned, but I will consult with stakeholders to see what they would prefer.

<i>Test #</i>	<i>Test description</i>	<i>Expected outcome</i>	<i>Actual outcome</i>	<i>Pass/fail</i>
<i>2a</i>	Spacecraft movement: accelerate spacecraft in default (facing X-positive) direction.	Spacecraft's velocity X component increases, while Y component remains the same.	As expected.	Pass
<i>2b</i>	Turn and accelerate facing top right direction.	Velocity X and Y components increase in correct proportion.	As expected.	Pass

2c	Turn and accelerate facing top left direction.	As above.	As expected.	Pass
2d	Turn and accelerate facing bottom right direction.	As above.	As expected.	Pass
2e	Turn and accelerate facing bottom left direction.	As above.	As expected.	Pass
3a	Spacecraft info panel: change radius value.	Change not accepted, and textbox reset as "N/A"	When textbox is clicked off, the text resets to "N/A".	Pass
3b	Change surface gravity value.	As above.	Program crashes.	Fail, fixed

```
Traceback (most recent call last):
  File "e:\NEA (USB)\proj\main.py", line 1125, in <module>
    i.textInput(event) #update active text boxes with event data
  ~~~~~
  File "e:\NEA (USB)\proj\main.py", line 487, in textInput
    if selectedObject.massFreezeFlag and selectedObject.radiusFreezeFlag: #if both are frozen, the gravity cannot change while keeping consistent physics
  ~~~~~
AttributeError: 'Spacecraft' object has no attribute 'massFreezeFlag'
PS E:\NEA (USB)\proj>
```

There is currently no check in the gravitational force `textInput()` method for the spacecraft as there is for the radius, so I implemented a simple check to exclude the spacecraft from any changes to its nonexistent gravitational force. Running again, the test was passed.

Test #	Test description	Expected outcome	Actual outcome	Pass/fail
4a	Build menu: select move object button, then close build menu	Move button deselected.	As expected.	Pass
4b	As above with delete object button.	Delete button deselected.	As expected.	Pass
4c	Select object with move tool.	Object is set as <code>pickedObject</code> .	As expected.	Pass
4d	Delete object with deletion tool.	All references to object are removed.	As expected.	Pass
4e	Place planet using planet tool.	New, unique planet object created and added to <code>physObjects</code> group.	As expected.	Pass
4f	Place star using star tool.	New, unique star object created and added to <code>physObjects</code> group.	As expected.	Pass

At the end of testing, I re-ran all tests from this section in order to ensure that none of my bugfixing had made any other undesired changes.

Stakeholder Feedback

After consulting stakeholders, they voted 2-1 to reset the velocity of the spacecraft as it is respawned. Otherwise, they were happy with the spacecraft and the build menu.

Stage 6 – Accessing the program, saving and loading scenarios

Success criteria

REQUIREMENT	DESCRIPTION	SUCCESS CRITERIA	CHECK
-------------	-------------	------------------	-------

5A	Preset scenarios	At least 3 preset scenarios are able to be loaded at some point in the program.	✓
5C	Save and load custom scenarios (E)	Options to save and load at least 3 custom scenarios that have been created in the program.	X
7C	Can be accessed online (E)	Program can be accessed by any computer with an internet connection.	✓
7D	Desktop shortcut to run the program (E)	Program can be run simply by clicking on a shortcut icon on desktop.	X

Testing plan

<i>Test #</i>	<i>Test description</i>	<i>Expected outcome</i>
1a	Preset scenarios: load scenario 1.	loadScenario() called; all objects deleted and scenario from file 1 loaded.
1b	Load scenario 2.	loadScenario() called; all objects deleted and scenario from file 2 loaded.
1c	Load scenario 3.	loadScenario() called; all objects deleted and scenario from file 3 loaded.
2a	Save and load custom scenarios: Save a scenario to file.	Scenario text file created, with correct data written to it.
2b	Load previously saved scenario.	loadScenario() called; all objects deleted and scenario from selected file loaded.
3a	Online access: Open program through itch.io.	Program runs with same performance and features as if run locally.
3b	Load each preset scenario in turn from browser.	loadScenario() called; all objects deleted and scenario from selected file loaded.
3c	Save scenario to file from browser.	Scenario text file created, with correct data written to it. File downloaded to user device.
3d	Load previously saved scenario in browser.	loadScenario() called; all objects deleted and scenario from selected file loaded.
4a	Desktop shortcut: Click shortcut to run program.	Program runs locally as if run through IDE.

Development

One of the main issues with my code so far is the fact users must install Python and the required libraries to run it. To fix this, I will make the simulation accessible online through itch.io, using the Pygbag library. I will attempt to do this before implementing the save and load/preset scenarios feature as if it works, it may become difficult to use a text file to save/load data – Pygbag has limitations that mean it is impossible to download a file to a user's device if the program is not being run locally.

If a file cannot be downloaded, then there is no way to locally save data for a user as the program completely resets every time it is run online, and so the save and load feature could not be implemented as intended. For the preset scenarios, as they should stay constant throughout all instances of the program, they should still be possible to implement.

To implement Pygbag, I followed the instructions on the Pygbag wiki (<https://pygame-web.github.io/wiki/pygbag/>, accessed 16/3/24 at 12:00) to import and add async functions to the program so that Pygbag would work with it.

```
import pygame      font = pygame.font.SysFont('Consolas', 30) #initialise font
import engine
import numpy
import math
import asyncio
font = pygame.font.SysFont('Consolas', 30) #initialise font
async def main(): #async function for Pygbag functionality
    #setup for star colour function - global variables so that
    redco = [ 1.62098281e-82, -5.03110845e-77, 6.66758278e-72
    greenco = [ 1.21775217e-82, -3.70265202e-77, -5.0429989e-82
    blueco = [ 1.21775217e-82, -3.70265202e-77, -5.0429989e-82
    await asyncio.sleep(0) #delay (pygbag)    asyncio.run( main() ) #run main program (pygbag)
```

After doing this, there were suddenly a lot of undefined calls in the object classes; this was because many of the objects created by the main loop were now local variables as part of the async main() method. To fix this, I passed through the relevant objects and groups into each affected method as parameters. I made many parameters optional, as they were non needed for most calls (especially the case with the button class).

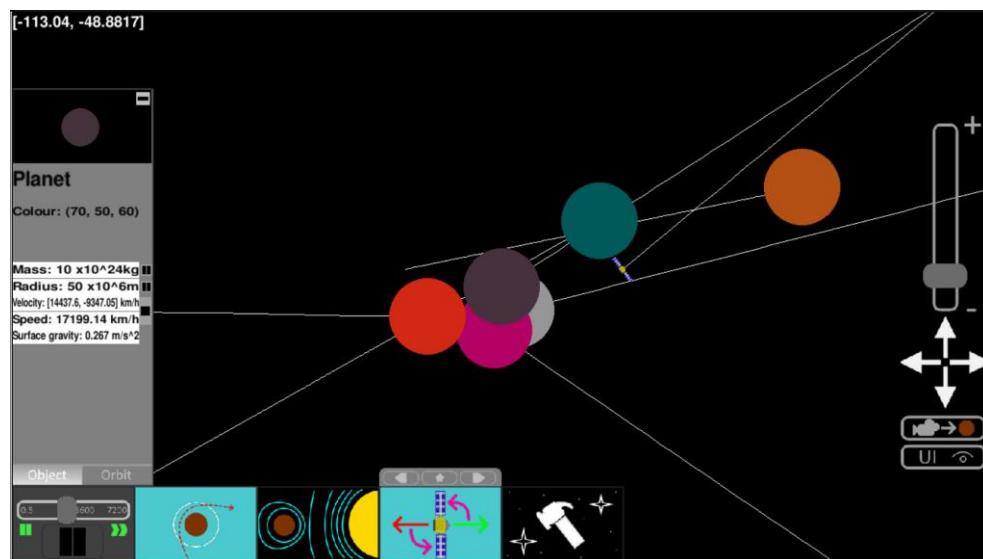
To make the program run in my local browser, I went to the folder above my project and ran the pygbag command on it.

```
PS C:\Users\chick\OneDrive\Desktop\NEA> python -m pygbag proj
*pygbag 0.8.6*

Serving python files from [C:\Users\chick\OneDrive\Desktop\NEA\proj\build\web]

with no security/performance in mind, i'm just a test tool : don't rely on me
```

I then clicked the link that was generated in the response, and my browser opened to the program:



The performance was relatively low (but still above the 20fps threshold) and it looked as if the font had changed, but as the warning above states, this is just a test server and so I assume performance would be better once it is uploaded to itch.io. Otherwise, the program worked exactly as it did locally. If I run the program through visual studio code instead of this command it still runs locally as before.

I decided to upload this version onto itch.io initially, to test how well it actually performs. To do this, I will need to upload a folder that was generated by Pygbag when I ran the program, located in my project directory. Pygbag generates a build folder, which I need to go into to zip a web folder.

📁 __pycache__	08/11/2023 11:05	File folder
📁 build	16/03/2024 13:42	File folder
📁 legacy	17/01/2024 22:58	File folder
📄 engine	15/03/2024 11:08	Python Source File 10 KB
🖼️ icon	21/11/2023 12:42	paint.net Image 70 KB
📄 main	16/03/2024 13:39	Python Source File 68 KB
🖼️ SAVELOADSCENARIOBUTTON	12/03/2024 18:44	paint.net Image 5 KB
<hr/>		
📁 web ← ZIP	16/03/2024 13:42	File folder
📁 web-cache	16/03/2024 13:42	File folder
📄 version	16/03/2024 13:42	Text Document 1 KB

The favicon that shows up at the top of the screen for the localhost is the Pygbag logo, but I can replace the image in the web folder before I zip it to make it the logo for my project instead. Finally, I just have to upload the zipped file to the project page on itch.io.

Uploads

Upload a ZIP file containing your game. There must be an `index.html` file in the ZIP. Or upload a `.html` file that contains your entire game. [Learn more →](#)

Any additional files you upload will be made available for download. You can apply a minimum price to the project after uploading additional downloadable files.

web.zip · Success

433kb · [Change display name](#)

Today at 1:49 PM

This file will be played in the browser

[More...](#) [Delete file](#)

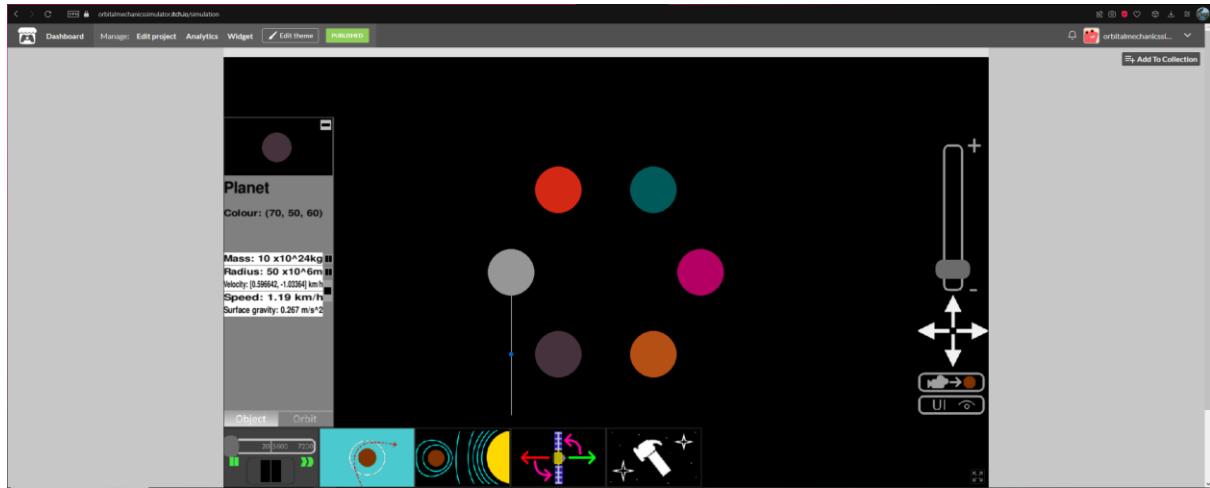
Upload files

or

 Choose from Dropbox

[Add External file](#) ?

After uploading, I published the project and ran it in my browser through itch.io to see if performance was any different.



The program performed exactly as it did in the Pygbag local server. However, when I pressed the button to hide the UI, performance noticeably increased. As I had this problem before back in stage 1, I went back and checked through my buttons to make sure that I had them all converted to the correct pixel format when they are loaded from file, which I had done.

The only thing that the hide UI button does is stop the `draw()` methods of the button sprite groups from being called; therefore, the issue must be with the way that the buttons are being drawn to the screen. I commented out the `draw()` method for each button group and tested the Pygbag local server in turn to see if it was an issue across every group, or just one in particular.

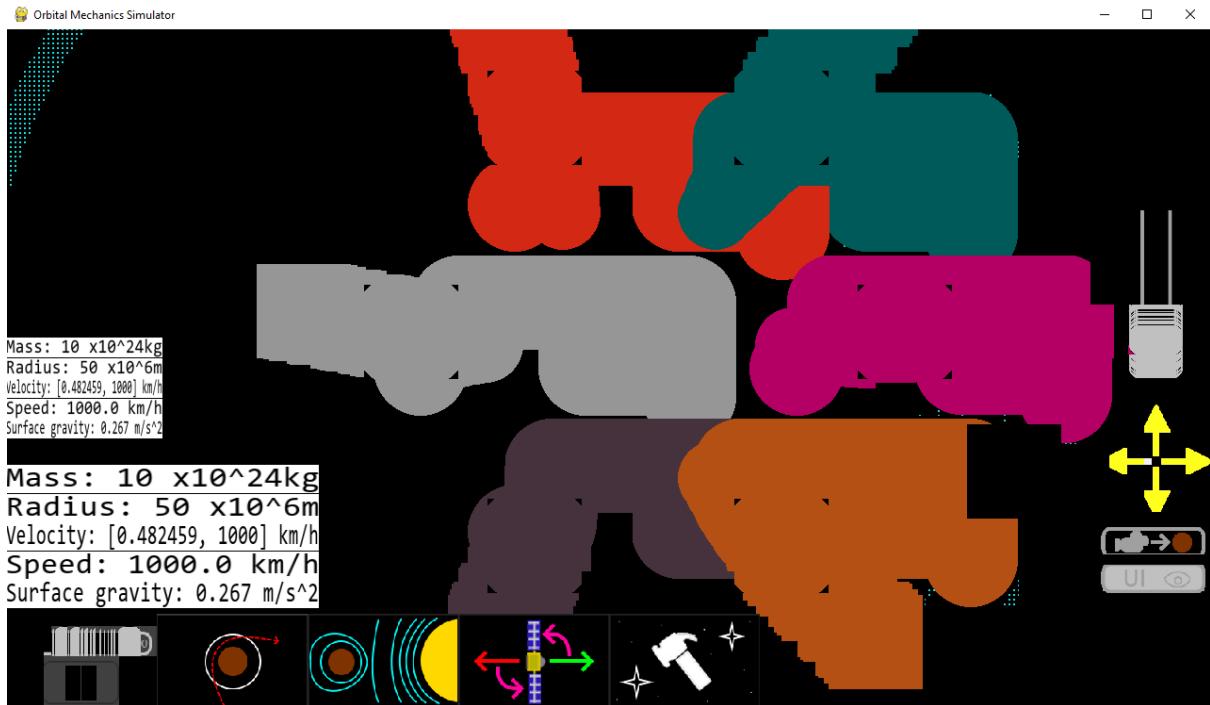
When I removed the draw for the `buttonGroup`, performance increased until I opened the info panel, at which point performance decreased again. I decided to remove the draw for the hide UI button as well as the info panel, to see whether it was due to the number of groups that had to be drawn in the same tick. Performance was still low with just the button group shown, which disproved this theory. There must be something in common between the info panel and the button group that is not in the hide UI button group. The main difference between these two groups and the hide UI group is that the hide UI group was created for only one sprite; I thought that maybe the `draw()` method that loops over and blits each sprite in the group is the issue, so I instead directly coded this into the main running loop for each button sprite group.

```
if not hideUIButtonGroup.sprites()[0].isClicked: #if UI is not hidden:  
    for i in buttonGroup.sprites(): #for each sprite:  
        screen.blit(i.image, i.rect) #blit sprite to screen  
    #buttonGroup.draw(screen) #draw buttons to screen
```

However, when I tested this in the browser, it made no difference in performance, so I just replaced it with the `draw()` method instead. I then tried converting each button image to the correct pixel format after it was transformed in the `rescale()` method, in case the transformations had changed it somehow. This, again, had no effect. I tried setting the `clock.tick` parameter to 30, which again had no effect.

I then resorted to only updating parts of the screen where changes were made in the previous frame; this is done by adding the rects (or the collision rects in the case of the buttons) to a list, then using the `pygame.display.update()` method instead of the more general `flip()` to only update the specific parts of the screen where movement or a change in state has happened. This list

of rects to update will be a global array called updateArea. I made it so that the rects of each button and planet would be added to the array as part of their update() methods, and ran to test.



As you can see, this did not work, so I reversed the changes I did and removed the array. I then did some more research into issues with pygbag and alpha blitting, but there was nothing online about it. I suspected that the use of many images with their alpha values all being blitted and overlapping might be causing some issues. To fix this, I decided to crop all the button images so that they included little to no alpha pixels; to make sure they still resized correctly, I will resize them based on their existing collision rects. In case this does not work or provides negative results, I will create a copy of my entire project before doing this.

In the rescale() method for each button case, I will resize the image based on the collideRect coordinates from before, and set the rect the same.

```
def rescale(self, physEng):
    case 0: #pause button:

        self.image = pygame.transform.scale(self.imageUnscaled, (screenWidth/16, screenHeight/9)) #update rectangular coordinates for pause button

        self.rect.update(screenWidth/32, screenHeight*(33/36), screenWidth/16, screenHeight/9)

    case 1 | 2 | 3 | 4 : #main menu buttons:

        self.image = pygame.transform.scale(self.imageUnscaled, (screenWidth/8, screenHeight*(5/36))) #update rectangular coordinates for main menu buttons

        self.rect.update((screenWidth/8) * self.designation, screenHeight*(31/36), screenWidth/8, screenHeight*(5/36))

    case 5: #camera focus button:

        self.image = pygame.transform.scale(self.imageUnscaled, (screenWidth, screenHeight))

        self.collideRect.update(screenWidth*0.9081, screenHeight*0.7333, screenWidth*0.0869, screenHeight*0.0422) #update rectangular coordinates for camera focus button

    case 6 | 7: #north/south camera movement buttons:

        self.image = pygame.transform.scale(self.imageUnscaled, (screenWidth*0.0225, screenHeight*0.08))

        self.rect.update(screenWidth*0.94375, screenHeight * 0.079 * (self.designation + 1), screenWidth*0.0225, screenHeight*0.08)
```

As the rect is now the same as the collideRect, there is no longer any need for a collideRect variable in many button cases. I will instead just check for the mouse cursor intersecting with the rect of an

object. Exceptions to this include the camera focus button; it will have to take up the whole screen owing to it being part of the pause button and camera slider backgrounds as well. In this and in any other case, I will add a check in the update() method for this designation.

```
if self.designation == 5:  
    if self.collideRect.collidepoint(pygame.mouse.get_pos()): #if mouse cursor is intersecting collision rect:  
  
        if self.isClicked: #if button is in clicked state  
            self.imageUnscaled = self.imageUnclicked #change graphic  
            self.isClicked = False #set logic to be in unclicked state  
  
        else: #if button is not in clicked state  
            self.imageUnscaled = self.imageClicked #change graphic  
            self.isClicked = True #set logic to be in clicked state  
  
        self.effect(physEng, physObjects, spacecraft, buttonGroup, buildMenuGroup, redco, greenco, blueco)  
        self.rescale(physEng) #update button image  
  
    elif self.rect.collidepoint(pygame.mouse.get_pos()): #if mouse cursor is intersecting collision rect:  
  
        if self.isClicked: #if button is in clicked state  
            self.imageUnscaled = self.imageUnclicked #change graphic  
            self.isClicked = False #set logic to be in unclicked state  
  
        else: #if button is not in clicked state  
            self.imageUnscaled = self.imageClicked #change graphic  
            self.isClicked = True #set logic to be in clicked state
```

I resized only the main button group and the hide UI button in this way to test if it would actually work. After running it through the Pybag local test server, performance was finally back to the normal speed as it was when run via Python itself. This did, however, result in the misalignment of a few buttons, but it was a relatively easy fix. I decided to leave the majority of the other buttons unchanged as they did not impact performance as negatively as the main menu did.

After running into a lot of issues with button collisions, I decided to re-implement the collideRect variable for all buttons for better maintainability. I also switched the unique variable initialisation for some buttons in the __init__() method to a match...case statement instead of a long if statement.

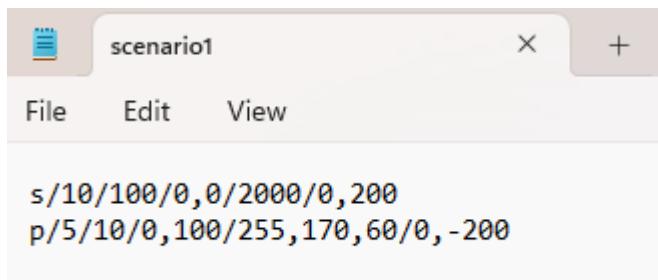
```
match self.designation: #collision rect positioning  
  
    case 0: #pause button:  
  
        self.image = pygame.transform.scale(self.imageUnscaled, (screenWidth/16, screenHeight/9)) #update rectangular  
        self.rect.update(screenWidth/32, screenHeight*(33/36), screenWidth/16, screenHeight/9)  
        self.collideRect = self.rect #set collision rect as rect of image  
  
    case 1 | 2 | 3 | 4 : #main menu buttons:  
  
        self.image = pygame.transform.scale(self.imageUnscaled, (screenWidth/8, screenHeight*(5/36))) #update rectangular  
        self.rect.update((screenWidth/8) * self.designation, screenHeight*(31/36), screenWidth/8, screenHeight*(5/36))  
        self.collideRect = self.rect #set collision rect as rect of image  
  
    case 5: #camera focus button:  
  
        self.image = pygame.transform.scale(self.imageUnscaled, (screenWidth, screenHeight))  
        self.collideRect.update(screenWidth*0.9081, screenHeight*0.7333, screenWidth*0.0869, screenHeight*0.0422) #update rectangular  
        self.rescale(physEng) #update button image  
  
    case 6 | 7: #north/south camera movement buttons:  
        self.image = pygame.transform.scale(self.imageUnscaled, (screenWidth, screenHeight))  
        self.collideRect.update(screenWidth*0.9081, screenHeight*0.7333, screenWidth*0.0869, screenHeight*0.0422) #update rectangular
```

I reuploaded this to the project page on itch.io, and the performance was increased. However, there was still a bit of slowdown when either the info panel or the build menu were opened, although they had a reduced impact on performance compared to before the changes were made; in the future, I might explore cropping the alpha values out of these buttons as well.

The next criteria to implement is the loading of preset scenarios; I will do this in the way I intended to program from the start, with loading from a text file. I created three text files called scenario1, scenario2 and scenario3 for each of the preset scenarios. At program start, the scenario saved in the scenario1 file will be loaded.

In my design, loading a scenario involves reading each line individually; each line holds all the data of one object, including its class, gamePos, radius, mass and variable freeze flags.

In scenario1, I wrote in the lines of data for two objects: a star and a planet. The class is denoted by the letter at the start (S or P), followed by the radius, mass, velocity and gamePos (temperature is also included in the case of the star, and colour in the case of the planet). The data is separated by forward slashes; initially I was going to use commas or full stops to do this, but it would not allow for floats or sequences of numbers (such as velocity) to be easily saved. I will get these initial core variables working with the save system before implementing the freeze flags.



For the scenario loading, I will create a new method independent of any classes called `loadScenario()`. This opens the requested file (passed as parameter), then reads each line. It splits the data into an array called `objData` before reformatting any data in the wrong format (including colour in the case of the planet object, and temperature for the star). An object is then instantiated based on the data in the `objData` array.

```
def loadScenario(scenarioNumber, physObjects, redco, greenco, blueco):
    try: #in case of error, program will continue to run
        file = open(f"scenario{scenarioNumber}.txt", "r") #open file object to read
        for line in file.readlines(): #for each object saved in file:
            objData = line.split("/") #write data for object to array

            #split velocity and gamePos variables to put in correct format
            for i in range(3,6,2): #velocity = objData[3], gamePos = objData[5], step of 2 to skip over objData[4]
                tempArray = objData[i].split(",") #split data into two components
                objData[i] = pygame.Vector2(tempArray[0], tempArray[1]) #format as Vector2

            if objData[0] == "s": #if object is a star:
                physObjects.add(Star(objData[1], objData[2], objData[3], objData[4], objData[5], redco, greenco, blueco)) #create object from data

            elif objData[0] == "p": #if object is a planet:
                #reformat colour data
                tempArray = objData[4].split(",") #split data into three components
                objData[4] = (tempArray[0], tempArray[1], tempArray[2]) #format as RGB tuple

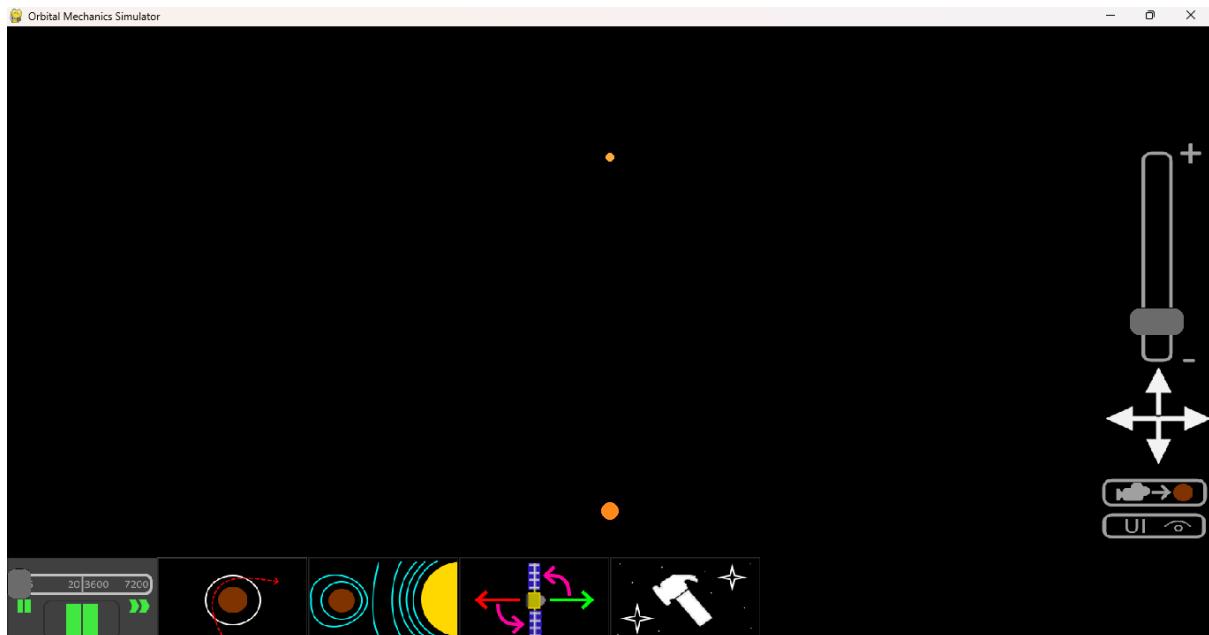
                physObjects.add(Planet(objData[1], objData[2], objData[3], objData[4], objData[5])) #create object from data

    finally: #always close file to avoid corruption of data
        file.close()
```

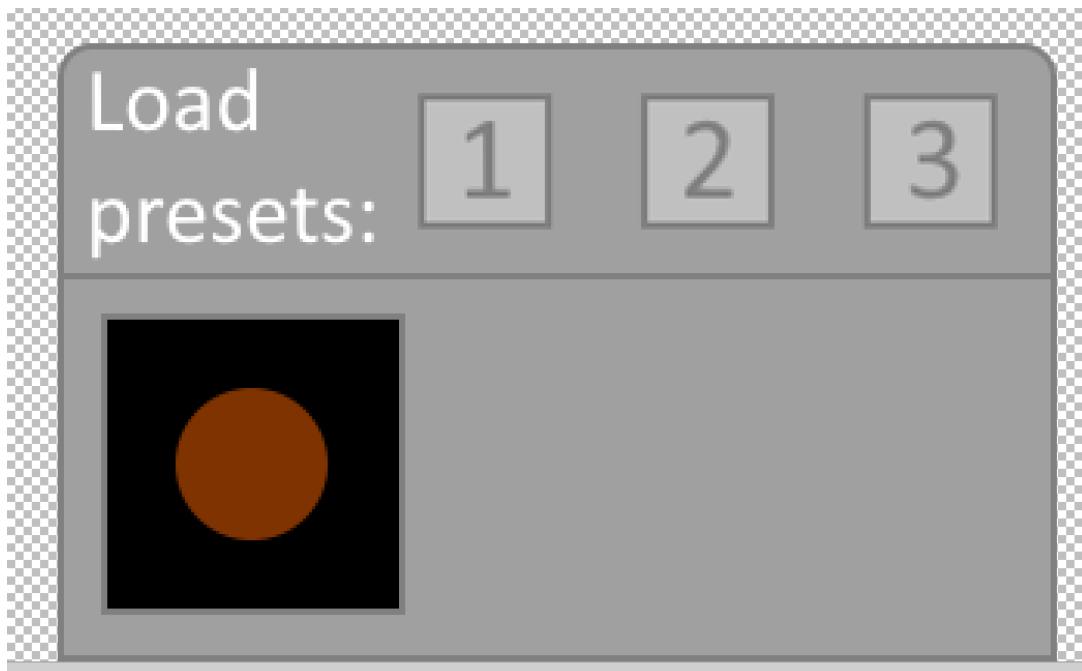
This will be called instead of directly instantiating the initial objects just before the main running loop. Running to test, I was met with this error:

```
File "d:\NEA (USB)\proj\main.py", line 869, in loadScenario
    objData[i] = pygame.Vector2(tempArray[0], tempArray[1]) #format as Vector2
                                         ^^^^^^^^^^^^^^^^^^
ValueError: Vector2 must be set with 2 real numbers, a sequence of 2 real numbers, or another Vector2 instance
PS D:\NEA (USB)\proj>
```

I tried casting this data as floats, as it seemed it was reading it as strings. Running it now:



The objects were initialised as expected. Now to make the button to load these scenarios functional: I will need an extra 3 buttons in addition to the load scenario button in order to select which scenario to load; with the load scenario button being designation 34, these will be designations 35-37. I created these buttons in paint.net, with the idea that they will be shown when the save/load button is active. Below is what it will look like in relation to the build menu:



I will also need a new group for these three buttons (hopefully the last) called loadPresetButtons to make them be drawn only when the save/load button is active. Clicking one of the three buttons above will cause all of the objects in the scene to be deleted, before the loadScenario() method is called with the corresponding scenario number as the parameter.

```
case 35 | 36 | 37: #Load scenario 1/2/3

    #delete all objects currently in scene
    for i in physObjects:

        if i.__class__.__name__ != "Spacecraft": #exclude spacecraft from deletion checks

            if i is selectedObject:
                selectedObject = 0 #clear info panel
            if i is physEng.focusedObject:
                physEng.focusedObject = 0 #clear object focus

            physObjects.remove(i) #remove from physics objects group

    loadScenario(self.designation - 34, physObjects, redco, greenco, blueco) #to get number for scenario, subtract 34 from designation

    self.isClicked = False #reset button
    self.imageUnscaled = self.imageUnclicked
```

After creating two more scenarios in the text files (one with a single star and planet, and the other meant to be an accurate solar system), I ran the program to test. When each button was pressed in turn, the objects present on the screen were removed before the objects described in the text file were created in the correct places, and with the correct parameters. After moving the camera around a bit, I realised that its position would be kept even when a scenario was loaded, so inside the effect() method for the load scenario buttons I added a line to clear the physics engine camera and zoom offsets, and to pause the simulation.

```
case 35 | 36 | 37: #Load scenario 1/2/3

    physEng.movementOffset = pygame.Vector2(0,0) #reset camera offset
    physEng.zoomScale = 1 #reset zoom offset

    if not physEng.isPaused: #pause simulation if unpaused
        buttonGroup.sprites()[9].isClicked = True #change pause button to pause state
        buttonGroup.sprites()[9].imageUnscaled = buttonGroup.sprites()[9].imageClicked
        buttonGroup.sprites()[9].rescale(physEng)
        physEng.isPaused = True #switch physics engine state

    #delete all objects currently in scene
    for i in physObjects:
```

Now that I am confident that these have been implemented, I added in the freeze flags for each object into the saved data, so that object variable freeze states would also be able to be saved and loaded in these scenarios. I had to make some changes to the loadScenario() method to accommodate for this, and add some new data points in the files.

```
#split velocity and gamePos variables to put in correct format
for i in range(3,6,2): #velocity = objData[3], gamePos = objData[5], step of 2 to skip over objData[4]
    tempArray = objData[i].split(",") #split data into two components
    objData[i] = pygame.Vector2(float(tempArray[0]), float(tempArray[1])) #format as Vector2

if objData[0] == "s": #if object is a star:

    obj = Star(float(objData[1]), float(objData[2]), objData[3], float(objData[4]), objData[5], redco, greenco, blueco) #create object from data

elif objData[0] == "p": #if object is a planet:

    #reformat colour data
    tempArray = objData[4].split(",") #split data into three components
    objData[4] = (float(tempArray[0]), float(tempArray[1]), float(tempArray[2])) #format as RGB tuple

    obj = Planet(float(objData[1]), float(objData[2]), objData[3], objData[4], objData[5]) #create object from data

#set variable freeze flags
if objData[6] == 1: #mass freeze flag
    obj.massFreezeFlag = True
if objData[7] == 1: #radius freeze flag
    obj.radiusFreezeFlag = True
if objData[8] == 1: #speed/velocity freeze flag
    obj.velocityFreezeFlag = True
if objData[9] == 1: #time period freeze flag
    obj.timePeriodFreezeFlag = True
if objData[10] == 1: #orbital radius freeze flag
    obj.orbitalRadiusFreezeFlag = True
if objData[11] == 1: #orbital velocity freeze flag
    obj.orbitalVelocityFreezeFlag = True

physObjects.add(obj)
```

After running this to test, the changes in data seemed to have no effect on the flags for the objects; this was because it was checking a string of 1 against an integer 1. This was solved by checking against a string 1 instead. The loading of scenarios has now been fully implemented.

For the saving/loading of custom scenarios, due to the online nature of my solution, I will not be able to save files across different instances of the program. Therefore, I have decided to leave this feature unimplemented. If I had more time, I would do some more research into potential ways to download a file onto a user's computer, but as far as I am aware with the current limits of itch.io and the pybag library, this is impossible with my current software.

Testing + Bug fixing

Test #	Test description	Expected outcome	Actual outcome	Pass/fail
1a	Preset scenarios: load scenario 1.	loadScenario() called; all objects deleted and scenario from file 1 loaded.	As expected.	Pass
1b	Load scenario 2.	loadScenario() called; all objects deleted and scenario from file 2 loaded.	As expected.	Pass
1c	Load scenario 3.	loadScenario() called; all objects deleted and scenario from file 3 loaded.	As expected.	Pass
2a	Save and load custom scenarios: Save a scenario to file.	Scenario text file created, with correct data written to it.	Feature not implemented.	N/A
2b	Load previously saved scenario.	loadScenario() called; all objects deleted and scenario from selected file loaded.	Feature not implemented.	N/A

3a	Online access: Open program through itch.io.	Program runs with same performance and features as if run locally.	As expected.	Pass
3b	Load each preset scenario in turn from browser.	loadScenario() called; all objects deleted and scenario from selected file loaded.	As expected.	Pass
3c	Save scenario to file from browser.	Scenario text file created, with correct data written to it. File downloaded to user device.	Feature not implemented.	N/A
3d	Load previously saved scenario in browser.	loadScenario() called; all objects deleted and scenario from selected file loaded.	Feature not implemented.	N/A
4a	Desktop shortcut: Click shortcut to run program.	Program runs locally as if run through IDE.	Feature not implemented.	N/A

Stakeholder Feedback

Kai and Oli were both happy with the ease of access of the project on the internet, and the ability to load the three preset scenarios. Mr Douglas-Scott said that this feature in particular was essential for him to be able to access the project, and that it was definitely easy enough to load up and use. The preset scenarios are also good enough to demonstrate the concepts that he intends to as part of beta testing.

However, the stakeholders all expressed disappointment at not being able to save and load scenarios. This is impossible for me to implement due to the limitations of itch.io and the libraries I am using.

Stage 7 – Misc features, changes and fixes

Success criteria

REQUIREMENT	DESCRIPTION	SUCCESS CRITERIA	CHECK
4A	Atmospheric height/effects (E)	An atmosphere around planets is present, that slows down the spacecraft when passing through.	✓
4C	Collision physics (E)	When two objects collide, they react instead of passing through one another.	X
6	Simplistic graphics	Graphics are simple enough to not be distracting, but detailed enough that at a glance it can be understood what each object represents.	✓
6A	Stars as background (E)	Small dots are present in the background to represent stars and more effectively show that it is outer space.	X

6B	Textures for planets (E)	Planets have at least one texture other than solid colour.	X
-----------	--------------------------	--	---

Testing plan

Test #	Test description	Expected outcome
1a	Atmospheric height: Atmosphere not drawn with no star object in simulation.	getAtmosphereHeight() method not called.
1b	Dual body system (1 star), planet placed at same position as star (boundary).	Height of atmosphere calculated as a large value.
1c	Star temperature 0 (boundary).	Height of atmosphere 0.
1d	Star temperature 2000 (normal). Simulation unpause, planet moves towards star.	As planet moves towards star, height of atmosphere increases.
1e	Three-body system (2 stars), planet placed equidistant between stars.	Heights of atmosphere due to each star is added together to produce the correct value.
2a	Atmospheric effects: Spacecraft experiences slowdown when passing through atmosphere.	Spacecraft speed slows once per tick when passing through atmosphere. Velocity decreases proportionally.
2b	Spacecraft passes through atmosphere with speed/velocity locked.	No change in speed/velocity.
3a	Collision physics: dual body system, 0 initial speed.	Objects bounce off of one another elastically and oscillate between midpoint and starting position.
3b	Both objects placed at same position.	Objects do not move.
3c	Both objects at same position, one with forwards velocity.	As object moves to the right, the other bounces and gains velocity proportional to momentum. Other object loses velocity proportional to momentum.
3d	Spacecraft and object collision. No initial velocities.	Objects bounce off of one another elastically and oscillate between midpoint and starting position.
4a	Stars as background: load program and move camera around.	'stars' are shown, and move with camera.
5a	Planet textures: load program with 6 planet system.	Planets load with the supplied texture applied to them. Image is converted correctly by Pygame convert() method, and does not affect performance in the Pygbag test environment.

Development

A small detour: after looking through some forums online, and this⁸ tutorial especially, I found a potential solution to the slow performance with calculating the equipotential lines; the use of multithreading using the inbuilt Python threading library. This would allow me to split tasks and hopefully improve processing time. I made a copy of my code before implementing this, in case it did not work as intended.

⁸ (CodersLegacy, n.d.)

To implement this, I will need a new method for each thread to process; this will be one for each pixel on the screen. The existing `calcEquipotential()` method is unsuitable for this as it is the method in which each pixel is looped over; consequently, I will move the calculations within this loop to a new method in the physics engine called `threadCalc()`.

```
def threadCalc(self, i, j, array, objects):
    force = 0 #set initial force on point to 0
    for k in objects: #for each physics object:
        force += self.calcGravBetweenObjects(k, k.screenPos.distance_to(Vector2(i,j))) #calculate force for each individual object
    if (9e9 < force < 10e9) | (19e9 < force < 20e9) | (29e9 < force < 30e9) | (39e9 < force < 40e9) | (49e9 < force < 50e9) | (59e9 < force < 60e9):
        array.append(Vector2(i,j)) #add to 2d array based on force it meets, if is the same as one of the forces
```

I create the threads inside the `calcEquipotential()` method:

```
def calcEquipotential(self, objects, screen, resolution):
    equipotentialArray = []
    threads = []

    #for each point on the visible screen:
    for i in range(0, screen.get_width(), resolution):
        for j in range(0, screen.get_height(), resolution):

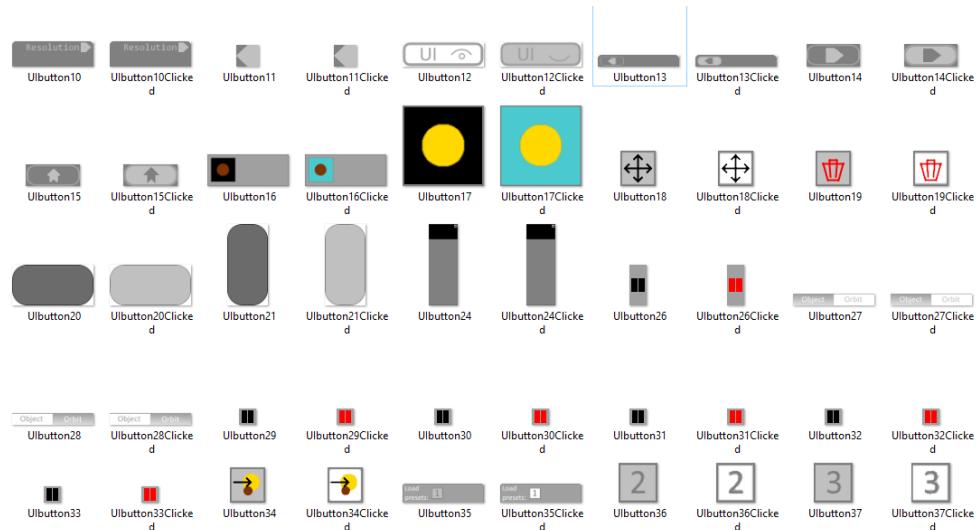
            thread = threading.Thread(target = self.threadCalc, args=(i,j,equipotentialArray,objects))
            threads.append(thread)
            thread.start()

    for thread in threads:
        thread.join() #join together threads to ensure unified end time

    return equipotentialArray
```

Running to test, this did not improve performance; in fact, it significantly harmed it. Therefore, I reverted back to the original version without any threading implemented.

On the topic of improving performance, I decided that now was the time to reformat the rest of the buttons so that the program would not experience slowdown when run through a browser. This involved updating the rects for the newly redone objects to be the same as the collision rects, but in the cases where buttons did not have the same clickable area as their image, I had to position the new rect from scratch.



The changed button rescale() method:

```
def rescale(self, physEng):
    screenWidth = screen.get_width() #cache screen width
    screenHeight = screen.get_height() #cache screen height

    match self.designation: #collision rect positioning

        case 0: #pause button:
            self.image = pygame.transform.scale(self.imageUnscaled, (screenWidth*0.063, screenHeight*0.063)) #rescale to window size
            self.rect.update(screenWidth*0.03, screenHeight*0.93, screenWidth*0.063, screenHeight*0.063)
            self.collideRect.update(screenWidth/32, screenHeight*(33/36), screenWidth/16, screenHeight/9)

        case 1 | 2 | 3 | 4 : #main menu buttons:
            self.rect.update((screenWidth/8) * self.designation, screenHeight*(31/36), screenWidth/8, screenHeight*(5/36)) #update rectangular coordinates for collision rect based on designation

        case 5: #camera focus button:
            self.image = pygame.transform.scale(self.imageUnscaled, (screenWidth, screenHeight))
            self.collideRect.update(screenWidth*0.9081, screenHeight*0.7333, screenWidth*0.0869, screenHeight*0.0422) #update rectangular coordinates for collision rect based on designation

        case 6 | 7: #north/south camera movement buttons:
            self.rect.update(screenWidth*0.944, screenHeight * 0.079 * (self.designation + 1), screenWidth*0.0225, screenHeight*0.08) #update rectangular coordinates for collision rect based on designation

        case 8 | 9: #east/west camera movement buttons:
            self.rect.update(screenWidth - screenWidth * 0.0445 * (self.designation - 7), screenHeight*0.6156, screenWidth*0.0425, screenHeight*0.0411) #update rectangular coordinates for collision rect based on designation

        case 10: #equipotential resolution increase:
            self.image = pygame.transform.scale(self.imageUnscaled, (screenWidth*0.125, screenHeight*0.069))
            self.rect.update(screenWidth*0.25, screenHeight*0.828, screenWidth*0.125, screenHeight*0.069)
            self.collideRect.update(screenWidth*0.355, screenHeight*0.82, screenWidth*0.0125, screenHeight*0.0411) #update rectangular coordinates for collision rect based on designation

        case 11: #equipotential resolution decrease:
            self.image = pygame.transform.scale(self.imageUnscaled, (screenWidth*0.125, screenHeight*0.069))
            self.rect.update(screenWidth*0.25, screenHeight*0.828, screenWidth*0.125, screenHeight*0.069)
            self.collideRect.update(screenWidth*0.355, screenHeight*0.82, screenWidth*0.0125, screenHeight*0.0411) #update rectangular coordinates for collision rect based on designation
```

Another new addition after the end of the match..case statement is a unified line to set the collideRects and perform the image translation for the buttons that have the same image as their clickable area. To filter this, I created an array of the designations that did not do this called exceptionButtons:

```
#collision rect set + image transformations
exceptionButtons = [0,5,10,13,16,21,24,27,28,35] #list of buttons that have different rects and collideRects: their images and collideRects are set within the match..case statement above

if not self.designation in exceptionButtons:

    self.image = pygame.transform.scale(self.imageUnscaled, (self.rect[2], self.rect[3])) #rescale image based on screen size
    self.collideRect = self.rect #create collision rect as area of image
```

Running on the Pybag test server, these changes dramatically improved performance compared to the previous version, and the program now only lost performance when the equipotential lines were active.

Now for the other extra features to implement: first off, the height of atmospheres for planets. The equation for atmospheric height is $H = \frac{kT}{Mg}$, where k is the Boltzmann constant (1.38×10^{-23}), T is the absolute/effective temperature of the object, M is the object's mass and g is the acceleration due to gravity on the planet's surface.

My research of the absolute temperature⁹ shows that this value depends on the distance of the object from a star as well as the star's luminosity. As mass and distance between two objects is already a central part of planet objects, it should be relatively simple to work out the absolute temperature. After going down a Wikipedia rabbit hole about blackbody radiation, the greenhouse effect and many more wonderful calculations that kind of mostly made sense, I got these three equations:

$$H = \frac{RT_e}{Mg_p}, L = \sigma AT_s^4, T_e = \sqrt[4]{\frac{L(1-a)}{16\pi\sigma D^2}}$$

is the atmospheric height, M is the average mass of one mol of atmospheric particles, L is the luminosity of a star, R is the gas constant, σ is the Stefan-Boltzmann constant, A is the surface area of

⁹ (Effective Temperature, n.d.) (Luminosity, n.d.) (Scale Height, n.d.)

the star, D is the distance between the object and the star, a is the albedo of the object and g is the gravitational field strength at the surface of the planet.

Rearranging these equations to get a single unified equation for atmospheric height:

$$H = \frac{8.31 \times T_s \times \sqrt[4]{\frac{0.7r_s^2}{4D^2}}}{0.029 \times g_p}$$

This is assuming that the albedo of all objects is 0.3, the same as Earth's average. Albedo is a measure of how much sunlight is reflected by a surface. This also assumes the average mass of one mol of atmospheric gas as 0.029kg/mol, again the same as the Earth's.

To calculate this, I will create another new method inside the planet class called `getAtmosphereHeight()`. This will loop over each star object in the simulation, calculating the atmospheric height effect for each one before adding them all together to produce a final atmospheric height for that planet. I will need to correct all the distances in the calculation to account for the $\times 10^6$ conversions with length/radius.

```
def getAtmosphereHeight(self, physicsObjects): #calculate atmospheric height (PLANET OBJECTS ONLY)
    atmosphereHeight = 0 #clear atmospheric height

    for i in physicsObjects: #for each object in simulation:
        if i.__class__.__name__ == "Star": #for each star object:
            atmosphereHeight += (8.31 * i.temperature * ((0.7 * i.correctedRadius**2) / (4e6 * self.gamePos.distance_to(i.gamePos))***(1/4))) / (0.029e6 * self.calcSurfaceGrav())

    return atmosphereHeight
```

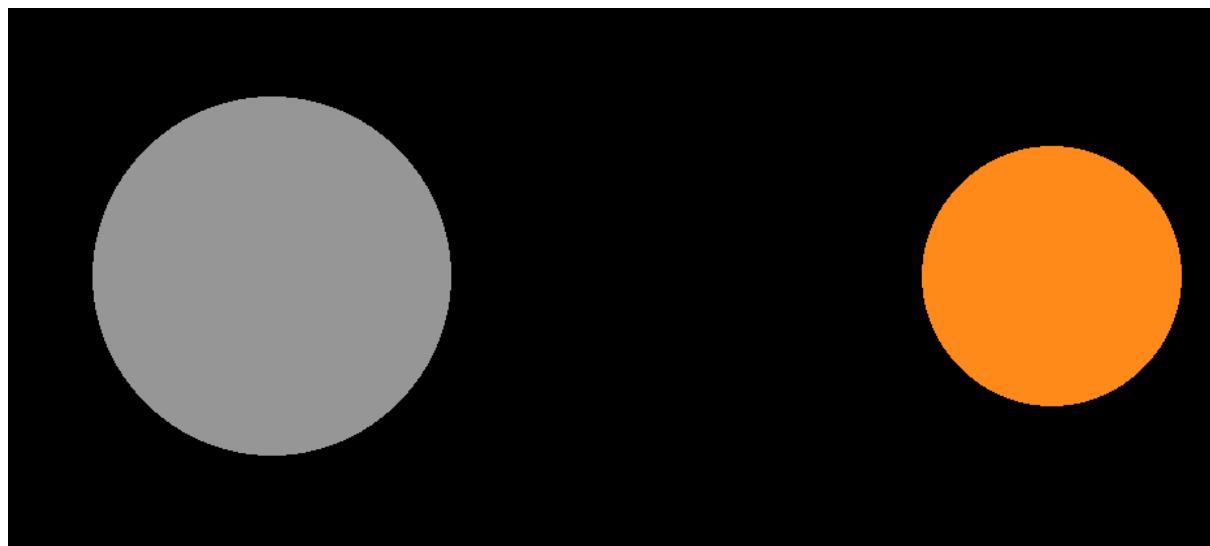
To produce the graphic for the atmosphere, I draw another circle on the screen at the same position as the main circle in the planet `update()` method with the radius being the normal radius of the object plus the result of this method.

```
if self.__class__.__name__ == "Planet": #if planet object:
    pygame.draw.circle(screen, (self.colour[0], self.colour[1], self.colour[2], 50), self.screenPos, (self.radius + self.getAtmosphereHeight(physicsSprites)) * physEng.zoomScale) #draw atmosphere
    print(self.getAtmosphereHeight(physicsSprites))

    pygame.draw.circle(screen, self.colour, self.screenPos, self.radius * physEng.zoomScale) #draw planet

    self.rect = self.image.get_rect(center=self.screenPos) #update rect, centred at the planet's position
```

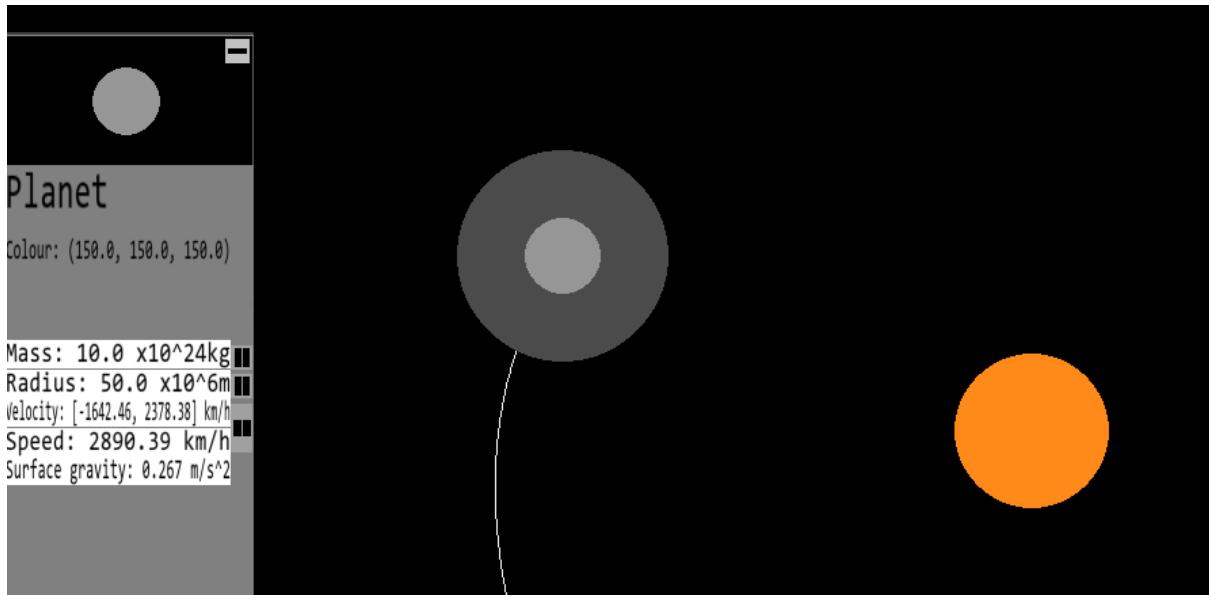
Running to test with the two-body system, the atmosphere was drawn at the correct height but was the full colour with no transparency.



After looking through the Pygame documentation, it looked like the alpha value in the colour to be drawn had to be enclosed in square brackets. After doing this, however, I was met with an error for an invalid colour argument. My next solution was to create a new pygame.color object from the planet's colour which I could then change the alpha of before passing through as a parameter for the draw() method.

```
alphaColour = pygame.Color(int(self.colour[0]), int(self.colour[1]), int(self.colour[2]), a=50)  
pygame.draw.circle(screen, (alphaColour), self.screenPos, (self.radius + self.getAtmosphereHeight(physEng.sprites)) * physEng.zoomScale) #draw atmosphere
```

This unfortunately still did not change the output of the program. My other option for representing the atmosphere was just darkening the colour of the planet by some factor before drawing it as the atmosphere. Instead of using an alpha value, I halved the colour values passed through in order to darken them, which led to this result:



This is far from ideal, as it covers up the orbit lines and does not look very good. I tried just drawing the boundary of the atmosphere by adding the optional line width parameter to the draw() method, and I also halved the colour values again to bring the total decrease to a quarter:



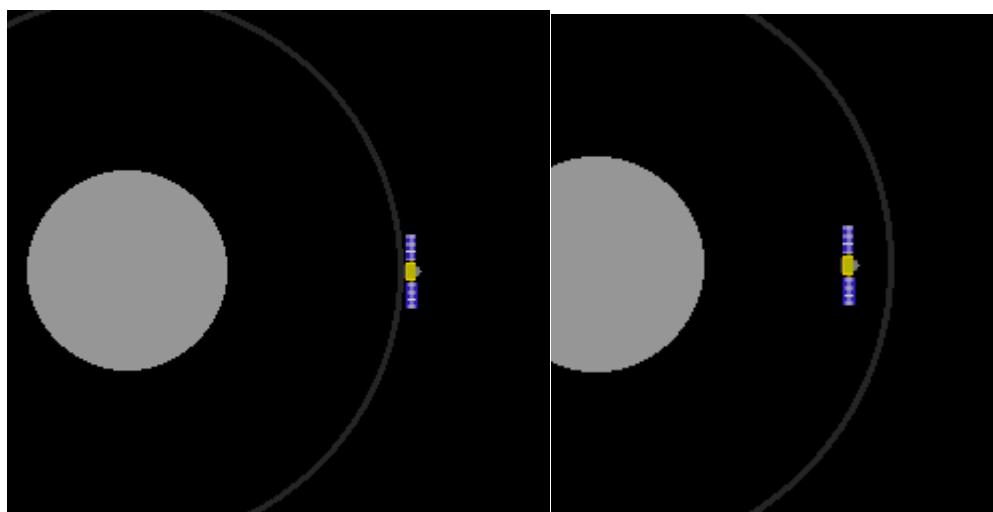
This is better, and does not interfere with the orbit lines. As the atmosphere should only affect the spacecraft, I will only allow the atmosphere to be drawn when the spacecraft controls button is active. In addition to this, the atmospheric height will be displayed in the info panel, but will not be able to be changed directly by the user through it.

To make the spacecraft slow down while in this region, I will run a check to see whether the rect of the spacecraft intersects with the rect of the atmosphere circle, and if so slow the spacecraft's speed. The issue with this is that the circle and spacecraft rects are both rectangular, meaning that if the spacecraft is tilted or intersects with the some areas just outside the circle, it will experience slowdown even when it is not meant to. The other option for these checks is to check the distance of the spacecraft from each object against the height of the atmosphere for each object, which would be a lot more accurate but in return could be slightly more inefficient.

I decided to program the latter option due to its accuracy. In the main running loop, if the spacecraft object is active, the check will be made:

```
#check for atmospheric slowdown
for i in physicsSprites: #for each physics object:
    if i.__class__.__name__ == "Planet": #only planets have atmospheres
        if spacecraft.gamePos.distance_to(i.gamePos) * 1e6 < i.getAtmosphereHeight(physicsSprites) + i.correctedRadius:
            print("flag") #debug
```

I first wrote this code above to test if the check was correctly applied.



Testing at the boundary on both sides, the flag was only printed when the spacecraft was within the circle, which showed that it worked as intended.

To apply the actual slowdown to the speed, I will simply multiply the speed by some decrease factor representing drag. This will be constant for all planets, since I am using the same atmospheric density in each of their height calculations already. To account for the time scale, I will also raise this slowed value by the power of the time scale as well. This code to change the speed will be a modified version of the same used in the speed textbox `textInput()` method.

Orbital Mechanics Simulator – Sam Prince

Centre number 64859 | Candidate number 6103

```
#check for atmospheric slowdown
for i in physicsSprites: #for each physics object:
    if i._class__name__ == "planet": #only planets have atmospheres
        if spacecraft.gamePos.distance_to(i.gamePos) < i.getAtmosphereHeight(physicsSprites) + i.radius: #if spacecraft is within atmosphere

            if spacecraft.velocity[0] == 0: #check for 0 in horizontal component to prevent div by 0 error
                if spacecraft.velocity[1] < 0: #if negative vertical component:
                    spacecraft.velocity[1] = -spacecraft.velocity[0] * 0.95*(physEng.getTimeScale()) #speed change affects only vertical velocity (change sign)
                else:
                    spacecraft.velocity[1] = spacecraft.velocity[0] * 0.95*(physEng.getTimeScale()) #speed change affects only vertical velocity

            else:
                magnitude = spacecraft.velocity.magnitude() + spacecraft.velocity.magnitude() - spacecraft.velocity.magnitude() * 0.95*(physEng.getTimeScale())
                angle = math.atan(spacecraft.velocity[1]/spacecraft.velocity[0])

                if spacecraft.velocity[0] < 0: #if negative horizontal component:
                    spacecraft.velocity[0] = - (magnitude * math.cos(angle)) / 1000 #result should be negative
                else:
                    spacecraft.velocity[0] = (magnitude * math.cos(angle)) / 1000

                if spacecraft.velocity[1] < 0: #if negative vertical components:
                    spacecraft.velocity[1] = - (magnitude * math.sin(angle)) / 1000 #result should be negative
                else:
                    spacecraft.velocity[1] = (magnitude * math.sin(angle)) / 1000
```

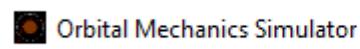
Running to test, however, the speed dropped to 0 and the velocity became a very small number when the spacecraft entered the atmosphere. It seems that the multiplier value of 0.95 is much too high a decrease in speed. After changing this value to 0.99, it still decreased to 0 almost immediately. Even changing the value to 0.99999, it still decreased to in almost the same time. I then changed it back to 0.95 and removed the division by 1000 in each of the velocity changes, as the units should be consistent with the velocity anyway. This resulted in the desired slowdown effect when the spacecraft passed through the test atmosphere.

At this point, I was slightly running out of time, and so I decided to clean up the rest of the implementation and move on to testing and evaluation.

I then went through all of my code, moved some lines around inside classes and commented anything unclear to make it more readable and maintainable. I also decided to remove any debug lines to make the file smaller for uploading to itch.io.

As the final touch, I changed the icon on the Pygame window to be the logo for my project.

```
pygame.display.set_caption('Orbital Mechanics Simulator') #names window
pygame.display.set_icon(pygame.image.load("icon.png")) #sets window icon
```



Testing + Bug fixing

Test #	Test description	Expected outcome	Actual outcome	Pass/fail
1a	Atmospheric height: Atmosphere not drawn with no star object in simulation.	getAtmosphereHeight() method not called.	As expected.	Pass
1b	Dual body system (1 star), planet placed at same position as star (boundary).	Height of atmosphere calculated as a large value.	Error: float division by 0	Fail, fixed

```
File "e:\NEA (USB)\proj\main.py", line 1329, in main
    physicsObjects.update(physEng, physicsSprites, buildMenuGroup, buttonGroup) #apply physics calculation results to objects + draw to screen
    ~~~~~
File "e:\NEA (USB)\proj\main.py", line 750, in getAtmosphereHeight
    atmosphereHeight += (8.31 * i.temperature * ((0.7 * i.correctedRadius**2) / (4e6 * self.gamePos.distance_to(i.gamePos))))*(1/4) / (0.029e6 * self.calcSurfaceGrav()) #calculate atmospheric height effect and add to variable
ZeroDivisionError: float division by zero
PS E:\NEA (USB)\proj>
```

This scenario should be near impossible to replicate outside of directly editing the scenario files as I have done to produce this situation. Nevertheless, as with the radius in the force calculations back in

stage 2, I will not calculate the additional atmospheric height due to that star if the distance between the planet and star is 0.

```
if self.gamePos.distance_to(i.gamePos) != 0: #if distance to star is not 0 (prevents div by 0 error):
    atmosphereHeight += (8.31 * i.temperature * ((0.7 * i.correctedRadius**2) / (4e6 * self.gamePos.dis
```

Running once more to test, the planet in the dual-body system I had set up had no atmospheric height, as expected.

Test #	Test description	Expected outcome	Actual outcome	Pass/fail
1c	Star temperature 0 (boundary).	Height of atmosphere 0.	As expected.	Pass
1d	Star temperature 2000 (normal). Simulation unpause, planet moves towards star.	As planet moves towards star, height of atmosphere increases.	As expected.	Pass
1e	Three-body system (2 stars), planet placed equidistant between stars.	Heights of atmosphere due to each star is added together to produce the correct value.	As expected.	Pass
2a	Atmospheric effects: Spacecraft experiences slowdown when passing through atmosphere.	Spacecraft speed slows once per tick when passing through atmosphere. Velocity decreases proportionally.	As expected.	Pass
2b	Spacecraft passes through atmosphere with speed/velocity locked.	No change in speed/velocity.	Spacecraft speed/velocity changes as before.	Fail, fixed

I seem to have forgotten to add a check in for the spacecraft's velocity freeze flag when applying the effects of the atmospheric drag. This is a one line fix in the main running loop:

```
for i in spacecraftControlGroup:
    i.mouseHoldUpdate(physEng, physObjects, spacecraft, buttonGroup, buildMenuGroup) #update position

if not spacecraft.velocityFreezeFlag: #if spacecraft velocity/speed is not frozen:

    #check for atmospheric slowdown
    for i in physicsSprites: #for each physics object:
        if i.__class__.__name__ == "Planet": #only planets have atmospheres
```

Running again, this now works as intended.

Test #	Test description	Expected outcome	Actual outcome	Pass/fail
3a	Collision physics: dual body system, 0 initial speed.	Objects bounce off of one another elastically and oscillate between midpoint and starting position.	Feature not implemented.	N/A
3b	Both objects placed at same position.	Objects do not move.	Feature not implemented.	N/A

3c	Both objects at same position, one with forwards velocity.	As object moves to the right, the other bounces and gains velocity proportional to momentum. Other object loses velocity proportional to momentum.	Feature not implemented.	N/A
3d	Spacecraft and object collision. No initial velocities.	Objects bounce off of one another elastically and oscillate between midpoint and starting position.	Feature not implemented.	N/A
4a	Stars as background: load program and move camera around.	'stars' are shown, and move with camera.	Feature not implemented.	N/A
5a	Planet textures: load program with 6 planet system.	Planets load with the supplied texture applied to them. Image is converted correctly by Pygame convert() method, and does not affect performance in the Pygbag test environment.	Feature not implemented.	N/A

At the end of testing, I re-ran all tests from this section in order to ensure that none of my bugfixing had made any other undesired changes.

I then re-uploaded the project to itch.io to update my changes there as well.

Stakeholder Feedback

In terms of how the atmosphere was being displayed, Oli said that it would have been more obvious to the user if the atmosphere was displayed as I had first attempted to implement, with the semi-transparent circle. The new representation could be misleading to users, as they could be mistaken for orbit lines. However, the fact that the colour of these atmosphere outlines were the same as the parent object was beneficial as it differentiated them from the white colour of the actual orbit lines.

The stakeholders also found it useful for the atmosphere outlines to only be displayed when the spacecraft was active, as it made them only show when relevant, and the screen felt less cluttered in normal use as a result. One improvement that was suggested was to display the height of the atmosphere in kilometres on the info panel for planets, in the same way as surface temperature is displayed for stars.

Since there were going to be no more development stages, I made the requested changes at this point. Luckily, this was as simple as copying the code for the spacecraft surface temperature font and instead making it call the selected object's `getAtmosphereHeight()` method.

```
if selectedObject.__class__.name == "Star": #if selected object is a star
    textBuffer.append(font.render(f"Surface temperature: {surfaceTempTextbox.text}K", True, (0,0,0))) #render font for surface temperature
    surfaceTempTextbox.update() #update star surface temperature textbox
    pygame.draw.rect(screen, surfaceTempTextbox.colour, surfaceTempTextbox.rect) #draw textbox for star surface temperature

elif selectedObject.__class__.name == "Planet": #if selected object is a planet:
    textBuffer.append(font.render(f"Atmospheric height: {round(selectedObject.getAtmosphereHeight(physicsSprites), 2)} km", True, (0,0,0)))
    textboxObjectGroup.update() #set values in textboxes to that of the selected object
```

I decided not to make this value editable from a textbox, as I would have to create a new group and work out all the logic for changing each dependent variable, which I have run out of time for.

Otherwise, the stakeholders weren't really that fussed with the collision physics remaining unimplemented; in fact, Mr Dougals-Scott said that with no collisions, he would be able to demonstrate the concept of gas giants when objects moved through one another. They also did not mind about the background stars and planet textures, as they felt that this was not the main focus of the program.

Evaluation

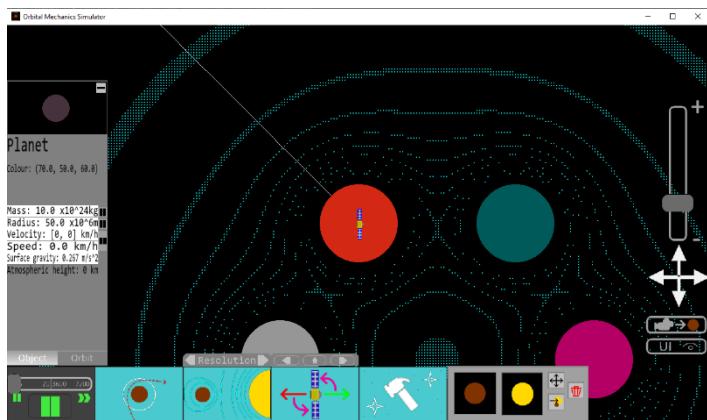
Post-dev alpha testing

Test #	Requirement(s)	Test description	Expected outcome	Pass/fail
1	1, 7E	Run program and resize window to screen.	Options menu should show on top of background, scaled. Window is correctly scaled to screen.	Pass

For this test, I ran the program both locally and through itch.io. The itch.io program scaled correctly when in fullscreen mode, but the aspect ratio would not change (due to the way that itch.io functions in the browser). Locally, I was able to resize the window as much as I liked, and the UI scaled correctly to match the size of the window, even when stretched to an extreme level.



Test #	Requirement(s)	Test description	Expected outcome	Pass/fail
2	1	Interact with buttons on the options menu.	Menu buttons should open the corresponding menu/change the program accordingly.	Pass



Testing each of the buttons on the screen at program start in turn, they all produce the desired effect. Repeating the test on itch.io, the same results were obtained.

Test #	Requirement(s)	Test description	Expected outcome	Pass/fail
3	1, 7A	Use interactive whiteboard to interact with buttons in options menu.	The buttons can be interacted with through the whiteboard's touch controls.	Pass

The buttons could all be selected through the whiteboard both when run locally and through itch.io. The only slight issue was that a student may have some difficulty reaching the top of the zoom slider if the whiteboard was mounted slightly higher, but the rest of the more frequently used controls are easily reachable from a low-reaching position, even if sitting in a chair next to the screen as a wheelchair user might; as such, I consider this aspect a pass.

Test #	Requirement(s)	Test description	Expected outcome	Pass/fail
4	5B	Draw planet at game start.	Planet should be drawn to screen along with the options menu and background.	Pass

Test was completed as expected, both locally and on itch.io.

Test #	Requirement(s)	Test description	Expected outcome	Pass/fail
5	5B	Draw planet while game is running.	Planet should be drawn to screen in front of background.	Pass

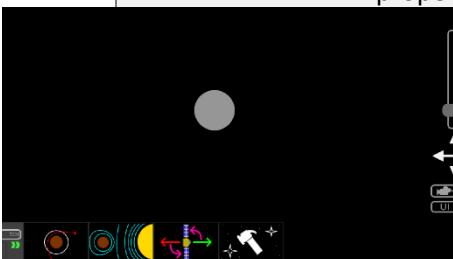
Unpausing the simulation, the planets were drawn as they moved. Using the build menu to place a planet and then a star, they were drawn correctly. The same results were obtained locally and on itch.io.

Test #	Requirement(s)	Test description	Expected outcome	Pass/fail
6	5, 5B	Draw second planet in different location while game is running.	Second planet should also be drawn to screen.	Pass

The test was completed both locally and through itch.io as expected.

Test #	Requirement(s)	Test description	Expected outcome	Pass/fail
7	5, 5B	Draw second planet at game start.	Second planet should also be drawn to screen.	Pass

Test was completed both locally and through itch.io as expected.

<i>Test #</i>	<i>Requirement(s)</i>	<i>Test description</i>	<i>Expected outcome</i>	<i>Pass/fail</i>
8	4	One planet object with physics properties.	No movement.	Pass
			Test was completed both locally and through itch.io as expected.	

<i>Test #</i>	<i>Requirement(s)</i>	<i>Test description</i>	<i>Expected outcome</i>	<i>Pass/fail</i>
9	4, 4B, 5	Two planet objects created on top of one another.	No movement.	Fail, fixed

Unpausing the simulation, it crashed due to a division by 0 error in the `update()` method of the physics engine.

```
File "e:\NEA (USB)\proj\main.py", line 1310, in main
    physEng.update(physObjects, screen) #run physics calculation step
    ~~~~~~
File "e:\NEA (USB)\proj\engine.py", line 36, in update
    i.velocity += Vector2((sprites[j].gamePos[0]-i.gamePos[0])/math.sqrt((sprites[j].gamePos[0]-i.gamePos[0])**2+(sprites[j].gamePos[1]-i.gamePos[1])**2),
BetweenObjects(sprites[j], i.gamePos.distance_to(sprites[j].gamePos) * 1e6) * self.getTimeScale() #calculate gravitational force
    ~~~~~~
ZeroDivisionError: float division by zero
PS E:\NEA (USB)\proj>
```

To fix this, I expanded the check for avoiding calculations for the object with itself to instead be for any object with the same gamePos.

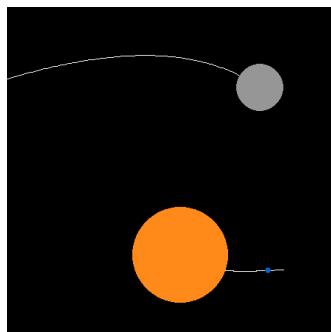
```
for i in sprites: #for each sprite in group:
    if not i.velocityFreezeFlag: #if velocity of object has not been frozen:
        j = 0
        while j < len(sprites): #for each other sprite:
            if not sprites[j].gamePos == i.gamePos: #skip over calculation for itself or others in same position
                i.velocity += Vector2((sprites[j].gamePos[0]-i.gamePos[0])/math.sqrt((sprites[j].gamePos[0]-i.gamePos[0])**2+(sprites[j].gamePos[1]-i.gamePos[1])**2),
                BetweenObjects(sprites[j], i.gamePos.distance_to(sprites[j].gamePos) * 1e6) * self.getTimeScale() #calculate gravitational force
            j += 1
```

Running the test again, both objects remain stationary on top of one another, as expected. After reuploading the project to itch.io, it functioned as expected there as well.

<i>Test #</i>	<i>Requirement(s)</i>	<i>Test description</i>	<i>Expected outcome</i>	<i>Pass/fail</i>
10	4, 4B, 5	Create two planet objects with physics properties, identical masses.	Planets should exert an equal force on each other and move as such.	Pass

The test was completed both locally and through itch.io as expected.

<i>Test #</i>	<i>Requirement(s)</i>	<i>Test description</i>	<i>Expected outcome</i>	<i>Pass/fail</i>
11	4, 4B, 5	As above, but one planet has a greater mass.	Planet with lesser mass should experience a greater force of attraction and should move as such.	Pass



The test was completed both locally and through itch.io as expected.

<i>Test #</i>	<i>Requirement(s)</i>	<i>Test description</i>	<i>Expected outcome</i>	<i>Pass/fail</i>
12	1E, 4	Play simulation in dual planet system and change simulation speed from minimum to maximum and vice versa.	Physics remains consistent. Planets slow down when simulation slows, and speed up when simulation sped up.	Pass

The test was completed both locally and through itch.io as expected.

<i>Test #</i>	<i>Requirement(s)</i>	<i>Test description</i>	<i>Expected outcome</i>	<i>Pass/fail</i>
13	8	Camera movement by pressing/moving mouse in area.	Effect of the screen moving when controls pressed corresponding to the 4 cardinal directions.	Pass

The test was completed both locally and through itch.io as expected, and the camera controls also functioned correctly when the arrow keys were pressed.

<i>Test #</i>	<i>Requirement(s)</i>	<i>Test description</i>	<i>Expected outcome</i>	<i>Pass/fail</i>
14	8, 8A	Camera zooms in and out (no focus on objects).	Objects move and grow/shrink in size accordingly with physics consistent.	Pass

The test was completed both locally and through itch.io as expected. I also moved the camera around and continued to change the zoom value, which still worked as intended.

<i>Test #</i>	<i>Requirement(s)</i>	<i>Test description</i>	<i>Expected outcome</i>	<i>Pass/fail</i>
15	8, 8B	Cycle camera focus through different objects.	Camera moves between each and every object in the scenario with physics consistent.	Pass

The test was completed both locally and through itch.io as expected, with no objects missed out of the loop even if added by the build menu or if it was the spacecraft. The info panel opened correctly whenever an object was focused on.

Test #	Requirement(s)	Test description	Expected outcome	Pass/fail
16	8, 8A, 8B	Camera zooms in and out (focus on objects).	Objects move and grow/shrink in size accordingly with physics consistent.	Partial pass

When the scenario is paused, this feature works as intended. However, when unpause and zoomed in past ~3x scale, the camera movement shakes and quickly flings the camera out of the scenario area. When the camera is subsequently zoomed out, it returns to the object in focus. This is likely due to the fact that the movement offset is set before the physics calculations are made in the physics engine update() method. I attempted to fix this by moving the set to after the motion of the objects has been calculated:

```
if not self.isPaused: #physics calculations: do not run when simulation is paused
    for i in sprites: #for each sprite in group:
        if not i.velocityFreezeFlag: #if velocity of object has not been frozen:
            j = 0
            while j < len(sprites): #for each other sprite:
                if not sprites[j].gamePos == i.gamePos: #skip over calculation for itself or others in same position
                    i.velocity += Vector2((sprites[j].gamePos[0]-i.gamePos[0])/math.sqrt((sprites[j].gamePos[0]-i.gamePos[0])**2+(sprites[j].gamePos[1]-i.gamePos[1])**2))
                j += 1
    if self.focusedObject != 0: #if there is an object to be focused on:
        self.movementOffset += Vector2(screen.get_width()/2, screen.get_height()/2) - self.focusedObject.screenPos #change movement offset to that of focused object
```

This, however, seemed to make no difference at all, likely due to the fact that the screenPos of the objects was only changed in the respective object's update() method, called after the physics calculation step was completed. My final try at fixing this was to move the set into the main running loop, referencing the physics engine, just after the movement of the objects was updated.

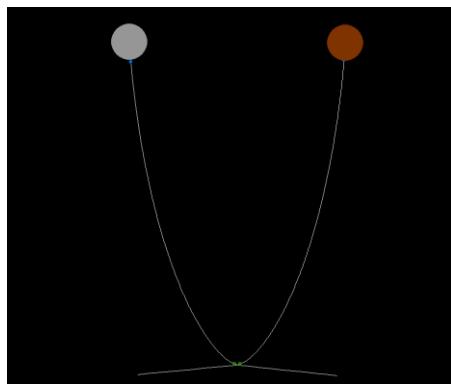
```
physObjects.update(physEng, physicsSprites, buildMenuGroup, buttonGroup) #apply physics calculation results to objects + draw to screen
if physEng.focusedObject != 0: #if there is an object to be focused on:
    physEng.movementOffset += pygame.Vector2(screen.get_width()/2, screen.get_height()/2) - physEng.focusedObject.screenPos #change movement offset to that of focused object
```

This again produced no difference in the result, so I moved the code back into the physics engine update() method. The zoom function does work, but only if the simulation is paused or if not zoomed in too far. At no point does it cause a crash in the program, so I am marking this test as a partial pass.

Test #	Requirement(s)	Test description	Expected outcome	Pass/fail
17	3	Draw orbit lines for single planet system with a sideways velocity.	Orbit line drawn straight in direction of travel.	Pass

The test was completed both locally and through itch.io as expected.

Test #	Requirement(s)	Test description	Expected outcome	Pass/fail
18	3, 4, 4B	Draw orbit lines for a dual planet system with both planets having equal upwards velocity.	Orbit lines bend to intersection point before correctly predicting collision/movement apart.	Pass

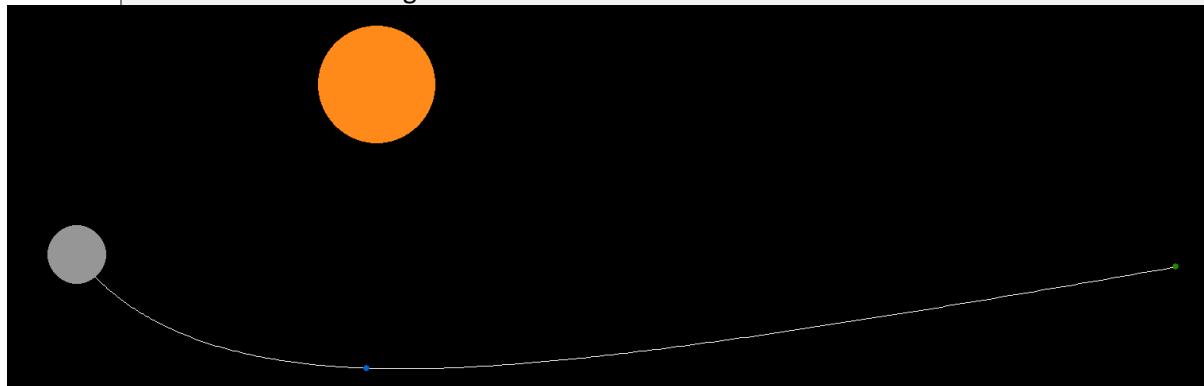


The test was completed both locally (shown on the right) and through itch.io as expected. When the simulation was unpause, the objects followed the path of the orbit lines before moving through each other with a large velocity at the intersection point, as predicted.

<i>Test #</i>	<i>Requirement(s)</i>	<i>Test description</i>	<i>Expected outcome</i>	<i>Pass/fail</i>
19	3, 3B	Draw orbit lines for a dual planet system with both planets having equal upwards velocity.	Apoapsis markers are present at point of intersection.	Pass

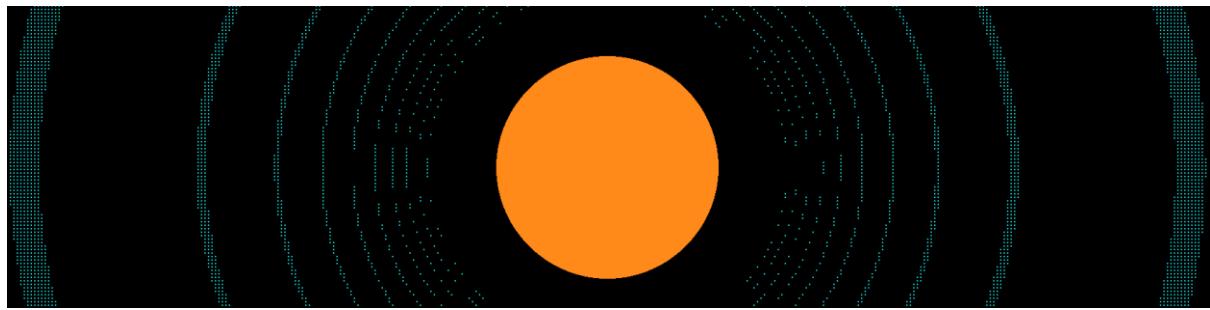
As seen in the test above, the green apoapsis markers are drawn correctly at the intersection point. This was also true in the itch.io version.

<i>Test #</i>	<i>Requirement(s)</i>	<i>Test description</i>	<i>Expected outcome</i>	<i>Pass/fail</i>
20	3, 3B	Draw orbit lines for dual planet system, one has velocity while other is stationary and has a higher mass.	Apoapsis and periapsis markers drawn at correct points along line.	Pass



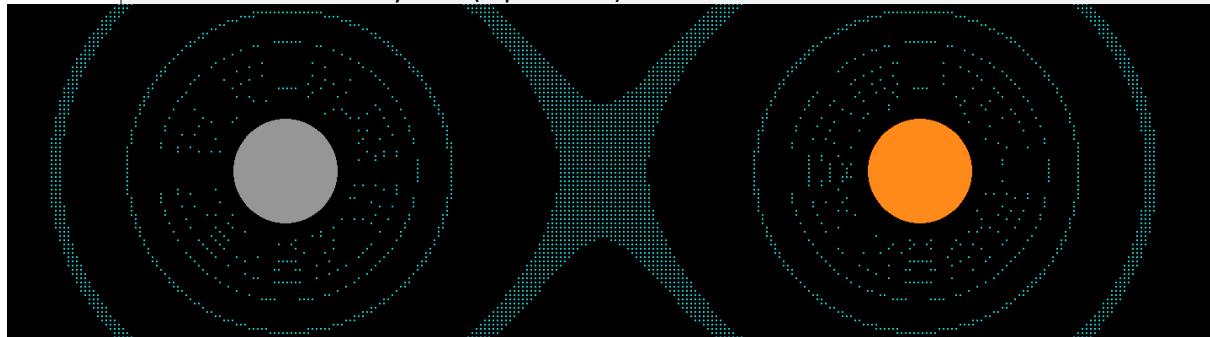
The test was completed both locally and through itch.io as expected, with the orange star having its velocity frozen to make it stationary throughout the test.

<i>Test #</i>	<i>Requirement(s)</i>	<i>Test description</i>	<i>Expected outcome</i>	<i>Pass/fail</i>
21	3A	Draw equipotential lines for single planet system.	Concentric circles drawn around planet.	Pass



The equipotential lines were drawn correctly (with some performance loss) in the local program, and in itch.io.

<i>Test #</i>	<i>Requirement(s)</i>	<i>Test description</i>	<i>Expected outcome</i>	<i>Pass/fail</i>
22	3A, 4B	Draw equipotential lines for dual planet system (equal mass).	Circular pattern with warped region in between planets.	Pass



The test was completed both locally and through itch.io as expected.

<i>Test #</i>	<i>Requirement(s)</i>	<i>Test description</i>	<i>Expected outcome</i>	<i>Pass/fail</i>
23	3A, 4B	Draw equipotential lines for dual planet system (different masses).	Equipotential lines more grouped/larger diameter of circular pattern around planet of larger mass.	Pass



The test was completed both locally and through itch.io as expected.

<i>Test #</i>	<i>Requirement(s)</i>	<i>Test description</i>	<i>Expected outcome</i>	<i>Pass/fail</i>
24	1F	Show information panel for each object in turn in a tri-planet system.	Correct information given for each planet.	Pass

The test was completed both locally and through itch.io as expected, with the correct information and freeze flag states displayed for each object.

<i>Test #</i>	<i>Requirement(s)</i>	<i>Test description</i>	<i>Expected outcome</i>	<i>Pass/fail</i>
25	4B, 4D	Dual planet system: lock all parameters for one planet, then play simulation.	No parameters change for locked planet, but do for unlocked planet. When switching panel between each planet, locked variables remain consistent.	Pass

The test was completed both locally and through itch.io as expected.

<i>Test #</i>	<i>Requirement(s)</i>	<i>Test description</i>	<i>Expected outcome</i>	<i>Pass/fail</i>
26	1A, 1F, 4, 4B	Dual planet system: increase/decrease mass of one planet while simulation is running.	Force of attraction increases/decreases between the two objects. Change is reflected in the info panel for the object.	Pass

The test was completed both locally and through itch.io as expected. Using preset 2, I increased the mass of the star object to 1000, which then made the grey planet move around in a tighter curve.

<i>Test #</i>	<i>Requirement(s)</i>	<i>Test description</i>	<i>Expected outcome</i>	<i>Pass/fail</i>
27	1B, 1F, 4, 4B	Single planet system: increase/decrease speed of planet while simulation is running.	Speed of object increases/decreases. Change is reflected in the info panel for the object. Velocity increases/decreases by correct amount in both component directions.	Pass

The test was completed both locally and through itch.io as expected.

<i>Test #</i>	<i>Requirement(s)</i>	<i>Test description</i>	<i>Expected outcome</i>	<i>Pass/fail</i>
28	1C, 1F, 4, 4B	Single planet system: increase/decrease velocity of planet in one component direction while simulation is running.	Speed of object increases/decreases. Change is reflected in the info panel for the object. Velocity increases/decreases by correct amount in affected component direction, while remaining constant in the other.	Pass

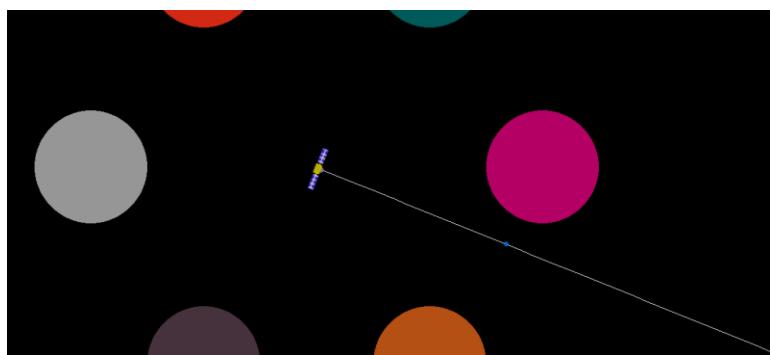
The test was completed both locally and through itch.io as expected.

<i>Test #</i>	<i>Requirement(s)</i>	<i>Test description</i>	<i>Expected outcome</i>	<i>Pass/fail</i>
29	1D, 1F, 4, 4B	Dual planet system: increase/decrease time period of one planet while simulation is running.	Orbital radius changes accordingly. Orbital velocity changes accordingly.	Pass

The test was completed both locally and through itch.io as expected. When the orbital radius of the grey planet in scenario 2 was decreased, it moved closer to the star object, with a higher orbital velocity.

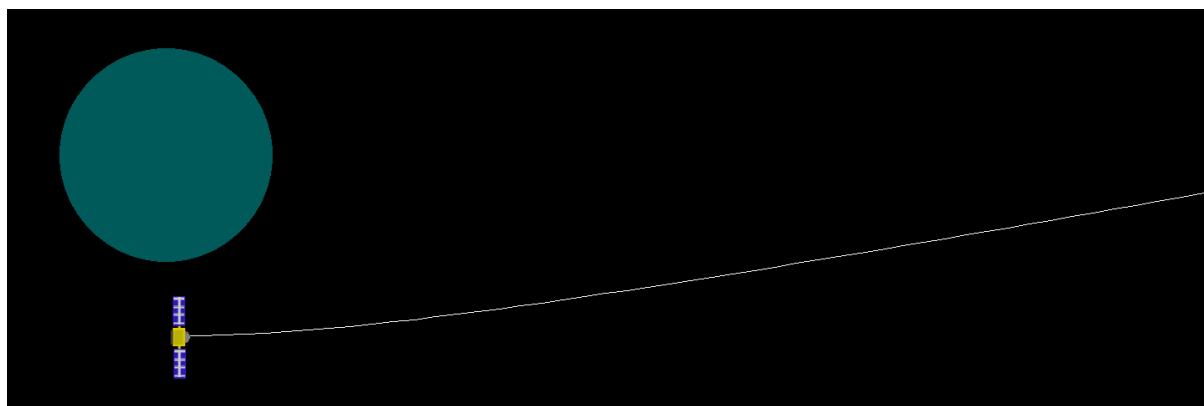
<i>Test #</i>	<i>Requirement(s)</i>	<i>Test description</i>	<i>Expected outcome</i>	<i>Pass/fail</i>
30	1, 2	Click spacecraft control show button, and rotate the spacecraft in both directions. Move spacecraft forwards.	Spacecraft control options show when corresponding button is clicked. Spacecraft moves in correct directions, and accelerates smoothly when respective button pressed.	Pass

The test was completed both locally and through itch.io as expected.



<i>Test #</i>	<i>Requirement(s)</i>	<i>Test description</i>	<i>Expected outcome</i>	<i>Pass/fail</i>
31	2, 4	Single planet system: spacecraft is given horizontal velocity below the planet.	Spacecraft experiences force upwards and moves as such.	Pass

The test was completed both locally and through itch.io as expected.



<i>Test #</i>	<i>Requirement(s)</i>	<i>Test description</i>	<i>Expected outcome</i>	<i>Pass/fail</i>
32	2, 4, 4A	Planet and star system: spacecraft moves into atmospheric region of planet.	Spacecraft experiences a frictional force inside the atmosphere, and slows down.	Pass

The test was completed both locally and through itch.io as expected.

<i>Test #</i>	<i>Requirement(s)</i>	<i>Test description</i>	<i>Expected outcome</i>	<i>Pass/fail</i>
33	2, 4C	Single planet system: spacecraft impacts planet.	Spacecraft bounces off surface of planet elastically.	N/A

34	4, 4B, 4C	Dual planet system: equal masses and radii.	Planets move towards each other and gain momentum before colliding elastically and moving away from one another while slowing down.	N/A
----	-----------	---	---	-----

Requirement 4C (collision between objects) was not implemented in the final version, so these tests are not applicable.

Test #	Requirement(s)	Test description	Expected outcome	Pass/fail
35	1, 5A	Load each preset scenario in turn.	Each scenario is loaded correctly, with simulation paused.	Pass

The test was completed both locally and through itch.io as expected. The scenarios would also reset correctly if loaded after changes were made inside the program.

Test #	Requirement(s)	Test description	Expected outcome	Pass/fail
36	1, 5C	Save custom scenario.	Scenario is saved in the correct location.	N/A
37	1, 5C	Load custom scenario just saved.	Scenario is loaded correctly.	N/A
38	1, 5C	Load custom scenario after closing and restarting program.	Scenario is loaded correctly.	N/A
39	6, 6A, 8	Start program, and move camera around.	Stars are represented on background, and move consistently when camera is moved.	N/A
40	6, 6B	Start program with any number of planets.	Textures for planets display correctly.	N/A

Requirements 5C (save/load custom scenarios), 6A (stars as background) and 6B (textures for planets) were not implemented in the final version, so these tests are not applicable.

Test #	Requirement(s)	Test description	Expected outcome	Pass/fail
41	7B	Run program on lowest spec device that I have access to.	Program runs with acceptable performance (above 20FPS).	Pass

<`Clock(fps=59.52)`>
<`Clock(fps=59.52)`>
<`Clock(fps=59.52)`>
<`Clock(fps=4.00)`>
<`Clock(fps=4.00)`>

The test was completed locally, and ran with an FPS of 59 in most situations except for when equipotential lines were being drawn on the screen, during which the FPS dropped to 4, increasing to 12 if set to the lowest equipotential resolution. This, however, is a known issue with how the equipotential lines are drawn, and they are optimised to the best of my ability. There is a warning within the program for increased resolutions of the lines, and so I will consider this a passed test with respect to the rest of the program.

Test #	Requirement(s)	Test description	Expected outcome	Pass/fail
42	7C	Run program from a website via a web browser.	Program runs from website with acceptable performance (above 20FPS).	Pass

In order to be able to display the FPS on the screen (as I had no access to the console in the browser), I added a line of code at the end of the main running loop to render font for the FPS directly onto the screen, just for the purpose of testing.

```
screen.blit(font.render(str(clock), True, (255,255,255)), (0,0)) #DEBUG: fps counter for itch.io performance testing  
pygame.display.update() #updates the screen|
```

I uploaded this version to itch.io and ran it to test.



I averaged 20 FPS both when paused and unpause, and when equipotential lines were drawn this dropped to 2-5 FPS. However, for normal use it is still within the acceptable region for performance, and so I consider this test a success. I removed the debug text and re-uploaded the original project.

Test #	Requirement(s)	Test description	Expected outcome	Pass/fail
43	7D	Run program using a desktop shortcut.	Program starts correctly.	N/A

Requirement 7D (desktop shortcut to run program) was not implemented in favour of accessing the simulation online, as it was more accessible to less tech-literate users. Therefore, this test is not applicable.

After completing these tests, I made the final re-upload of the program to itch.io.

Stakeholder beta testing/final feedback

For beta testing, I sent the link for the public itch.io page for the program and a zip file containing all the files for running it locally to my stakeholders, along with the post-dev test plan for reference.

Mr Douglas-Scott

Mr Douglas-Scott said that the buttons were intuitive to use after I gave him a quick explanation of the functions of the program. He found that the build menu buttons were the most difficult to instantly understand the meaning from the image, but he was able to learn these quickly. He also liked the range of preset scenarios available; in particular, he found that scenario 2 was the most useful to him as it represented a planet already in an orbit around a star.

In terms of the program functions, he thought that markers to show the previous path of an object as it has been moving would be ideal in lieu of more accurate orbit lines. He also said that instead of having to calculate or guess at the correct velocity to put an object into an orbit, it would be very useful to have either a button to automatically change the velocity of an object to place it into a stable orbit.

In terms of usability, he found it extremely convenient to be accessible on a browser page; he would not be comfortable with installing the software requirements on his own device as he doesn't consider himself as tech-literate, and so the itch.io version is the one that he would use. It was also easy and fast to load up the program with a scenario that he could use in a lesson. I asked him about the colourblindness requirement that he had brought up in the initial stakeholder questionnaire, and he said that someone colourblind should have little issue with the program.

I then went over the more qualitative success criteria with him to see if I had met the requirements in his opinion:

Requirement # and description	Success Criteria	Comments
1	Options menu	An easy to understand and navigable UI.
1F	Options to show labels above objects to display information about them	Labels on objects that give the user the basic information of it when selected.
4	Realistic physics	Objects in the game behave as in real life, calculations used to determine forces must be accurate.
6	Simplistic graphics	Graphics are simple enough to not be distracting, but detailed enough that at a glance it can be understood what each object represents.
7	Accessibility needs	Program is able to be easily navigated through simple (point and click) means. Program can be easily accessed and run.

Overall, Mr Douglas-Scott was very impressed with my solution, and mentioned that he may use it to teach his year 12 classes in the future. He also mentioned showing it to his year 11 GCSE classes as well.

Kai

Kai was able to navigate the entire program with ease without much introduction to it. Like Mr Douglas-Scott, he found the build menu buttons to be the least intuitive, but he could still understand their purpose. He also thought that the range of concepts that the three preset scenarios covered was good.

In terms of performance, he did not find that the lower FPS through running it on itch.io compared to locally made much of a noticeable difference, and it did not impact how usable the program was. However, he did note the slowdown when the equipotential lines were turned on, even at lower equipotential resolutions. He said that it was much easier to run it through a browser, as the school

had download restrictions on its devices, making it extremely difficult to install Python to run it locally, let alone the required libraries.

In terms of the readability of code, Kai said after skimming through the code for the program that he understood the gist of it; if he read into it for longer, he was sure that he could understand enough what each line did without too much effort.

I then went over the more qualitative success criteria with him to see if I had met the requirements in his opinion:

Requirement # and description		Success Criteria	Comments
1	Options menu	An easy to understand and navigable UI.	It is obvious what each button is meant represent.
1F	Options to show labels above objects to display information about them	Labels on objects that give the user the basic information of it when selected.	There is enough information; it covers everything we would learn about in the A level course, and a bit beyond in the case of the atmospheric height.
4	Realistic physics	Objects in the game behave as in real life, calculations used to determine forces must be accurate.	If you are using the correct equations, this must be correct – planets behave as they should in the program, so this looks good.
6	Simplistic graphics	Graphics are simple enough to not be distracting, but detailed enough that at a glance it can be understood what each object represents.	The graphics are simple enough, and it is easy to tell what is meant to be what.
7	Accessibility needs	Program is able to be easily navigated through simple (point and click) means. Program can be easily accessed and run.	Program can be accessed online easily enough.

Overall, Kai felt that the program would be useful to him in a classroom setting for teaching the targeted concepts.

Oli

Oli tried his best to create a scenario that would make the program crash, but was not able to with the software's available tools. Like the others, Oli found that running the program through itch.io would be much easier for the average user, although he himself ran it locally in the majority of his own use.

One thing that he wanted while using the program was the ability to zoom the program out to a greater extent; he found that if a planets gained much more velocity than another, it was difficult to follow both before one quickly went outside of the visible area, even when zoomed out to the greatest extent that the program allowed.

In a similar way to the other feedback, Oli thought that the menus were intuitive to navigate, and he could just try out the function of buttons that he did not know the function of to quickly work out their purpose.

I then went over the more qualitative success criteria with him in detail to see if I had met the requirements in his opinion:

Requirement # and description	Success Criteria	Comments
1 Options menu	An easy to understand and navigable UI.	The UI is easy to understand, but the buttons could look a bit better.
1F Options to show labels above objects to display information about them	Labels on objects that give the user the basic information of it when selected.	Seems like it has enough information on it, but entering data into the textboxes is a bit clunky.
4 Realistic physics	Objects in the game behave as in real life, calculations used to determine forces must be accurate.	Everything behaves like I would expect it to.
6 Simplistic graphics	Graphics are simple enough to not be distracting, but detailed enough that at a glance it can be understood what each object represents.	The graphics are probably the correct level for a classroom tool like this.
7 Accessibility needs	Program is able to be easily navigated through simple (point and click) means. Program can be easily accessed and run.	You can do pretty much whatever you want with the mouse, but changing the parameters of variables seems to be a big part of the software.

Overall, Oli felt that the program was robust and easily understood. Like with Kai, I asked Oli if the comments I had made on my code were sufficient for him to at least vaguely understand what was happening in terms of the processes being done, and he said that it was adequate.

Success criteria comparison

Note: All fully completed criteria are marked with a green box. Criteria that were marked as 'desired' and have not been completed are marked by a red evidence box. Incomplete criteria marked as 'extra' are marked by a blue evidence box. Any partially completed criteria are marked with a yellow box. Any criteria that have not been implemented in favour of other criteria are marked grey.

Requirement # and description		Desired feature or extra (time limitation)?	Success Criteria	Evidence / reason for failure	Completed?
1	Options menu	Desired	An easy to understand and navigable UI.	Post-Dev: Tests 1-3, 30, 35 Stakeholder beta testing - qualitative	Fully
1A	Options to manipulate mass of objects	Desired	Ability to change the mass of objects in-game, and the effects of the change reflecting on the physics.	Post-Dev: Test 26	Fully
1B	Options to manipulate speed of objects	Desired	Ability to change the speed of objects in-game, and the effects of the change reflecting on the physics.	Post-Dev: Test 27	Fully
1C	Options to manipulate direction of travel of objects	Desired	Ability to change the direction of travel of objects in-game, and the effects of the change reflecting on the physics.	Post-Dev: Test 28	Fully
1D	Options to manipulate time period of objects	Extra	Ability to change the time period of objects in-game, and the effects of the change reflecting on the physics.	Post-Dev: Test 29	Fully
1E	Option to control simulation speed	Extra	Speed of every object in the simulation can be sped up or stopped while keeping physics consistent.	Post-Dev: Test 12	Fully
1F	Options to show labels above objects to display information about them	Desired	Labels on objects that give the user the basic information of it when selected.	Post-Dev: Tests 24, 26-29	Fully

2	Controllable spacecraft/object	Desired	Object that the player can directly control via a separate menu.	Post-Dev: Tests 30-32	Fully
3	Orbit lines	Desired	Accurate lines that predict a reasonable distance ahead what the trajectory of an object will be.	Post-Dev: Tests 17-20	Fully
3A	Equipotential lines	Extra	Accurate equipotential lines between objects that can be toggled on or off.	Post-Dev: Tests 21-23	Fully
3B	Apoapsis and periapsis markers	Extra	Accurate apoapsis/periapsis markers displayed at correct points on an orbit line.	Post-Dev: Tests 19-20	Fully
4	Realistic physics	Desired	Objects in the game behave as in real life, calculations used to determine forces must be accurate.	Post-Dev: Tests 8-12, 18, 26-29, 31-32	Fully
4A	Atmospheric height and effects	Extra	An atmosphere around planets is present, that slows down the spacecraft when passing through.	Post-Dev: Test 32	Fully
4B	Planets exert gravitational force on each other	Desired	Planets' gravitational force on each other is considered when determining movement.	Post-Dev: Tests 9-11, 18, 22-23, 25-29	Fully
4C	Collision physics	Extra	When two objects collide, they react instead of passing through one another.	Time constraint; low on priority list.	Incomplete
4D	Keep selected parameters from changing as the simulation runs	Desired	Ability to 'lock' variables of choice so that they do not change as the simulation runs.	Post-Dev: Test 25	Fully
5	Multiple planets/stars	Desired	Multiple planets and stars can be represented in the software at the same time.	Post-Dev: Tests 6-7, 9-11	Fully

5A	Preset scenarios	Desired	At least 3 preset scenarios are able to be loaded at some point in the program.	Post-Dev: Test 35 Stakeholder beta testing – Mr Douglas-Scott	Fully
5B	Create + destroy planets	Desired	Users can place at least two celestial bodies at variable locations. Users can destroy objects.	Post-Dev: Tests 4-7	Fully
5C	Save + load custom scenarios	Extra	Options to save and load at least 3 custom scenarios that have been created in the program.	Limitations with Pygbag/ itch.io – cannot download files locally	Incomplete
6	Simplistic graphics	Desired	Graphics are simple enough to not be distracting, but detailed enough that at a glance it can be understood what each object represents.	Stakeholder beta testing - qualitative	Fully
6A	Stars as background	Extra	Small dots are present in the background to represent stars and more effectively show that it is outer space.	Time constraint; low on priority list.	Incomplete
6B	Textures for planets	Extra	Planets have at least one texture other than solid colour.	Time constraint; low on priority list.	Incomplete
7	Accessibility needs	Desired	Program is able to be easily navigated through simple (point and click) means. Program can be easily accessed and run.	Stakeholder beta testing - qualitative	Fully
7A	Navigable using just the mouse	Desired	All features of the program are accessible via clicking or holding the mouse buttons.	Post-Dev: Test 3	Fully
7B	Able to be run on low-end computers	Desired	May be able to be satisfied by requirement 7C. Program performs well (>20fps) on devices at a low (~2GB RAM) spec.	Satisfied by requirement 7C. Post-Dev: Tests 41-42	Fully

7C	Can be accessed online	Extra	Program can be accessed by any computer with an internet connection.	https://orbitalmechanicssimulator.itch.io/simulation Post-Dev: Test 41 Stakeholder beta testing: All stakeholders	Fully
7D	Desktop shortcut to run program	Extra	Program can be run simply by clicking on a shortcut icon on desktop. Would be made redundant by requirement 7C.	Satisfied by requirement 7C.	Not implemented
7E	Program window scales to any device resolution	Desired	Program correctly sizes to all monitor sizes.	Post-Dev: Test 1	Fully
8	Camera movement	Desired	Camera is able to be moved in cardinal directions while physics remains consistent.	Post-Dev: Tests 13-16	Fully
8A	Camera zoom	Extra	Camera can be zoomed in and out, with physics remaining consistent.	Post-Dev: Tests 14, 16	Partial (unstable when focused and zoomed)
8B	Camera focus on specific objects	Extra	Camera tracks and follows an object in motion, keeping it in the centre of the screen.	Post-Dev: Tests 15-16	Fully

In further development, the partially met and unmet criteria could be addressed as follows:

- **4C: Collision physics**

In a similar way to how the spacecraft is detected as inside an atmosphere, the sum of the radii of each pair of objects in the scenario could be compared to the distance between them; if less than the distance, then the velocities of the objects would be reversed along the line of centres between the two objects.

- **5C: Save + load custom scenarios**

Due to the limitations of Pygbag and itch.io, I would likely have to code this entire project again from scratch in order to be able to implement this feature for online users.

I would need to learn JavaScript in order to request files from a user's device, which has interesting security implications. Instead of hosting the program on itch.io, I would have to instead host on a rented server, which may limit the amount of time that the software is available by the cost of that server.

- **6A: Stars as background**

I intended for these to be implemented as randomly placed dot objects that would fade in and out periodically. They would each have their own gamePos value that would allow them to move consistently with the camera. In order not to harm performance, I would most likely have to only generate the objects that are within the camera view, likely by taking the top left and bottom right of the screen to create a bounding box within which these objects would be created. They would then be removed as they moved outside of the view of the camera.

- **6B: Textures for planets**

In a similar way to how the image is loaded for the spacecraft, I would load one of a number of preset circle textures for each planet object. This may also further differentiate them from the star objects.

- **8A: Camera zoom**

I have found through testing that this bug is due to the calculations for the camera offset lagging behind the movement of the objects themselves, causing them to overshoot. To fix this, I would likely have to improve the speed of the calculation for the offset somehow, or change the calculation for it altogether.

Success of usability features

In terms of running the software locally on different devices, I could have definitely made it much easier to access by coding in a different language such as JavaScript that could be run on devices without the need for a translator and specific libraries to also be installed. The use of Pygbag to upload the project to itch.io has however been very beneficial in making this software accessible to as many users as possible, as no downloads or setup is required if the software is run through the website, as shown in my stakeholder beta testing.

The program is fully navigable by use of an interactive whiteboard (except for inputting values into the textboxes), and with the controls lower down on the board it is easy for most students to reach.

In the initial stakeholder interviews, Mr Douglas-Scott mentioned that colourblindness was a concern for usability. As shown in my stakeholder beta feedback for him, this feature has been a success.

Maintainability

The code I have written has all been commented to aid with future maintenance, and split into sections based on what the code represents (classes, individual methods, variable declarations, main running loop, etc).

While the use of two separate files to split the code may make it difficult to follow processes logically, it makes the final solution much more modular in nature and so easier to maintain or optimise. By using a designation system for the buttons, it is easy to create new images for the clicked and unclicked states, and by using match..case statements adding functionality to new buttons is as easy as adding a new case in both the rescale and effect methods. One area in which maintainability can be improved is the button groups; there are a lot of them that are updated in similar ways, only for the purpose of showing them at different times. Some groups even have only one object contained within them; if I were to redo this project or optimise it in the future, I would make more of an effort to cut down on the number of these groups.

The online aspect of this program relies on both pygbag/itch.io and pygame/pygbag compatibility – if itch.io stops supporting the file generated by Pygbag (which is unlikely as many other programs on

itch.io will use the same format), if Pygbag has an update that stops the functionality of some aspects of the program or if Pygame is updated to a form which Pygbag is not able to convert, it may become impossible to update the online version from its current form.

The program should be compatible with updates to Python as it will support backwards compatibility; it is again just a question of whether the libraries I am using will function correctly on any new versions.

Approaching limitations

Feature	Reason	Approach
Creation of custom spacecraft	Time/outside scope	This would require either another build menu or another entire screen view separate to the simulation, as with spacecraft simulator and kerbal space program. I would have to create multiple different parts and chain them together as sprites in the actual simulation to make up the spacecraft. As far as effects of using different parts, I would have to include values of thrust, mass etc for each part and use them to calculate acceleration and other things.
Use of 3D graphics	Time/complexity	I looked into the implementation of this and came to the conclusion that the easiest method of doing this, at least in Python, would be to use the Panda3D library. This would likely require most if not all of the program to be rewritten. Use of a game engine such as Unity would however be my first choice, as the 3D engine is already built into the software.
Music and sound design	Outside scope	Using the Pygame.mixer module and downloading free sound effects online, it would be relatively simple to add appropriate sound effects.
Factors such as atmospheric composition of planets	Outside scope	Many more variables such as percentages of certain greenhouse gases would need to be recorded. There are likely many different and complex calculations for the density of atmospheres with these factors that I would have to implement in some way.
Lighting effects from stars	Complexity/time	This would require another level of rendering; I would most likely have to case rays out from each star object and make the colour value of whatever planet object the rays hit lighter.
Availability on mobile devices	Outside scope/complexity	Since my program already runs through a browser, I went to itch.io and ran it through my phone – the program actually ran with good performance, and the only issues were with the small size of the buttons (especially in the case of the variable freeze buttons), and the fact that text could not be entered into the textboxes. To make this software more mobile-friendly, I would likely have to make it into a dedicated mobile app to allow for these better controls.

Final Code

main.py

```

1 #####IMPORTS#####
2
3 import pygame
4 import engine
5 import numpy
6 import math
7 import asyncio
8
9 #####CLASS DEFINITIONS#####
10
11 class Button(pygame.sprite.Sprite):
12
13     def __init__(self, designation, physEng): #init method
14
15         pygame.sprite.Sprite.__init__(self) #initialise Pygame sprite
16
17         self.designation = designation
18
19         self.imageUnclicked = pygame.image.load("UIbutton" + str(designation) + ".png").convert_alpha() #Load images from file based on designation
20         self.imageClicked = pygame.image.load("UIbutton" + str(designation) + "Clicked.png").convert_alpha()
21
22         self.isClicked = False #set initial state
23         self.imageUnscaled = self.imageUnclicked #set initial image
24
25         #set unique initial variables for buttons:
26         match designation:
27
28             case 0: #if pause button:
29                 self.isClicked = True #set initial pause state
30                 self.imageUnscaled = self.imageClicked #set initial image as clicked
31
32             case 5: #if camera focus button:
33                 self.objectNumber = -1 #initialise variable for object to focus on
34
35             case 18: #if build menu move button:
36                 self.pickedObject = 0 #set initial picked object (cleared)
37
38         self.collideRect = pygame.Rect(0,0,0,0) #initialise rect for collisions
39
40         self.rect = pygame.Rect(0, 0, screen.get_width(), screen.get_height()) #get initial rect
41
42         self.rescale(physEng) #initial rescale
43
44
45     def rescale(self, physEng):
46
47         screenWidth = screen.get_width() #cache screen width
48
49         screenHeight = screen.get_height() #cache screen height
50
51         match self.designation: #collision rect positioning
52
53             case 0: #pause button:
54                 self.image = pygame.transform.scale(self.imageUnscaled,(screenWidth*0.063, screenHeight*0.063)) #rescale to window size
55                 self.rect.update((screenWidth*0.0), screenHeight*0.03, screenWidth*0.063, screenHeight*0.063)
56                 self.collideRect.update((screenWidth/32), screenHeight*(33/36), screenWidth/16, screenHeight/9)
57
58             case 1 | 2 | 3 | 4: #main menu buttons:
59                 self.rect.update((screenWidth/8)* self.designation, screenHeight*(31/36), screenWidth/8, screenHeight*(5/36)) #update rectangular coordinates for mouse cursor detection: menu at base of screen
60
61             case 5: #camera focus button:
62                 self.image = pygame.transform.scale(self.imageUnscaled, (screenWidth, screenHeight))
63                 self.collideRect.update(screenWidth*0.0081, screenHeight*0.7333, screenWidth*0.0089, screenHeight*0.8422) #update rectangular coordinates for mouse cursor detection: camera focus
64
65             case 6 | 7: #north/south camera movement buttons:
66                 self.rect.update((screenWidth*0.944 * 0.879 * (self.designation + 1)), screenHeight*0.0225, screenWidth*0.08) #update rectangular coordinates for mouse cursor detection: north/south camera movement
67
68             case 8 | 9: #east/west camera movement buttons:
69                 self.rect.update((screenWidth - screenWidth * 0.0445 * (self.designation - 7)), screenHeight*0.0156, screenWidth*0.0425, screenHeight*0.0411) #update rectangular coordinates for mouse cursor detection: north/south camera movement
70
71             case 10: #equipotential resolution increase:
72                 self.image = pygame.transform.scale(self.imageUnscaled, (screenWidth*0.125, screenHeight*0.009))
73                 self.rect.update(screenWidth*0.25, screenHeight*0.028, screenWidth*0.125, screenHeight*0.009)
74                 self.collideRect.update(screenWidth*0.355, screenHeight*0.02, screenWidth*0.0125, screenHeight*0.0411) #update rectangular coordinates for equipotential resolution increase
75
76             case 11: #equipotential resolution decrease:
77                 self.rect.update((screenWidth*0.253, screenHeight*0.032, screenWidth*0.015, screenHeight*0.025)) #update rectangular coordinates for equipotential resolution decrease
78
79             case 12: #hide UI button:
80                 self.rect.update((screenWidth*0.9081, screenHeight*0.79, screenWidth*0.0089, screenHeight*0.0422)) #update rectangular coordinates for mouse cursor detection: hide UI
81
82             case 13: #spacecraft turn left button:
83                 self.image = pygame.transform.scale(self.imageUnscaled, (screenWidth*0.125, screenHeight*0.033))
84                 self.rect.update((screenWidth*0.375, screenHeight*0.028, screenWidth*0.125, screenHeight*0.033))
85                 self.collideRect.update((screenWidth*0.55, screenHeight*0.02, screenWidth*0.055, screenHeight*0.0411)) #update rectangular coordinates for spacecraft turn left
86
87             case 14: #spacecraft turn right button:
88                 self.rect.update((screenWidth*0.46, screenHeight*0.032, screenWidth*0.035, screenHeight*0.028)) #update rectangular coordinates for spacecraft turn right
89
90             case 15: #spacecraft accelerate button:
91                 self.rect.update((screenWidth*0.42, screenHeight*0.032, screenWidth*0.035, screenHeight*0.027)) #update rectangular coordinates for spacecraft accelerate
92
93

```

Orbital Mechanics Simulator – Sam Prince
Centre number 64859 | Candidate number 6103

Orbital Mechanics Simulator – Sam Prince
Centre number 64859 | Candidate number 6103

Orbital Mechanics Simulator – Sam Prince

Centre number 64859 | Candidate number 6103

```

320     self.rect.update(screenWidth* 0.931, pygame.mouse.get_pos()[1] - screenHeight/44, screenWidth/22, screenHeight/22) #change position based on mouse position
321
322     physEng.zoomScale = (0.55 * screenHeight - pygame.mouse.get_pos()[1])/(0.07 * screenHeight) #change zoom value based on Y position
323
324     case 21: #time scale slider
325
326         screenWidth = screen.get_width() #cache screen width
327         screenHeight = screen.get_height() #cache screen height
328
329         screen.blit(font.render(f"Time scale: {str(round(physEng.timeScale, 2))}", True, (255, 255, 255)), (0, screenHeight*0.05)) #draw time scale value on screen
330
331         if screenWidth > 0.115 > pygame.mouse.get_pos()[0] > screenWidth * 0.001: #checks if mouse cursor is within bounds of the slider area:
332
333             self.rect.update(pygame.mouse.get_pos()[0] - screenWidth*0.005, screenHeight* 0.88, screenWidth/22, screenHeight/22) #change position based on mouse position
334
335             if pygame.mouse.get_pos()[0] < screenWidth * 0.055: #if in the real time scale side of slider:
336                 physEng.timeScale = (pygame.mouse.get_pos()[0])/(0.00275 * screenWidth) #change time scale value based on X position
337             else: #if in celestial time scale side:
338                 physEng.timeScale = (pygame.mouse.get_pos()[0])/(0.00000764 * screenWidth * 2) #change time scale value based on X position
339
340     case 24: #close info panel button
341         selectedObject = 0 #clear selected object
342
343     case 26: #freeze velocity/speed
344         selectedObject.velocityFreezeFlag = self.isClicked #set flag in object
345
346     case 27: #move to orbit page of info panel
347         sidePanelPage = 1
348
349     case 28: #move to object page of info panel
350         sidePanelPage = 0
351
352     case 29: #freeze time period
353         selectedObject.timePeriodFreezeFlag = self.isClicked
354
355     case 30: #freeze orbital radius
356         selectedObject.orbitalRadiusFreezeFlag = self.isClicked
357
358     case 31: #freeze orbital velocity
359         selectedObject.orbitalVelocityFreezeFlag = self.isClicked
360
361     case 32: #freeze mass
362         selectedObject.massFreezeFlag = self.isClicked
363
364     case 33: #freeze radius
365         selectedObject.radiusFreezeFlag = self.isClicked
366
367     case 35 | 36 | 37: #load scenario 1/2/3
368
369         physEng.movementOffset ~ pygame.Vector2(0,0) #reset camera offset
370         physEng.zoomScale = 1 #reset zoom offset
371
372         if not physEng.isPaused: #pause simulation if unpause
373             buttonGroup.sprites()[10].isClicked = True #change pause button to pause state
374             buttonGroup.sprites()[10].imageUnscaled = buttonGroup.sprites()[9].imageClicked
375             buttonGroup.sprites()[10].rescale(physEng)
376             physEng.isPaused = True #switch physics engine state
377
378         #delete all objects currently in scene
379         for i in physObjects:
380
381             if i.__class__.__name__ != "Spacecraft": #exclude spacecraft from deletion checks
382
383                 if i is selectedObject:
384                     selectedObject = 0 #clear info panel
385                 if i is physEng.focusedObject:
386                     physEng.focusedObject = 0 #clear object focus
387
388             physObjects.remove(i) #remove from physics objects group
389
390         loadScenario(self.designation - 34, physObjects, redco, greenco, blueco) #to get number for scenario, subtract 34 from designation
391
392         self.isClicked = False #reset button
393         self.imageUnscaled = self.imageClicked
394
395         case _: #catch error for undispatched buttons (SHOULD never happen)
396             print("ERROR: Button designation (self.designation) not recognised for effect.")
397
398
399     def mouseHoldEnd(self, physEng): #called once when the mouse button is unclicked
400
401         if (self.designation > 6 and self.designation <= 9) or self.designation > 19 or self.designation == 5 or (self.designation > 13 and self.designation <= 15): #if a camera control button or a slider:
402
403             self.imageUnscaled = self.imageUnclicked #change graphic
404             self.isClicked = False #set logic to be in unclicked state
405
406             self.rescale(physEng) #update button image
407
408
409     def mouseHoldUpdate(self, physEng, physObjects, spacecraft, buttonGroup, buildMenuGroup): #called every tick when the mouse button is down
410
411         if ((self.designation > 6 and self.designation <= 9) or self.designation > 19 or (self.designation >= 13 and self.designation <= 15)) and self.isClicked: #if a camera movement button or slider, and is currently active:
412
413             self.effect(physEng, physObjects, spacecraft, buttonGroup, buildMenuGroup) #apply button effect
414

```

Orbital Mechanics Simulator – Sam Prince

Centre number 64859 | Candidate number 6103

```

415
416     def update(self, physEng, physObjects, spacecraft, buttonGroup, buildMenuGroup=0, redco=0, greenco=0, blueco=0): #called on mouse click
417
418         if self.collideRect.collidepoint(pygame.mouse.get_pos()): #if mouse cursor is intersecting collision rect:
419
420             if self.isClicked: #if button is in clicked state
421                 self.imageUnscaled = self.imageUnclicked #change graphic
422                 self.isClicked = False #set logic to be in unclicked state
423
424             else: #if button is not in clicked state
425                 self.imageUnscaled = self.imageClicked #change graphic
426                 self.isClicked = True #set logic to be in clicked state
427
428             self.effect(physEng, physObjects, spacecraft, buttonGroup, buildMenuGroup, redco, greenco, blueco) #apply effect of button
429             self.rescale(physEng) #update button image
430
431
432     class TextBox(pygame.sprite.Sprite):
433
434         def __init__(self, designation):
435
436             pygame.sprite.Sprite.__init__(self) #initialise Pygame sprite
437
438             self.text = "0" #set initial empty value
439             self.active = False #set initial state as inactive
440             self.designation = designation
441
442             self.rect = pygame.Rect(0,0,0,0) #set initial rect
443
444             self.colour = (255,255,255) #initially colour textbox white
445
446             self.rescale()
447
448
449         def textInput(self, input, masterObject=0, redco=0, greenco=0, blueco=0):
450
451             global selectedObject
452
453             validInputs = ["0","1","2","3","4","5","6","7","8","9","[","]", ".", ",", "-"]
454
455             if input.key == pygame.K_BACKSPACE: #if backspace pressed, and there is text in the textbox:
456
457                 self.text = str(self.text)[:-1] #set text as itself up to last character (set as string just in case another datatype is passed through)
458
459             elif input.key == pygame.K_RETURN: #if enter key pressed:
460
461                 if len(self.text) == 0 and self.designation == 3: #if the text box for velocity is empty:
462
463                     self.text = "0,0" #set in vector2 format
464
465                     if self.designation != 3: #if not velocity:
466
467                         self.text = self.text.strip(" ,[]-N/A") #ensure positive and valid value for non-negative variables
468
469                         if len(self.text) == 0: #if the text box is empty:
470                             self.text = "0"
471
472                         decimal = False
473                         tempArray = []
474                         for char in self.text: #for each character in input:
475                             if char == ".": #if decimal point:
476                                 if not decimal: #if there has already been a decimal point, do not add this character to the array. otherwise:
477                                     decimal = True #set flag
478                                     tempArray.append(char) #add to array
479                             else: #if a digit:
480                                 tempArray.append(char) #add to array
481
482                         self.text = float("".join(tempArray)) #join chars in array, and convert to float
483
484                         #make changes to object based on text
485                         match self.designation:
486
487                             case 1: #mass
488                                 selectedObject.mass = self.text
489                                 if selectedObject.__class__.__name__ != "Spacecraft":
490                                     selectedObject.correctedMass = selectedObject.mass * (10**24) #convert to corrected mass
491                                 else:
492                                     selectedObject.correctedMass = self.text #otherwise, mass is corrected mass
493
494                             case 2: #radius
495                                 if selectedObject.__class__.__name__ != "Spacecraft": #check for spacecraft: spacecraft has no radius variable
496                                     selectedObject.radius = self.text
497                                     selectedObject.correctedRadius = selectedObject.radius * (10**6) #convert to corrected radius
498
499                             case 3: #velocity
500
501                                 #split into two values (x,y)
502                                 values = self.text.split(",")
503                                 values[0] = float(values[0].strip(" [,)")) #strip values of spaces, commas, square brackets
504                                 values[1] = float(values[1].strip(" [,)"))
505
506                                 selectedObject.velocity = pygame.Vector2(values[0], values[1]) / 1000 #convert back from km/h

```

Orbital Mechanics Simulator – Sam Prince
Centre number 64859 | Candidate number 6103

Orbital Mechanics Simulator – Sam Prince

Centre number 64859 | Candidate number 6103

```

641     |     return #do not update textbox while value is being changed
642
643     match self.designation:
644
645         case 1: #mass
646             self.text = str(round(selectedObject.mass, 1))
647
648         case 2: #radius
649
650             if selectedObject.__class__.__name__ != "Spacecraft": #spacecraft has no radius, so this will not be shown
651                 self.text = str(round(selectedObject.radius, 1))
652             else:
653                 self.text = "N/A"
654
655         case 3: #velocity
656             self.text = str(selectedObject.velocity * 1000) #convert to km/h
657
658         case 4: #speed
659             self.text = str(round(selectedObject.velocity.magnitude() * 1000, 2)) #convert to km/h and round
660
661         case 5: #surface gravity
662
663             if selectedObject.__class__.__name__ != "Spacecraft": #spacecraft has no radius, so this will not be shown
664                 self.text = str(round(selectedObject.calcSurfaceGrav(), 3))
665             else:
666                 self.text = "N/A"
667
668         case 6: #surface temperature (STAR OBJECTS ONLY)
669             self.text = str(selectedObject.temperature)
670
671         case 7: #time period
672
673             self.text = str(round((2*math.pi*(selectedObject.gamePos.distance_to(masterObject.gamePos)**(3/2)))/(math.sqrt(6.67e-11*masterObject.correctedMass)), 2)) #calculate time period, in seconds
674
675         case 8: #orbital radius
676             self.text = str(round(selectedObject.gamePos.distance_to(masterObject.gamePos), 1)) #get orbital radius as distance to master sprite
677
678         case 9: #orbital velocity
679             self.text = str(round(math.sqrt(greatestForce*selectedObject.gamePos.distance_to(masterObject.gamePos)), 2)) #get orbital velocity as root(force * distance to sprite)
680
681         case _: #no designation/unknown
682
683             print("ERROR: Textbox designation {self.designation} unrecognised for update.")
684
685
686 class Planet(pygame.sprite.Sprite):
687
688     def __init__(self, radius, mass, velocity, colour, pos): #init method
689
690         pygame.sprite.Sprite.__init__(self) #initialise Pygame sprite
691
692         self.colour = colour
693         self.radius = abs(radius) #absolute to avoid any negative values messing with pygame
694         self.mass = mass
695
696         #set initial variable freeze flags
697         self.velocityFreezeFlag = False
698         self.timePeriodFreezeFlag = False
699         self.orbitalRadiusFreezeFlag = False
700         self.orbitalVelocityFreezeFlag = False
701         self.massFreezeFlag = False
702         self.radiusFreezeFlag = False
703
704         self.gamePos = pos
705         self.screenPos = pos #set position on screen initially
706         self.velocity = velocity
707
708         #correct values for use in calculations
709         self.correctedMass = self.mass * (10**24)
710         self.correctedRadius = self.radius * (10**6)
711
712         self.image = pygame.Surface([self.radius*2, self.radius*2]) #get surface of sprite to draw onto screen
713         self.image.set_alpha(0) #set surface as invisible
714
715         self.rect = self.image.get_rect(center=self.screenPos) #gets rectangular coordinates of surface for sprite
716
717
718     def calcSurfaceGrav(self): #calculate what the gravitational force would be on the surface
719         # $g = GM/R^2$ 
720
721         if self.correctedRadius == 0:
722             return 0 #prevent div by 0 error
723
724         g = (6.67*(10**-11)*self.correctedMass) / (self.correctedRadius*self.correctedRadius)
725
726         return g
727
728
729     def getAtmosphereHeight(self, physicsObjects): #calculate atmospheric height (PLANET OBJECTS ONLY)
730
731         atmosphereHeight = 0 #clear atmospheric height

```

Orbital Mechanics Simulator – Sam Prince

Centre number 64859 | Candidate number 6103

```

731
732     for i in physicsObjects: #for each object in simulation:
733         if i.__class__.__name__ == "Star": #for each star object:
734
735             if self.gamePos.distance_to(i.gamePos) != 0: #if distance to star is not 0 (prevents div by 0 error):
736
737                 atmosphereHeight += (0.31 * i.temperature * ((0.7 * i.correctedRadius**2) / (4e6 * self.gamePos.distance_to(i.gamePos))))**((1/4)) / (0.829e6 * self.calcSurfacegrav()) #calculate atmospheric height effect and add to variable
738
739             return atmosphereHeight
740
741
742     def update(self, physEng, physicsSprites, buildMenuGroup, buttonGroup):
743
744         self.gamePos += self.velocity * physEng.getTimeScale() #move sprite in terms of simulation
745
746         if self in buildMenuGroup.sprites()[2].pickedObject: #if picked object, go to mouse cursor position
747
748             self.screenPos = pygame.Vector2(pygame.mouse.get_pos())
749
750         else: #otherwise:
751
752             self.screenPos = physEng.getScreenPos(self.gamePos, screen) #convert gamePos > screenPos
753
754     #check for out of bounds, and clamp to within bounds if so
755     if self.screenPos[0] < -2147480000:
756         self.screenPos[0] = -214748000
757     elif self.screenPos[0] > 214748000:
758         self.screenPos[0] = 214748000
759     self.screenPos[0] = self.screenPos[0] % 4096
760
761     if self.screenPos[1] < -214748000:
762         self.screenPos[1] = -214748000
763     elif self.screenPos[1] > 214748000:
764         self.screenPos[1] = 214748000
765
766     if self.__class__.__name__ == "Planet" and buttonGroup.sprites()[2].isClicked: #if planet object and spacecraft controls active:
767
768         pygame.draw.circle(screen, (self.colour[0]*0.25, self.colour[1]*0.25, self.colour[2]*0.25), self.screenPos, (self.radius + self.getAtmosphereHeight(physicsSprites)) * physEng.zoomScale, width=3) #draw atmosphere
769
770         pygame.draw.circle(screen, self.colour, self.screenPos, self.radius * physEng.zoomScale) #draw planet
771
772         self.rect = self.image.get_rect(center=self.screenPos) #update rect, centred at the planet's position
773
774
775
776     class Star(Planet):
777
778         def __init__(self, radius, mass, velocity, temperature, pos, redco, greenco, blueco): #init method
779
780             self.temperature = abs(temperature) #set temperature variable, absolute
781
782             colour = self.getColour(self.temperature, redco, greenco, blueco) #get colour from temperature/RGB curve
783
784             super().__init__(radius, mass, velocity, colour, pos) #initialise parent class
785
786
787         def getColour(self, temperature, redco, greenco, blueco): #gets colour for star (code for function by Docleonard on Stack Overflow)
788
789             if temperature > 40000: #clamp temperature if outside range of graph
790                 return (0, 0, 255)
791
792             red = redco(temperature) #get value for each colour from precalculated curve
793             green = greenco(temperature)
794             blue = blueco(temperature)
795
796             if red > 255: #clamp values to RGB limits
797                 red = 255
798             elif red < 0:
799                 red = 0
800             if green > 255:
801                 green = 255
802             elif green < 0:
803                 green = 0
804             if blue > 255:
805                 blue = 255
806             elif blue < 0:
807                 blue = 0
808
809             return (int(red), int(green), int(blue)) #return colour
810
811
812
813     class Spacecraft(pygame.sprite.Sprite):
814
815         def __init__(self, mass, pos, velocity):
816
817             pygame.sprite.Sprite.__init__(self) #initialise Pygame sprite
818
819             self.mass = mass
820             self.correctedMass = mass #spacecraft mass does not need to be upscaled as planets do
821             self.imageUnscaled = pygame.image.load("spacecraft.png").convert_alpha(screen) #Load spacecraft image from file
822             self.image = self.imageUnscaled #set initial image
823             self.rect = self.image.get_rect() #set initial rect

```

Orbital Mechanics Simulator – Sam Prince

Centre number 64859 | Candidate number 6103

```

824
825     #set initial variable freeze flags
826     self.velocityFreezeFlag = False
827     self.timePeriodFreezeFlag = False
828     self.orbitalRadiusFreezeFlag = False
829     self.orbitalVelocityFreezeFlag = False
830
831     self.gamePos = pos
832     self.screenPos = pos #set position on screen initially
833     self.velocity = velocity
834     self.angle = 0 #set initial angle (in degrees) for rotation of spacecraft
835
836
837     def addVelocity(self):
838
839         radianAngle = (self.angle*math.pi)/180 #convert to radians for sin/cos methods
840
841         if radianAngle <= math.pi/2:
842
843             self.velocity[0] += math.cos(radianAngle)
844             self.velocity[1] += math.sin(radianAngle)
845
846         elif radianAngle <= math.pi:
847
848             self.velocity[0] -= math.cos(math.pi - radianAngle)
849             self.velocity[1] -= math.sin(math.pi - radianAngle)
850
851         elif radianAngle <= math.pi * (3/2):
852
853             self.velocity[0] -= math.cos(radianAngle - math.pi)
854             self.velocity[1] += math.sin(radianAngle - math.pi)
855
856         else:
857
858             self.velocity[1] += math.cos(radianAngle - (math.pi * (3/2)))
859             self.velocity[0] += math.sin(radianAngle - (math.pi * (3/2)))
860
861
862     def update(self, physEng, physObjects, buildMenuGroup, buttonGroup): #buildMenuGroup, buttonGroup and physObjects passed through here as it is passed through in planet class
863
864         self.image = pygame.transform.rotate(pygame.transform.scale_by(self.imageUnscaled, physEng.zoomScale * 0.1), self.angle) #resize image by zoom factor, and rotate to angle
865
866         self.gamePos += self.velocity * physEng.getTimeScale() #move sprite in terms of simulation
867
868         self.screenPos = physEng.getScreenPos(self.gamePos, screen) #convert gamePos -> screenPos
869
870
871         #check for out of bounds, and clamp to within bounds if so
872         if self.screenPos[0] < -214748600:
873             self.screenPos[0] = -214748600
874         elif self.screenPos[0] > 214748600:
875             self.screenPos[0] = 214748600
876         if self.screenPos[1] < -214748600:
877             self.screenPos[1] = -214748600
878         elif self.screenPos[1] > 214748600:
879             self.screenPos[1] = 214748600
880
881         self.rect = self.image.get_rect(center=self.screenPos) #update rect
882
883
884 #####INDEPENDENT METHODS#####
885
886
887     def loadScenario(scenarioNumber, physObjects, redo, greenco, blueco):
888
889         try: #in case of error, program will continue to run
890
891             file = open(f"scenario{scenarioNumber}.txt", "r") #open file object to read
892             for line in file.readlines(): #for each object saved in file:
893
894                 objData = line.split("/") #write data for object to array
895
896                 #split velocity and gamePos variables to put in correct format
897                 for i in range(3,6,2): #velocity = objData[3], gamePos = objData[5], step of 2 to skip over objData[4]
898                     tempArray = objData[i].split(",") #split data into two components
899                     objData[i] = pygame.Vector2(float(tempArray[0]), float(tempArray[1])) #format as Vector2
900
901                 if objData[0] == "s": #if object is a star:
902
903                     obj = Star(float(objData[1]), float(objData[2]), objData[3], float(objData[4]), objData[5], redo, greenco, blueco) #create object from data
904
905                 elif objData[0] == "p": #if object is a planet:
906
907                     #reformat colour data
908                     tempArray = objData[4].split(",") #split data into three components
909                     objData[4] = (float(tempArray[0]), float(tempArray[1]), float(tempArray[2])) #format as RGB tuple
910
911                     obj = Planet(float(objData[1]), float(objData[2]), objData[3], objData[4], objData[5]) #create object from data
912

```

Orbital Mechanics Simulator – Sam Prince

Centre number 64859 | Candidate number 6103

```
913     |     |     #set variable freeze flags
914     |     |     if objData[6] == "1": #mass freeze flag
915     |     |     obj.massFreezeFlag = True
916     |     |     if objData[7] == "1": #radius freeze flag
917     |     |     obj.radiusFreezeFlag = True
918     |     |     if objData[8] == "1": #speed/velocity freeze flag
919     |     |     obj.velocityFreezeFlag = True
920     |     |     if objData[9] == "1": #time period freeze flag
921     |     |     obj.timePeriodFreezeFlag = True
922     |     |     if objData[10] == "1": #orbital radius freeze flag
923     |     |     obj.orbitalRadiusFreezeFlag = True
924     |     |     if objData[11] == "1": #orbital velocity freeze flag
925     |     |     obj.orbitalVelocityFreezeFlag = True
926
927     physObjects.add(obj)
928
929     finally: #always close file to avoid corruption of data
930     |     file.close()
931
932
933 #####PROGRAM INITIALISATION#####
934
935 pygame.init() #pygame setup
936
937 screen = pygame.display.set_mode(size=(1280, 720), flags=(pygame.RESIZABLE | pygame.DOUBLEBUF)) #sets up window to be able to be resized
938 pygame.display.set_caption('Orbital Mechanics Simulator') #names window
939 pygame.display.set_icon(pygame.image.load("icon.png")) #sets window icon
940
941 clock = pygame.time.Clock() #initialise pygame clock
942
943 font = pygame.font.SysFont('Consolas', 30) #initialise font
944
945 async def main(): #async function for Pybag functionality
946
947     #setup for star colour function - global variables so that they do not have to be stored in multiple star objects. code for function by Docleonard on Stack Overflow
948     redco = [ 1.62098281e-82, -5.8311845e-77, 6.66429493e-62, -4.71441850e-67, 1.66429493e-62, -1.50701672e-59, -2.42533086e-53, 8.42586475e-49, 7.94816523e-45, -1.68655179e-39, 7.25404556e-35, -1.28724846e-59, 1.84618419e-53, 6.43113038e-49, 6.85135293e-45, -1.28642374e-39, 5.52273817e-35,
949     greenco = [ 1.21775217e-82, -3.79265302e-77, 5.04300808e-72, -3.57741292e-67, 1.26763387e-62, -1.28724846e-59, 1.84618419e-53, 6.43113038e-49, 6.85135293e-45, -1.28642374e-39, 5.52273817e-35,
950     blueco = [ 2.17374683e-82, -6.82574350e-77, 9.17262316e-72, -6.60390151e-67, 2.40324203e-62, -5.77694976e-59, -3.42234361e-53, 1.26662864e-48, 8.75794575e-45, -2.45089758e-39, 1.10698770e-34,
951
952     redco = numpy.poly1d(redco) #create curves for star function from above data points
953     greenco = numpy.poly1d(greenco)
954     blueco = numpy.poly1d(blueco)
955
956     physEng = engine.PhysicsEngine() #initialise physics engine
957
958 #####CREATE SPRITE GROUPS#####
959
960
961     physObjects = pygame.sprite.Group() #create physics object sprite group
962     buttonGroup = pygame.sprite.Group() #initialise main buttons group
963     equipotentialResolutionButtonGroup = pygame.sprite.Group() #initialise group for resolution buttons
964     hideUIButtonGroup = pygame.sprite.Group() #create button group for the hide UI button
965     sidePanelButtons = pygame.sprite.Group() #initialise side menu buttons group
966     textboxObjectGroup = pygame.sprite.Group() #initialise group for textbox in side panel object page
967     textboxOrbitGroup = pygame.sprite.Group() #initialise group for textbox in side panel orbit page
968     sidePanelOrbitButtons = pygame.sprite.Group() #initialise side menu orbit page buttons group
969     sidePanelCloseButton = pygame.sprite.Group() #initialise close button group for side panel
970     spacecraftGroup = pygame.sprite.Group() #initialise group for spacecraft
971     spacecraftControlGroup = pygame.sprite.Group() #initialise group for spacecraft controls
972     buildMenuGroup = pygame.sprite.Group() #initialise group for build menu
973     loadPresetButtons = pygame.sprite.Group() #initialise group for preset scenario load buttons
974
975
976 #####INITIALISE BUTTONS#####
977
978     for i in range(1,10): #create buttons
979         buttonGroup.add(Button(i, physEng)) #instantiate button and add to update group
980
981     buttonGroup.add(Button(0, physEng)) #create pause button (to be drawn on top of button 5)
982
983     for i in range(20,22): #create slider buttons
984         buttonGroup.add(Button(i, physEng)) #instantiate button and add to update group
985
986
987     hideUIButtonGroup.add(Button(12, physEng)) #create hide UI button and add to its own group
988
989
990     sidePanelCloseButton.add(Button(24, physEng)) #side panel close button is present on both pages, so it will be drawn in its own group
991
992     for i in range(26,28): #create side panel buttons for object page
993         sidePanelObjectButtons.add(Button(i, physEng))
994
995     sidePanelObjectButtons.add(Button(32, physEng)) #create freeze mass button
996     sidePanelObjectButtons.add(Button(33, physEng)) #create freeze radius button
997
998     for i in range(28,32): #create side panel buttons for orbit page
999         sidePanelOrbitButtons.add(Button(i, physEng))
1000
1001
1002
```

Orbital Mechanics Simulator – Sam Prince

Centre number 64859 | Candidate number 6103

```
1003 for i in range(1,6): #create textboxes for side panel pages
1004     textboxObjectGroup.add(Textbox(i))
1005     for i in range(7,10):
1006         textboxOrbitGroup.add(Textbox(i))
1007
1008 surfaceTempTextbox = Textbox(6) #create textbox for star surface temperature
1009
1010
1011 equipotentialResolutionButtonGroup.add(Button(10, physEng)) #create equipotential resolution buttons
1012 equipotentialResolutionButtonGroup.add(Button(11, physEng))
1013
1014 spacecraft = Spacecraft(1, (0,0), pygame.Vector2(0,0)) #initialise spacecraft object
1015 spacecraftGroup.add(spacecraft) #add spacecraft to its draw group
1016 spacecraftImage = pygame.image.load("spacecraft.png").convert_alpha()#cache spacecraft image for drawing to side panel
1017
1018
1019 for i in range(13,16): #create spacecraft control buttons:
1020     spacecraftControlGroup.add(Button(i, physEng))
1021
1022
1023 for i in range(16,20): #create build menu buttons:
1024     buildMenuGroup.add(Button(i, physEng))
1025 buildMenuGroup.add(Button(34, physEng)) #add save/Load scenario button
1026
1027
1028 for i in range(35,38): #create preset scenario selection buttons:
1029     loadPresetButtons.add(Button(i,physEng))
1030
1031
1032 loadScenario(1, physObjects, redco, greenco, blueco) #Load initial scenario from file
1033
1034
1035 #####GLOBAL VARIABLE SETUP#####
1036
1037
1038 global selectedObject
1039 selectedObject = 0 #set no object to be selected at program start
1040 global resolution
1041 resolution = 5 #set initial equipotential line resolution as 5
1042 global sidePanelPage
1043 sidePanelPage = 0 #set initial side panel page to 0 (object info)
1044
1045
1046 ######MAIN RUNNING LOOP#####
1047
1048 running = True
1049 while running: #running Loop
1050
1051     clock.tick(60) #limits FPS to 60
1052
1053     physicsSprites = physObjects.sprites()
1054
1055     for event in pygame.event.get(): #for every key press:
1056
1057         if event.type == pygame.QUIT: #if quit button (top right x) pressed:
1058             running = False
1059
1060         elif event.type == pygame.WINDOWRESIZED: #if the window has been resized:
1061
1062             for i in buttonGroup: #for each button on screen:
1063                 i.rescale(physEng) #rescale all buttons in group
1064
1065             #side panel rescaling:
1066             for i in sidePanelObjectButtons:
1067                 i.rescale(physEng)
1068             for i in textboxObjectGroup:
1069                 i.rescale()
1070
1071             for i in sidePanelOrbitButtons:
1072                 i.rescale(physEng)
1073             for i in textboxOrbitGroup:
1074                 i.rescale()
1075             surfaceTempTextbox.rescale()
1076             sidePanelCloseButton.sprites()[0].rescale(physEng)
1077
1078             hideUIButtonGroup.sprites()[0].rescale(physEng) #rescale hide UI button
1079
1080             for i in equipotentialResolutionButtonGroup:
1081                 i.rescale(physEng)
1082
1083             for i in spacecraftControlGroup:
1084                 i.rescale(physEng)
1085
1086             for i in buildMenuGroup:
1087                 i.rescale(physEng)
1088
1089             for i in loadPresetButtons:
1090                 i.rescale(physEng)
```

Orbital Mechanics Simulator – Sam Prince

Centre number 64859 | Candidate number 6103

```

1095     elif event.type == pygame.MOUSEBUTTONDOWN: #if the mouse has been clicked (or interactive whiteboard tapped)
1096
1097         if buildMenuGroup.sprites()[2].pickedObject == 0: #if no object selected:
1098
1099             buttonGroup.update(physEng, physObjects, spacecraft, buttonGroup, buildMenuGroup) #run checks for buttons
1100             hideUIButtonGroup.update(physEng, physObjects, spacecraft, buttonGroup, buildMenuGroup)
1101
1102         if selectedObject != 0: #if an object is currently selected:
1103
1104             sidePanelCloseButton.update(physEng, physObjects, spacecraft, buttonGroup, buildMenuGroup)
1105
1106         if sidePanelPage == 0: #if on object info page:
1107             sidePanelObjectButtons.update(physEng, physObjects, spacecraft, buttonGroup, buildMenuGroup)
1108             for i in textBoxObjectGroup.sprites():
1109                 i.clicked() #check if textbox clicked
1110             surfaceTempTextBox.clicked()
1111
1112         elif sidePanelPage == 1: #if on orbit info page:
1113             sidePanelOrbitButtons.update(physEng, spacecraft, physObjects, buttonGroup, buildMenuGroup)
1114             for i in textBoxOrbitGroup.sprites():
1115                 i.clicked() #check if textbox clicked
1116
1117         if buttonGroup.sprites()[1].isClicked: #if equipotential lines are being shown:
1118             equipotentialResolutionButtonGroup.update(physEng, physObjects, spacecraft, buttonGroup)
1119
1120         if buttonGroup.sprites()[2].isClicked: #if spacecraft controls active:
1121             spacecraftControlGroup.update(physEng, physObjects, spacecraft, buttonGroup)
1122
1123         if buttonGroup.sprites()[3].isClicked: #if build menu active:
1124             buildMenuGroup.update(physEng, physObjects, spacecraft, buttonGroup, buildMenuGroup, redco, greenco, blueco)
1125             if buildMenuGroup.sprites()[4].isClicked: #if save/load scenarios button pressed:
1126                 loadPresetsButtons.update(physEng, physObjects, spacecraft, buttonGroup, redco-redco, greenco-greenco, blueco-blueco)
1127
1128         if buildMenuGroup.sprites()[5].isClicked: #if deletion button is active:
1129
1130             #deletion checks
1131             for i in physObjects:
1132                 if i.__class__.__name__ != "Spacecraft" and i.rect.collidepoint(pygame.mouse.get_pos()): #exclude spacecraft from deletion checks. if mouse is over object:
1133
1134                 if i is selectedObject:
1135                     selectedObject = 0 #clear info panel
1136                     if i is physEng.focusedObject:
1137                         physEng.focusedObject = 0 #clear object focus
1138
1139             physObjects.remove(i) #remove from physics objects group
1140
1141
1142     for i in physObjects: #for each physics object:
1143
1144         if i.rect.collidepoint(pygame.mouse.get_pos()): #if mouse is over object:
1145
1146             if buildMenuGroup.sprites()[2].isClicked: #if build menu move button is active:
1147
1148                 if buildMenuGroup.sprites()[2].pickedObject == 0: #if no object has been picked up already:
1149
1150                     selectedObject = 0 #clear any selected object
1151                     physEng.focusedObject = 0 #clear any focused object
1152                     buildMenuGroup.sprites()[2].pickedObject = i #set object as picked object for movement
1153
1154             elif i is selectedObject: #if object is already set as selected object:
1155
1156                 physEng.updateCamera(i) #set object as focused object
1157
1158             else: #if not selected object:
1159
1160                 selectedObject = i #set object as the selected object
1161                 objectButtons = sidePanelObjectButtons.sprites() #cache for efficiency
1162                 orbitButtons = sidePanelOrbitButtons.sprites()
1163
1164                 if i.__class__.__name__ != "Spacecraft": #spacecraft object does not have these flags, so they are skipped over if it is the object:
1165                     objectButtons[2].isClicked = i.massFreezeFlag #and mass
1166                     objectButtons[3].isClicked = i.radiusFreezeFlag #and radius
1167
1168                     if i.massFreezeFlag: #set image of the button as the correct state
1169                         objectButtons[2].imageUnscaled = objectButtons[2].imageClicked
1170                     else:
1171                         objectButtons[2].imageUnscaled = objectButtons[2].imageUnclicked
1172
1173                     if i.radiusFreezeFlag: #set image of the button as the correct state
1174                         objectButtons[3].imageUnscaled = objectButtons[3].imageClicked
1175                     else:
1176                         objectButtons[3].imageUnscaled = objectButtons[3].imageUnclicked
1177
1178                     objectButtons[0].isClicked = i.velocityFreezeFlag #switch velocity freeze button to object flag
1179
1180                     orbitButtons[1].isClicked = i.timePeriodFreezeFlag #do the same for time period
1181                     orbitButtons[2].isClicked = i.orbitalRadiusFreezeFlag #and orbital radius
1182                     orbitButtons[3].isClicked = i.orbitalVelocityFreezeFlag #and orbital velocity
1183
1184                     if i.velocityFreezeFlag: #set image of the button as the correct state
1185                         objectButtons[0].imageUnscaled = objectButtons[0].imageClicked
1186                     else:
1187                         objectButtons[0].imageUnscaled = objectButtons[0].imageUnclicked
1188
1189                     if i.timePeriodFreezeFlag: #set image of the button as the correct state

```

Orbital Mechanics Simulator – Sam Prince

Centre number 64859 | Candidate number 6103

```

1190     orbitButtons[i].imageUnscaled = orbitButtons[i].imageClicked
1191     else:
1192         orbitButtons[i].imageScaled = orbitButtons[i].imageUnclicked
1193
1194     if i.orbitalRadiusFreezeFlag: #set image of the button as the correct state
1195         orbitButtons[i].imageUnscaled = orbitButtons[i].imageClicked
1196     else:
1197         orbitButtons[i].imageUnscaled = orbitButtons[i].imageUnclicked
1198
1199     if i.orbitalVelocityFreezeFlag: #set image of the button as the correct state
1200         orbitButtons[i].imageUnscaled = orbitButtons[i].imageClicked
1201     else:
1202         orbitButtons[i].imageUnscaled = orbitButtons[i].imageUnclicked
1203
1204     #update buttons
1205     for j in objectButtons:
1206         j.rescale(phsEng)
1207     for j in orbitButtons:
1208         j.rescale(phsEng)
1209
1210     else: #if an object is already picked up, place down object:
1211
1212     buildMenuSprites = buildMenuGroup.sprites() #cache for readability/efficiency
1213
1214     pos = buildMenuSprites[2].pickedObject.screenPos #cache for readability/efficiency
1215
1216     buildMenuSprites[2].pickedObject.gamePos = pygame.Vector2((pos[0] - screen.get_width()/2)/phsEng.zoomScale) - phsEng.movementOffset[0], ((pos[1] - screen.get_height()/2)/phsEng.zoomScale) - phsEng.movementOffset[1]
1217
1218     buildMenuSprites[2].pickedObject = 0 #place down picked object
1219
1220     for i in range(0,2): #for planet and star placement buttons:
1221
1222         if buildMenuSprites[i].isClicked: #if button is active:
1223
1224             buildMenuSprites[i].isClicked = False #turn button off
1225             buildMenuSprites[i].imageUnscaled = buildMenuSprites[i].imageUnclicked #reset image
1226             buildMenuSprites[i].rescale(phsEng)
1227
1228     elif event.type == pygame.MOUSEBUTTONDOWN: #if mouse button up (after click)
1229
1230         for i in buttonGroup:
1231             i.mouseHoldEnd(phsEng) #run effects when mouse is unpressed
1232
1233         for i in spacecraftControlGroup:
1234             i.mouseHoldEnd(phsEng) #update spacecraft control buttons for mouse unpress
1235
1236
1237
1238     elif event.type == pygame.KEYDOWN: #if any key has been pressed:
1239
1240         #check for active text boxes
1241         for i in textBoxObjectGroup:
1242             if i.active:
1243                 i.textInput(event) #update active text boxes with event data
1244
1245         for i in textBoxOrbitGroup:
1246
1247             if i.active:
1248
1249                 if i.designation == 7 or i.designation == 8 or i.designation == 9: #if time period or orbital radius or orbital velocity box:
1250                     #determine master object
1251                     greatestForce = -1 #set initial greatest force
1252                     masterObject = 0 #set master object as 0
1253
1254                     for j in range(len(physicsSprites)): #for each other sprite in group:
1255
1256                         if selectedObject is not physicsSprites[j]: #skip if object is the same as the object calculating force for
1257
1258                             force = phsEng.calcGravBetweenObjects(physicsSprites[j], physicsSprites[j].gamePos.distance_to(selectedObject.gamePos) * 1e6) #calculate force between objects
1259
1260                             if force > greatestForce: #if this is new greatest force:
1261
1262                                 greatestForce = force #set new greatest force
1263                                 masterObject = physicsSprites[j] #set master object as object force was calculated from
1264
1265                     if not masterObject == 0: #if a master object has been found:
1266                         i.textInput(event, masterObject) #pass master object as parameter
1267
1268                 else:
1269                     i.textInput(event)
1270
1271             if surfaceTempTextbox.active:
1272                 surfaceTempTextbox.textInput(event, redco-redco, greenco-greenco, blueco-blueco)
1273
1274
1275 #####CAMERA MOVEMENT INPUTS#####
1276 if pygame.key.get_pressed([pygame.K_s] | pygame.key.get_pressed()[pygame.K_DOWN]): #down movement key
1277     phsEng.updateCamera("N")
1278
1279 elif pygame.key.get_pressed([pygame.K_w] | pygame.key.get_pressed()[pygame.K_UP]): #up movement key
1280     phsEng.updateCamera("S")
1281
1282 elif pygame.key.get_pressed([pygame.K_d] | pygame.key.get_pressed()[pygame.K_RIGHT]): #right movement key
1283     phsEng.updateCamera("E")
1284
1285
1286 elif pygame.key.get_pressed([pygame.K_a] | pygame.key.get_pressed()[pygame.K_LEFT]): #left movement key
1287     phsEng.updateCamera("W")
1288
1289 screen.fill("black") #background texture, may replace with image later
1290
1291 for i in buttonGroup: #for each button:
1292     i.mouseHoldUpdate(phsEng, physObjects, spacecraft, buttonGroup, buildMenuGroup) #update methods for holding down mouse button
1293
1294 physEng.update(physObjects, screen) #run physics calculation step
1295
1296
1297 #####ORBIT LINES#####
1298 if buttonGroup.sprites()[0].isClicked: #if orbit lines are being shown (orbit lines are first in buttonGroup list as pause button is created later):
1299
1300     pointsToDraw = phsEng.predictOrbit(physicsSprites, 300, screen) #calculate orbit lines
1301
1302     for i in range(len(pointsToDraw) - 2): #for each object's orbit Line (-2 for apoapsis and periapsis sections):
1303
1304         pygame.draw.lines(screen, (255,255,255), False, pointsToDraw[i]) #draw orbit Line
1305
1306         #draw apoapsis markers
1307         for i in pointsToDraw[ len(pointsToDraw) - 2 ]:
1308             pygame.draw.circle(screen, (42,128,0), i, 4) #draw marker
1309
1310         #draw periapsis markers
1311         for i in pointsToDraw[ len(pointsToDraw) - 1 ]:
1312             pygame.draw.circle(screen, (0,102,204), i, 4) #draw marker
1313
1314
1315 physObjects.update(phsEng, physicsSprites, buildMenuGroup, buttonGroup) #apply physics calculation results to objects + draw to screen
1316
1317
1318 #####EQUIPOTENTIAL LINES#####
1319 if buttonGroup.sprites()[1].isClicked: #if equipotential lines are being shown:
1320
1321     equipotentialLines = phsEng.calculEquipotential(physicsSprites, screen, resolution) #get array of equipotential points (can get very large on high resolutions)
1322     for i in equipotentialLines: #for each point in equipotential line:
1323         pygame.draw.line(screen, (22,224,224), i, i) #one point for each pixel on line
1324
1325 if not hideUIButtonGroup.sprites()[0].isClicked:
1326     equipotentialResolutionButtonGroup.draw(screen) #draw resolution settings buttons
1327
1328 if resolution < 5:
1329     screen.blit(pygame.transform.smoothscale(font.render("Warning: there may be performance loss at higher equipotential line resolutions.", True, (255,255,255)), (screen.get_width()*0.7, screen.get_height()*0.04)), (0, 0))
1330

```

Orbital Mechanics Simulator – Sam Prince

Centre number 64859 | Candidate number 6103

```

1331
1332 #####SPACECRAFT CONTROLS#####
1333 if buttonGroup.sprites()[2].isClicked: #if spacecraft controls are active and the UI isn't hidden:
1334
1335     for i in spacecraftControlGroup:
1336         i.mouseHoldUpdate(physEng, physObjects, spacecraft, buttonGroup, buildMenuGroup) #update spacecraft control buttons
1337
1338     if not spacecraft.velocityFreezeFlag: #if spacecraft velocity/speed is not frozen:
1339
1340         #check for atmospheric slowdown
1341         for i in physicsSprites: #for each physics object:
1342             if i._class__._name__ == "Planet": #only planets have atmospheres
1343                 if spacecraft.gamePos.distance_to(i.gamePos) < i.getAtmosphereHeight(physicsSprites) + i.radius: #if spacecraft is within atmosphere:
1344
1345                     if spacecraft.velocity[0] == 0: #check for 0 in horizontal component to prevent div by 0 error
1346                         if spacecraft.velocity[1] < 0: #if negative vertical component:
1347                             spacecraft.velocity[1] = -spacecraft.velocity[0] * 0.9**(physEng.getTimeScale()) #speed change affects only vertical velocity (change sign)
1348                         else:
1349                             spacecraft.velocity[1] = spacecraft.velocity[0] * 0.9**(physEng.getTimeScale()) #speed change affects only vertical velocity
1350
1351             else:
1352                 magnitude = spacecraft.velocity.magnitude() * 0.9**physEng.getTimeScale()
1353                 angle = math.atan2(spacecraft.velocity[1]/spacecraft.velocity[0])
1354
1355                 if spacecraft.velocity[0] < 0: #if negative horizontal component:
1356                     spacecraft.velocity[0] = - (magnitude * math.cos(angle)) #result should be negative
1357                 else:
1358                     spacecraft.velocity[0] = (magnitude * math.cos(angle))
1359
1360                 if spacecraft.velocity[1] < 0: #if negative vertical component:
1361                     spacecraft.velocity[1] = - (magnitude * math.sin(angle)) #result should be negative
1362                 else:
1363                     spacecraft.velocity[1] = (magnitude * math.sin(angle))
1364
1365             spacecraftGroup.draw(screen)
1366
1367         if not hideUIButtonGroup.sprites()[0].isClicked: #if UI isn't hidden:
1368             spacecraftControlGroup.draw(screen)
1369
1370 #####BUILD MENU#####
1371 if buttonGroup.sprites()[3].isClicked and not hideUIButtonGroup.sprites()[0].isClicked: #if build menu is active and the UI isn't hidden:
1372
1373     for i in buildMenuGroup:
1374         i.mouseHoldUpdate(physEng, physObjects, spacecraft, buttonGroup, buildMenuGroup) #update spacecraft control buttons
1375
1376     buildMenuGroup.draw(screen) #draw build menu to screen
1377
1378
1379 if buildMenuGroup.sprites()[4].isClicked: #if save/Load scenario button is pressed:
1380     loadPresetButtons.draw(screen) #draw buttons for Loading preset scenarios
1381
1382
1383 hideUIButtonGroup.draw(screen) #draw hide UI button to screen
1384
1385 if not hideUIButtonGroup.sprites()[0].isClicked: #if UI is not hidden:
1386     buttonGroup.draw(screen) #draw buttons to screen
1387
1388 #####SIDE INFO PANEL#####
1389 if selectedObject != 0 and not hideUIButtonGroup.sprites()[0].isClicked: #if an object is currently selected and the UI isn't hidden:
1390
1391     sidePanelCloseButton.draw(screen) #draw info panel to screen
1392
1393     if sidePanelPage == 0: #object info page
1394
1395         textBuffer = [0,0,0,0,0,0] #generate array for text surfaces
1396
1397         textboxes = textboxObjectGroup.sprites() #cache sprites for efficient call
1398
1399         sidePanelObjectButtons.draw(screen) #draw buttons on side panel
1400
1401         if selectedObject._class__._name__ == "Spacecraft": #if spacecraft object:
1402             textBuffer[1] = font.render("Mass: {textboxes[0].text} kg", True, (0,0,0)) #render font for mass
1403             else:
1404                 textBuffer[2] = font.render("Mass: {textboxes[0].text} x10^24kg", True, (0,0,0)) #render font for mass
1405
1406             textBuffer[3] = font.render("Radius: {textboxes[1].text} x10^6m", True, (0,0,0)) #render font for radius
1407             textBuffer[4] = font.render("Velocity: {textboxes[2].text} km/h", True, (0,0,0)) #render font for velocity, converted to kilometres per hour
1408             textBuffer[5] = font.render("Speed: {textboxes[3].text} km/h", True, (0,0,0)) #render font for speed, converted to kilometres per hour
1409             textBuffer[6] = font.render("Surface gravity: {textboxes[4].text} m/s^2", True, (0,0,0)) #render font for surface gravity
1410
1411         if selectedObject._class__._name__ == "Star": #if selected object is a star
1412
1413             textBuffer.append(font.render("Surface temperature: {surfaceTempTextbox.text}K", True, (0,0,0))) #render font for surface temperature
1414             surfaceTempTextbox.update() #update star surface temperature textbox
1415             pygame.draw.rect(screen, surfaceTempTextbox.colour, surfaceTempTextbox.rect) #draw textbox for star surface temperature
1416
1417         elif selectedObject._class__._name__ == "Planet": #if selected object is a planet:
1418
1419             textBuffer.append(font.render("Atmospheric height: {round(selectedObject.getAtmosphereHeight(physicsSprites), 2)} km", True, (0,0,0))) #render font for atmospheric height
1420
1421         textboxObjectGroup.update() #set values in textboxes to that of the selected object
1422
1423         #place textboxes on screen
1424         for i in textboxObjectGroup.sprites():
1425

```

Orbital Mechanics Simulator – Sam Prince
Centre number 64859 | Candidate number 6103

```

1426     |     pygame.draw.rect(screen, i.colour, i.rect)
1427
1428
1429 elif sidePanelPage == 1: #orbit info page
1430
1431     sidePanelOrbitButtons.draw(screen) #draw buttons on side panel
1432
1433     #determine master object
1434     greatestForce = -1 #set initial greatest force as -1
1435     masterObject = selectedObject #set master object as itself for time being
1436
1437     for j in range(len(physicsSprites)):
1438
1439         if selectedObject != physicsSprites[j]: #skip if object is the same as the object calculating force for
1440
1441             force = physEng.calcGravBetweenObjects(physicsSprites[j], physicsSprites[j].gamePos.distance_to(selectedObject.gamePos) * 1e6) #calculate force between objects
1442
1443             if force > greatestForce: #if this is new greatest force:
1444
1445                 greatestForce = force #set new greatest force
1446                 masterObject = physicsSprites[j] #set master object as object force was calculated from
1447
1448
1449     textBuffer = [0,0,0,0,0] #generate array for text surfaces
1450
1451     textboxes = textBoxOrbitGroup.sprites() #cache sprites for efficient call
1452
1453     textBuffer[2] = font.render("Time period: " + textboxes[0].text, True, (0,0,0)) #calculate time period and render to font
1454     textBuffer[3] = font.render("Orbital radius: " + textboxes[1].text, True, (0,0,0)) #render font for orbital radius
1455     textBuffer[4] = font.render("Orbital velocity: " + textboxes[2].text, True, (0,0,0)) #render font for orbital velocity
1456
1457     textBoxOrbitGroup.update(masterObject, greatestForce) #update textbox values to that of the selected object
1458
1459     #place textboxes on screen
1460     for i in textBoxOrbitGroup.sprites():
1461         pygame.draw.rect(screen, i.colour, i.rect)
1462
1463
1464     #render side info panel
1465     if selectedObject.__class__.__name__ == "Spacescraft": #if spacecraft object:
1466
1467         image = pygame.transform.rotate(pygame.transform.scale_by(spacescraftImage, screen.get_width()*0.0001), spacescraft.angle) #scale spacecraft image (uses separate variable to keep quality) and rotate to spacecraft angle
1468         screen.blit(image, (screen.get_width()*0.07, screen.get_height()*0.37)) #paste to screen
1469         textBuffer[1] = font.render("Colour: N/A", True, (0,0,0)) #render font for colour
1470
1471     else:
1472
1473
1474         pygame.draw.circle(screen, selectedObject.colour, (screen.get_width()*0.07, screen.get_height()*0.21), screen.get_width()*0.02) #draw copy of selected object at top of info box
1475         textBuffer[1] = font.render("Colour: " + str(selectedObject.colour), True, (0,0,0)) #render font for colour
1476
1477         textBuffer[0] = font.render(str(selectedObject.__class__.__name__), True, (0,0,0)) #render font for class name (planet/star/spacescraft)
1478
1479         #rescale and position text
1480         for i in range(1, len(textBuffer)): #for each rendered text surface (excludes class name):
1481
1482             textBuffer[i] = pygame.transform.smoothscale(textBuffer[i], (screen.get_width()*0.13, screen.get_height()*0.03)) #rescale
1483
1484             if i == 1: #if colour text (separate to add spacing):
1485                 screen.blit(textBuffer[i], (0, screen.get_height()*0.35)) #blit colour to screen
1486             else:
1487                 screen.blit(textBuffer[i], (0, screen.get_height()*(0.425 + 0.03*(i - 1)))) #blit text to screen
1488
1489         textBuffer[0] = pygame.transform.smoothscale(textBuffer[0], (screen.get_width()*0.06, screen.get_height()*0.05)) #rescale class name text
1490         screen.blit(textBuffer[0], (0, screen.get_height()*0.28)) #blit class name to screen
1491
1492
1493     if physEng.focusedObject != 0: #if there is an object in focus:
1494         screen.blit(font.render(str(physEng.focusedObject.gamePos), True, (255, 255, 255)), (0,0)) #draw coordinates of focused object on screen
1495
1496     pygame.display.update() #updates the screen
1497
1498     await asyncio.sleep(0) #delay (pythag)
1499
1500     pygame.quit() #destroys window and quits pygame on program end
1501
1502     asyncio.run(main()) #run main program (pythag)

```

Orbital Mechanics Simulator – Sam Prince

Centre number 64859 | Candidate number 6103

```

40     case "W": moveUp
41         self.movementOffset[1] -= 5
42
43     case "S": moveDown
44         self.movementOffset[1] += 5
45
46     case "A": moveLeft
47         self.movementOffset[0] -= 5
48
49     case "D": moveRight
50         self.movementOffset[0] += 5
51
52     case '_': objectToFocusOnPassedThrough
53
54         self.focusedObject = direction #set focused object
55
56     #self.movementOffset += Vector2(screen.get_width()/2, screen.get_height()/2) - self.focusedObject.screenPos #set movement offset to that of focused object and account for previous offset
57
58
59 def getScreenPos(self, gamePos, screen): #convert gamePos to screenPos
60
61     screenPos = Vector2((self.movementOffset[0] + gamePos[0]) * self.zoomScale + screen.get_width()/2, (self.movementOffset[1] + gamePos[1]) * self.zoomScale + screen.get_height()/2) #set position on screen to position in terms of game, with zoom and focus offset
62
63     return screenPos
64
65
66 def getTimeScale(self): #check pause state, and return time scale to planets if unpause
67
68     if self.isPaused:
69         return 0 #paused: no movement
70     else:
71         return self.timeScale / 3000
72
73
74 def calculatePotential(self, objects, screen, resolution):
75
76     equipotentialArray = []
77
78     #for each point on the visible screen:
79     #for i in range(0, screen.get_width(), resolution):
80     #    for j in range(0, screen.get_height(), resolution):
81
82         force = 0 #set initial force on point to 0
83         #for k in objects: #for each physics object:
84
85             #    force += self.calcGravBetweenObjects(k, k.screenPos.distance_to(Vector2(i,j))) #calculate force for each individual object on that point
86
87             #if (999 < force < 1000) | (1999 < force < 2000) | (2999 < force < 3000) | (3999 < force < 4000) | (4999 < force < 5000) | (5999 < force < 6000) | (7999 < force < 8000) | (7999 < force < 8000) | (8999 < force < 9000) | (9999 < force < 10000):
88
89                 equipotentialArray.append(Vector2(i,j)) #add to 2d array based on force it meets, if is the same as one of the forces
90
91
92     return equipotentialArray
93
94
95 def predictOrbit(self, sprites, points, screen):
96
97     pointArray = [[Vector2(0,0) for l in range(points+1)] for j in range(len(sprites))] #create array of points for number of points to look ahead for, for each object
98
99     velocityArray = [] #create array to store velocity of objects at future points
100
101    #for l in range(len(sprites)):
102    #    pointArray[0][l] = sprites[l].gamePos #set the initial line to be drawn as the centre of the planet
103    #    velocityArray.append(Vector2(sprites[l].velocity)) #get velocities of all sprites in object group
104
105    #for l in range(1, points+1): #for the requested number of points:
106    #    masters = self.generatePointList(l, pointArray, sprites, velocityArray) #generate a set of points for l steps in the future
107
108    #for l in pointArray: #for each object:
109
110        #for j in range(0, points+1): #for each vector2 coordinate:
111            #    l[j] = self.getScreenPos([l[j]], screen) #convert to screenPos (pass through 0 for object)
112
113    self.calculateApogeeAndPerigee(pointArray, sprites, masters) #calculate + append points for apogee and perigee markers
114
115    return pointArray
116
117
118 def generatePointList(self, pointLevel, array, sprites, velocities):
119
120     masterObjects = [0 for l in range(len(sprites))] #create array of master sprites for each object for apogee/perigee calculations
121
122     for l in range(len(sprites)): #for each sprite:
123
124         greatestForce = 0 #set greatest force acting on that object to 0 initially
125
126         if not sprites[l].velocityFreezeFlag: #if velocity has not been frozen:
127
128             for j in range(len(sprites)): #for each other sprite in group:
129
130                 if l != j: #skip if object is the same as the object calculating force for
131                     force = self.calcGravBetweenObjects(sprites[l], array[l][pointLevel-1].distance_to(array[j][pointLevel-1]) * 1e6) #calculate force between objects
132
133                 if force > greatestForce: #if this is new greatest force:
134
135                     greatestForce = force #set new greatest force
136                     masterObjects[l] = j #set master object as index in sprites[] of object force was calculated from
137
138             velocities[l] += Vector2((sprites[l].gamePos[0]*math.sqrt((sprites[l].gamePos[0]-sprites[l].gamePos[1])**2+(sprites[l].gamePos[1]-sprites[l].gamePos[2])**2), (sprites[l].gamePos[1]-sprites[l].gamePos[2])))
139
140             array[l][pointLevel] = array[l][pointLevel-1] + velocities[l] #set point at level in array as predicted point
141
142     return masterObjects #return array of master objects for apogee/perigee calculations
143
144
145 def calcApogeeAndPerigee(self, pointArray, sprites, masters):
146
147     perigeeArray = []
148     apogeeArray = []
149
150     for l in range(len(sprites)): #for each object:
151
152         minDist = 99999999999999 #set initial min distance as a large number (can't do -1 so this is the best we are getting)
153         minPointIndex = pointArray[l][0] #set initial minimum point as the centre of the object
154         maxDist = -1 #set initial max distance to -1
155         maxPointIndex = pointArray[l][0] #set initial maximum point as the centre of the object
156
157         for j in pointArray[l]: #for each point in the object's set of points:
158
159             pointDist = sprites[masters[l]].screenPos.distance_to(j) #calculate distance between point and master object
160
161             if pointDist < minDist: #if master object of sprite's distance to point is less than the minimum:
162                 minDist = pointDist #reset min distance
163                 minPointIndex = j
164
165             if pointDist > maxDist: #if master object of sprite's distance to point is greater than the maximum:
166                 maxDist = pointDist #reset max distance
167                 maxPointIndex = j
168
169         apogeeArray.append(maxPointIndex)
170         perigeeArray.append(minPointIndex)
171
172     pointArray.append(apogeeArray)
173     pointArray.append(perigeeArray)
174
175
176 def calcGravBetweenObjects(self, objA, objB, radius):
177
178     #Force = (G * mass1 * mass2)/radius**2
179     #G = 6.67e-11
180     #r = radius
181     #a = r * f/m
182     #f = G * m1 * m2 / r**2
183     #v = aT
184     #r = v * t
185
186     #if radius == 0:
187     #    return 0 #no force - prevents divide by 0 in next step
188
189     force = (6.67e-11 * objB.correctedMass)/(radius**2) #physics equation, not including mass of object A as it cancels later
190
191     return force

```

References

- Cesare. (2018, June 7). Retrieved December 15, 2023, from Universe Sandbox:
<https://universesandbox.com/forum/index.php?topic=21647.0>
- CodersLegacy. (n.d.). *Multithreading Tutorial*. Retrieved March 19, 2024, from CodersLegacy:
<https://coderslegacy.com/python/multithreading-tutorial/>
- DocLeonard. (2019, March 20). *Star B-V color index to apparent RGB color*. Retrieved January 4, 2024, from Stack Overflow: <https://stackoverflow.com/questions/21977786/star-b-v-color-index-to-apparent-rgb-color>
- Effective Temperature*. (n.d.). Retrieved March 21, 2024, from Wikipedia:
https://en.wikipedia.org/wiki/Effective_temperature
- Given, D. (2017, March 10). Retrieved December 15, 2023, from Stack Exchange:
<https://space.stackexchange.com/questions/7768/ksp-like-interactive-map-of-the-real-solar-system>
- Larkin, R. (2018, February 27). Retrieved December 15, 2023, from Gaming Nexus:
<https://www.gamingnexus.com/Article/5588/Kerbal-Space-Program-Enhanced-Edition/>
- Luminosity*. (n.d.). Retrieved March 21, 2024, from Wikipedia:
<https://en.wikipedia.org/wiki/Luminosity>
- Morojna, S. M. (2022, February 18). Retrieved December 15, 2023, from Steam:
https://store.steampowered.com/app/1718870/Spaceflight_Simulator/
- Pygame Documentation*. (n.d.). Retrieved November 23, 2023, from Pygame.org:
<https://www.pygame.org/docs/>
- Scale Height*. (n.d.). Retrieved March 21, 2024, from Wikipedia:
https://en.wikipedia.org/wiki/Scale_height#:~:text=As%20a%20function%20of%20temperature,heights%20for%20representative%20air%20temperatures.
- spaceflightsim1157. (2022, March 23). Retrieved December 15, 2023, from TikTok:
<https://www.tiktok.com/@spaceflightsim1157/video/7078420557552831787>