Orbyte

Author: Joshua James-Lee

Candidate number: 7130

Centre name: Magdalen College School Oxford

Centre number: 62323

Qualification code: 7517D

Date: May 2023

Analysis

Background information

Understanding orbits and gravitational fields is important. So important that exam boards, such as AQA, have stipulated that physics students should study the "Orbits of planets and satellites" (A-Level Physics Specification). Even at GCSE, according to specification point 4.8.1.3 of the AQA GCSE Physics specification, students should learn: how "satellites stay in orbit" with reference to the sun, earth, moon and artificial satellites"; "The Circular Motion of Satellites" and "How the Speed of a Satellite affects its radius". It's therefore necessary for there to be graphical simulations of orbits in order to aid the teaching of these principles.

Investigation

Interview with a Physics Teacher

The following are responses to questions given to Mr Rice, a physics teacher at Magdalen College School Oxford, who teaches both A Level and GCSE Physics.

How does the teaching of gravitational fields and orbits benefit from the use of graphical computer simulations?
|| "Gravitational fields can be effectively taught through simulations, especially sandbox-style simulations, as a result of the ability to become hands-on with the Physics and transport gravity, which normally is not viable to show many demonstrations of in the lab, into a practical science for the students." ||

Can you think of any limitations with current simulations used in classes? (e.g. the Phet Orbit Simulation)
|| "Relatively limited options in the PhET simulation for which satellites are in play and how many of them you would want (although this may become computationally expensive quite quickly)." ||

What are some features you like to see in a 3D orbital simulation?
|| "Free movement of the camera, multiple possible bodies and their interactions." ||

Should the simulation include planets, satellites, or both?
|| "Both." ||

This software is being designed with education in mind, so are there any other features for an educational simulation that you would like to see implemented? (That perhaps aren't included in others at the moment.)

|| "Force vectors, perhaps even a measurement of the energies of different bodies such that you can see conservation of energy during non-circular orbits." ||

Analysis of the current industry standard

The following investigation[1] into available orbit simulations has been informed by what is used in a classroom to educate GCSE and A-Level students, and a correspondence with the UK Space Agency.

Classroom

Currently, the PHET orbital simulation is used in classrooms as a teaching aid and a way for students to interact with what can otherwise be quite and abstract topic in the Physics courses. The following analysis has been done on the version publicly accessible during 08/2022.

This simulation is in 2 Dimensions only and features limited options for what is orbiting within the simulation. At most, only 3 bodies are simulated. The simulation draws the path of the orbiting bodies, their velocity vector (represented as an arrow) and the vector of gravitational force acting on the orbiting body. The actual values of force, energy and velocity are excluded from the demonstration, though masses can be included. The screen can also become cluttered with arrows and lines making it difficult to visualize what is going on in the simulation if zoomed out. The speed at which time passes is also limited, making it slow to see patterns or divergent paths emerge.

Gravity and Orbits
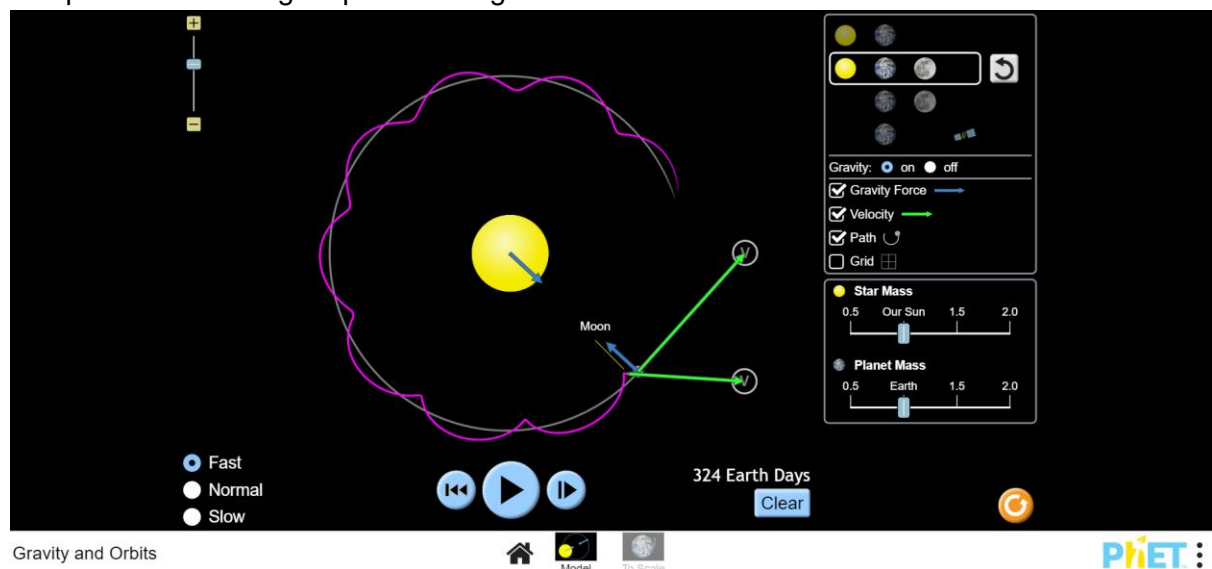
Satellite visualization

"Satvis" [SATVIS] is an online satellite orbit visualization, featuring real-time display of satellite positions in 3D. This implementation excels in its user interface which is both compact and detailed enough to provide sufficient customizability of the display.

The biggest draw-back of this example is that it doesn't display any of the physics properties of the orbiting body. It is also a simulation limited to satellites and not a general purpose orbit simulator. What should be taken away from examining this solution is the benefits of a

---

[1] Each example presented is an amazing solution. Observations, praise and criticisms have been made of the simulations in comparison to what I am trying to achieve. These are superb resources, and I am grateful for being able to access such simulations online.

compact user interface, especially the ability to select an individual orbiting body to display more information about it.
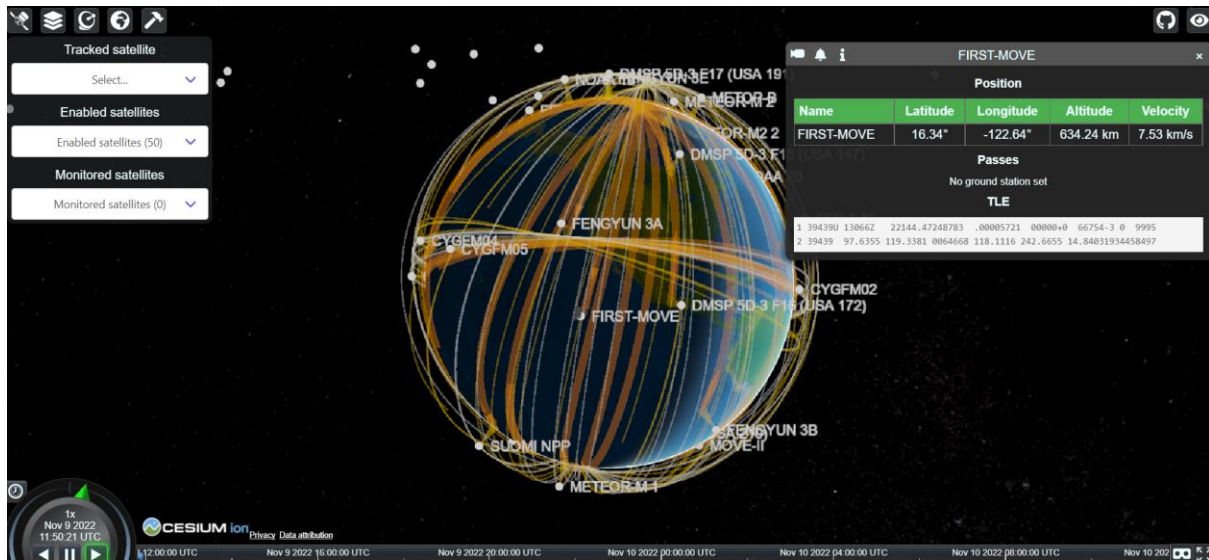
Satellite Explorer

"SatelliteXplorer" [SATELLITE EXPLORER] is another website that exclusively displays the orbit of satellites around the earth. It is similar to "Satvis", and features a clean minimalist User Interface. Both the landing page view and the individual satellite view are aesthetically pleasing. The page displaying an individual satellite's information is a good endorsement for the use of a sidebar to keep the screen tidy.

This simulation is also real-time and features information about the orbiting body that is interesting to read, but largely academically irrelevant. There seems to be no way to speed up the passage of time, and if modified for educational purposes, it would be helpful to remove 3D models (though aesthetic) as a necessary abstraction to see the orbiting bodies as shapes with regular geometry to aid with the thought process driving academic physics. Like "Satvis", the website is not a general-purpose orbit simulation and does not display the physical attributes of the object, such as force, acceleration and velocity vectors. My implementation should therefore draw from the clean presentation of this solution, expand upon its capabilities, and develop upon its content for a classroom, engineer or hobbyist.
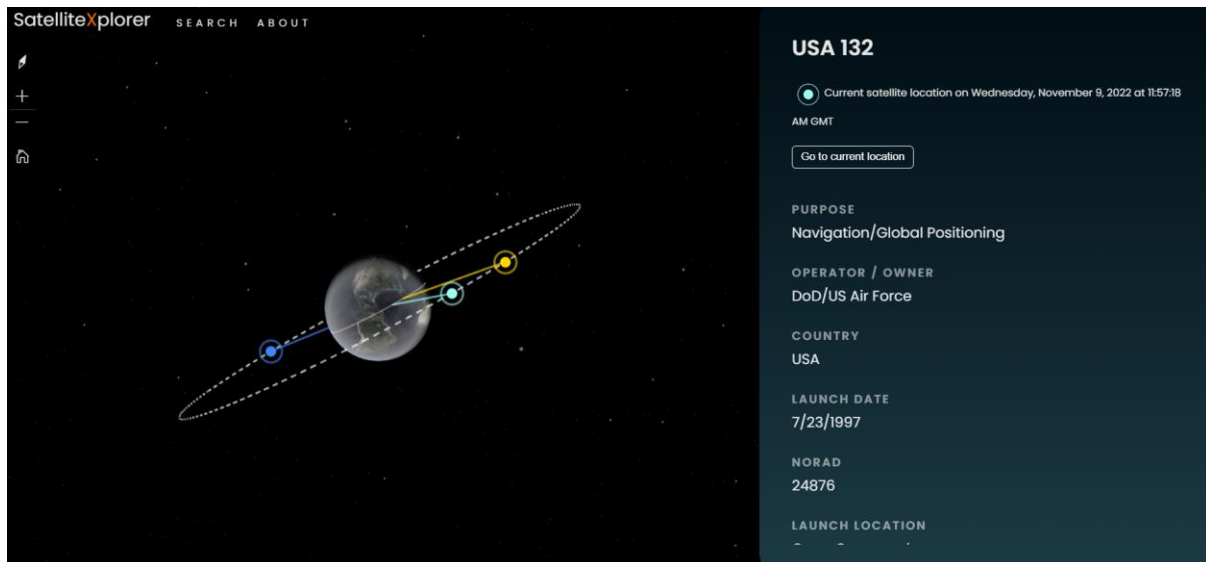
Overview of problem

As outlined above, it is important to understand how objects in orbit behave. In class, students learn about how the velocity of a satellite affects its radius of orbit and how the mass of bodies involved can affect the trajectories. In the spirit of inspiring and encouraging interest in the subject that could grow into a love for astrophysics, the simulation ought to be both detailed and aesthetically pleasing, which is not the case for current simulations.

Current solutions to orbital simulations are mostly 2-Dimensional, with limited graphics, details and little customizability. If you want to see the orbital path of many satellites all in orbit at once, in 3D, then a new simulation is required.

Limitations and constraints

The simulation is intended for educational purposes and will therefore need to be easy to use for both students and teachers. The range of ages most likely to be interacting with the software is those studying the GCSE syllabus to those studying A-Level and beyond. The software should also be easy to use and intuitive for teachers, in order to streamline the teaching process and avoid wasted lesson time.

This "ease of use" will manifest itself as a minimalist graphical user interface, with most of the screen space being dedicated to the graphical simulation, in order to ensure engagement and immersion rather than distraction with the peripheral parameters. Naturally, as an educational simulation, the details of orbiting bodies should be displayed in a compact way that both facilitates adequate detail without cluttering the graphic.

The mathematics behind simulating multiple body orbits, especially past 2 body orbits, becomes difficult to implement and may prove confusing to the students that are using the simulation as this mathematics transcends the A Level specification. It may be necessary to make some approximations in order to keep the simulation useful as a teaching resource at this level.

Performance is also an important consideration. In the interest of wide accessibility of the simulation, optimisations will be made such that the simulation is still performant on lower tier systems. This can be done in either "graphics tier" features or restrictions made to the number of orbiting bodies and calculations made per second, such that the load on a lower end processor be reduced.

Input, Process, Storage, Output (IPSO)

IPSO	Information	Evidence
Input	Orbiting Body Information: - Name - Initial Position - Initial Velocity - Gravitational Constant - Mass of central body - Mass of orbiting body	Educational simulations and satellite position trackers.
Output	Orbiting Body Annotations: - Force Vectors - Velocity Vectors - Path Orbiting Body Information: - Acceleration - Velocity - Period - Radius	Interview and current solution analysis.
Storage	Update Queue: - Orbiting body objects GUI elements Last simulation state and setup.	
Processing	- Runge-Kutta 4 for solving ordinary differential equations - World space to screen space calculation for 3D camera projection. - Rotation of bodies & entire system by rotation matrices. - Hashing algorithm to save data to hash table for between-session storage.	Mathematics behind simulating orbits. Mathematics behind camera projections. A Level Further Mathematics: Rotation Matrices.

Data dictionary

Abstract data types: Vector

Data item	Data type	Validation	Sample data
Velocity	Vector		(1, 1, 1)

Mass	Float		5.972 × 10^24
Position	Vector		(1, 1, 1)
Gravitational Constant	Float		6.6743 × 10^-11
Acceleration	Vector		(1,1,1)
Time Step	Float		100

Data volumes

As seen in the Phet physics simulation. Satellite trackers have more satellites on display than the educational simulation. In all there is only one central body.

Data object	Volume of data
Orbiting Planets	~2
Central Body	1
Satellites	~1

System flowchart
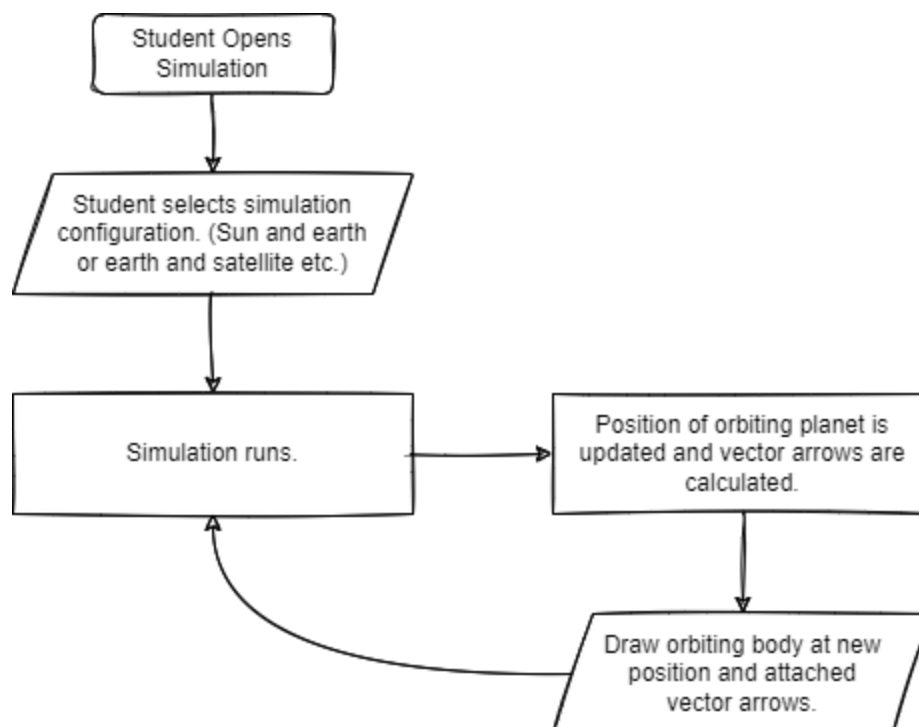
Runge-Kutta 4

Given an initial position and velocity relative to a central mass, used as a focus for the orbiting object. An Ordinary Differential Equation solver is required. For this reason RK4 is used as it provides a more accurate approximation than the Euler method.

$$\overrightarrow{a_{gravity}} = \dot{v} = \dot{s}$$
$$\vec{r}(t) = \int \vec{v} \, dt = \iint \vec{a} \, dt$$

Handling 3D Vectors. Acceleration due to gravity is equal to the derivative of velocity and the second derivative of displacement. It is necessary to solve for displacement given an acceleration and velocity for the purposes of this simulation.

$$\vec{a} = \frac{-\mu}{|r|^3} \times \vec{r}$$

Where:

$$\mu = Gm_{central}$$

From this we extrapolate the ODE Method:

    Function ode(time [float], position [Vector3], velocity [Vector3])
        s = position
        *a = -mu * r / (s.magnitude * s.magnitude * s.magnitude)*
        return [velocity, a]
    EndFunction

This method is then used within each RK4 step:
Function rk4_step(time [float], position [Vector3], velocity [Vector3], step_size [float])
        k1 = f(time, position, velocity)
        *k2 = f(time + 0.5 * step_size, position + 0.5 * k1 * step_size, velocity + 0.5 * k1 * step_size)*
        *k3 = f(time + 0.5 * step_size, position + 0.5 * k2 * step_size, velocity + 0.5 * k1 * step_size)*
        *k4 = f(time + step_size, position + k3 * step_size, velocity + k3 * step_size)*
        *return [position + step_size / 6.0 * (k1 + k2 * 2 + k3 * 3 + k4), velocity + step_size / 6.0 * (k1 + k2 * 2 + k3 * 3 + k4)]*
EndFunction

Objectives

The solution I will implement will involve using the Simple DirectMedia Layer (SDL), which is a "cross-platform development library designed to provide low level access to audio, keyboard, mouse, joystick, and graphics hardware via OpenGL/Direct3D/Metal/Vulkan. It is used by video playback software, emulators, and popular games." [SDL_WIKI]

Its ability to directly interface with graphics hardware and low-level nature will make it a suitable environment to develop a performant simulation, giving me control over how memory and other resources are being used, and freedom to implement graphics and UI

how I choose to, in order to best accomplish the goal of making an educational general orbit simulation.

The general mathematics governing (broadly) how the simulation works and the pseudocode describing the functions that I will implement RK4 with have been explained above. The mathematics behind how 3D shapes will be represented will be covered in the Documented Design section, along with an expanded description of each simulation step.

No.	Objective	Performance criteria
Version 0.0	**Fundamental Setup**	
0.0.1	Create project and set up using SDL.	Successfully setup SDL dependencies. Compile with no errors.
0.0.2	Divide initialization steps into separate procedures and draw window.	Window is drawn and can be closed.
0.0.3	Get User Keyboard Input and log it to console	User can press keys and corresponding debug message will be outputted to console.
Version 0.1	**Orbit Object Implementation**	
0.1.1.0	Set up Central Object Class (Accessors and Placeholder Methods)	Debugging procedures and Accessor methods within the class to access and display class attributes.
0.1.1.1	Set up Orbiting Body Class (Accessors and Placeholder Methods)	Debugging procedures and Accessor methods within the class to access and display class attributes. Must follow good OOP practices.
0.1.1.2	Set up child class for satellites (Accessors and Placeholder Methods)	Debugging procedures and Accessor methods within the class to access and display class attributes.
0.1.2.0	Implement Update Method for Orbit Body Class and Satellite Class	Mostly comments for what procedures need to occur each simulation step. Connect internal methods that should be called every frame, so that the entire update can be handled via one public method.

0.1.3.0	Create Simulation Class with attributes and accessors	Should contain necessary attributes, including a queue of orbiting bodies, a central body and various parameters. Implement all necessary methods / placeholders and include comments.
0.1.3.1	Outline Simulation Class Update Method	Method in place to handle entire application update.
Version 0.2	**Orbital Simulation Mathematics**	
0.2.1.0	Implement Runge-Kutta 4 step in a procedure.	2nd ODE solver set up so that orbits can be simulated in realtime (1 second = 1 second). Using mathematics covered above.
0.2.2.0	Integrate new RK4 method into the main program update procedure.	Connect RK4 method with Body update method and implement means to update all simulation bodies via simulation mainloop.
0.2.3.0	Create Abstract Data Type: Vector3	Data type with accessor methods for x, y, z.
0.2.3.1	Create Magnitude Function in Vector Class	Function returns magnitude of the Vector.
0.2.3.2	Create Normalize Function in Vector Class	Function returns the normalized vector.
0.2.3.3	Overload operators for Vector3 struct	Implement vector addition, scalar-vector multiplication, dot product.
Version 0.3	**Simulation Step**	
0.3.1.0	Configure simulation class such that only one orbiting body is in the simulation queue. To sense-check results, make the parameters of the central body that of the sun and the parameters of the orbiting body that of the earth.	Earth instantiated at the correct distance from the sun, with correct velocity. Calculate orbit period (projection) and sense-check results: result should have error under 2%.
0.3.2.0	In Orbital Body Update Method, pass parameters to RK4 step and start to calculate position as a function of time. (Note this will	Debug methods each update to confirm values are changing, verification

	be modified time, depending on the time-step parameter of the simulation.)	of correct calculation will be done at a later stage.
0.3.3.0	Write methods for the Orbital Body Class to output the current position.	Find the time period of the earth's orbit in the simulation and compare to known values. Note the error (if any), and then refer to previous steps to fix bugs (if any).
0.3.4.0	Simulation should have acceptable performance for reasonable numbers of orbits.	25 orbits in the simulation should have at least 30 FPS.
Version 0.4	**Graphics I**	
0.4.1.0	Create method to draw a pixel to the screen given an x and y coordinate, using the SDL libraries.	Pixel drawn at expected location.
0.4.2.0	Create method to draw a line of pixels between two points.	Draw a 3D shape on screen.
0.4.3.0	Create method to rotate a collection of 3D points about a centroid.	Rotating 3D shape visible on screen. Constant 3D rotation over time (via update method) with no scaling / distortion.
Version 0.5	**Satellite Child Class**	
0.5.1.0	Instantiate one satellite in the simulation class main method. Set it's "parent" to be the earth and set its parameters to be similar to that of the moon. (*Note: We are presently forming a reductive version of our solar system, with only the sun, earth and moon, so that we can sense-check the results of our simulation.*)	Debug Methods to confirm attributes are set correctly.
Version 0.6	**Graphics II**	
0.6.1.0	Draw the central body to the screen using custom graphics implementation.	Have a 3D shape drawn to the screen at the position corresponding to the origin.
0.6.2.0	Draw Orbiting bodies to the screen using the custom graphics implementation. Have the draw function in the update method for the orbiting body class.	Have a 3D shape drawn to the screen corresponding to the current position of the orbiting body. Drawn every frame.

0.6.3.0	Draw Satellites.	Have a 3D shape drawn to the screen corresponding to the current position of the satellite. Drawn every frame.
Version 0.7	**User Interface I**	**Draw User Interface using a library for GUIs in SDL.**
0.7.1.0	Draw sidebar with placeholder text corresponding to parameters for the simulation.	Text is drawn to the sidebar, occupying a portion of the side of the screen.
0.7.2.0	Add Placeholder buttons	Clicking button outputs a debug message to console.
0.7.2.1	Add Button to instantiate a new orbit body	Clicking button instantiates a new orbit with default parameters.
0.7.2.2	Add Button to pause simulation	Clicking button pauses simulation.
0.7.3.0	Add a panel that functions as an "inspector." *This will describe attributes of the object currently in focus.*	Placeholder panel.
Version 0.8	**User Interface II**	
0.8.1.0	Connect simulation to the User Interface. UI should access and display orbiting body class attributes.	See object attributes shown for 1 orbiting body.
0.8.2.0	Allow user to interact with any orbiting body such that its information is shown on the extended UI.	Inspector window shows selected object's properties
0.8.3.0	Add input fields to general simulation parameters.	Able to change simulation parameters via input fields.
0.8.3.1	Add input fields to inspector panel for orbit bodies.	Able to change orbit parameters via input fields.
0.8.3.2	Add Button within inspector to instantiate satellite orbit around current body.	Clicking button instantiates a new satellite around current body.
0.8.3.3	Add Button within inspector to reset selected	Clicking button resets

	orbit.	orbit to initial parameters.
0.8.3.4	Add Button within inspector to delete selected orbit.	Clicking button removes body from orbit queue. Effectively deletes the orbit.
0.8.4.0	Add method to display orbit object vector attributes, such as Velocity and Force. (Including option to hide these.)	Arrows are draw to the screen representing the direction and magnitude of the vectors.
0.8.5.0	Add method to interact with orbit body such that the camera locks to it and follows its position around the simulation.	Camera's centre
Version 0.9	**Data Storage**	
0.9.1.0	Write simulation data to storage.	Show representation of simulation data in storage system in console.
0.9.1.1	Read simulation data from storage.	Show read data from storage solution in console.
0.9.2.0	Add ability to create new simulations & read / write to them.	Show new simulation in storage.
0.9.2.1	Add input method to select simulations to read from / save to.	Use input method to save to and read from a simulation.

Documented design

Database design

Orbyte will need a basic database system so that it can store previous simulations. Other simulations do not do this, so it will be a novel feature. Persistence of simulation state is a valuable utility to implement, as it allows users to continue working on the same simulation across different sessions or utilise templates for simulations made by others.

Saved simulations can be used in a classroom situation where a template is opened by students, previously made by the teacher before the class started.

The database will be containing the fundamental data required to resume the simulation from where it was last left off. No encryption will be required as there will be no sensitive data stored.

Entity Relationship diagram (ERD)
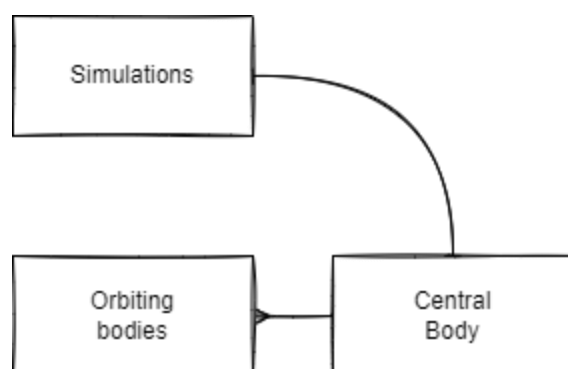
Entity Attribute Model (EAM)
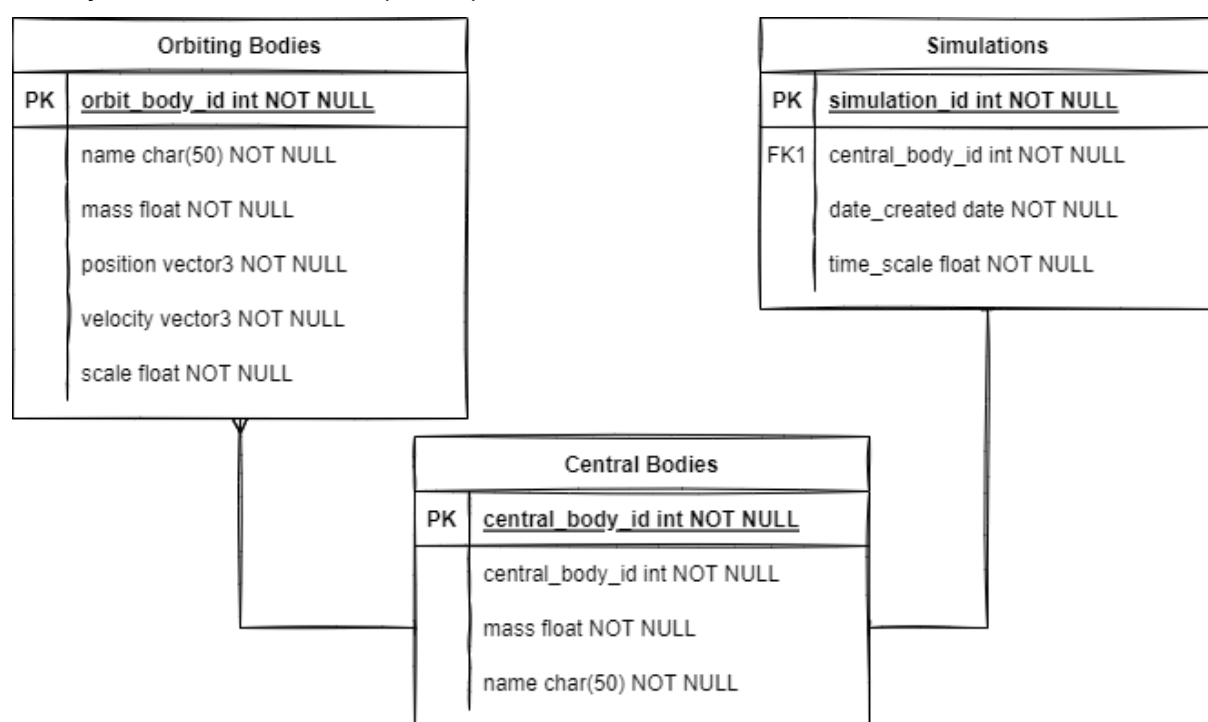
Overall System Design

UML Diagram

Central Body

Attributes:

- Mass: FLOAT
- Color: HEXADECIMAL
- Diameter: FLOAT
- Name: STRING

Methods:

- get_gconstant() -> FLOAT
- get_mass() -> FLOAT

Orbiting Body

Attributes:

- Mass: FLOAT
- Velocity: VECTOR
- Position: VECTOR
- Color: HEXADECIMAL
- Diameter: FLOAT
- Name: STRING
- CentralBody: CENTRAL_BODY

Methods:

- get_velocity() -> VECTOR
- get_position() -> VECTOR
- get_mass() -> FLOAT
- Update()

Satellite

Attributes:

- Mass: FLOAT
- Velocity: VECTOR
- Position: VECTOR
- Color: HEXADECIMAL
- Diameter: FLOAT
- Name: STRING
- Parent: ORBIT_BODY

Methods:

- get_velocity() -> VECTOR
- get_position() -> VECTOR
- get_mass() -> FLOAT
- Update()

Simulation

Attributes:

- Gravitational Constant: FLOAT
- Time Step: FLOAT

Methods:

- Update()
- Save()

Camera

Attributes:

- Position: VECTOR
- Clipping_z: FLOAT

Methods:

- WorldSpace_To_ScreenSpace() -> VECTOR

Application Process Diagram
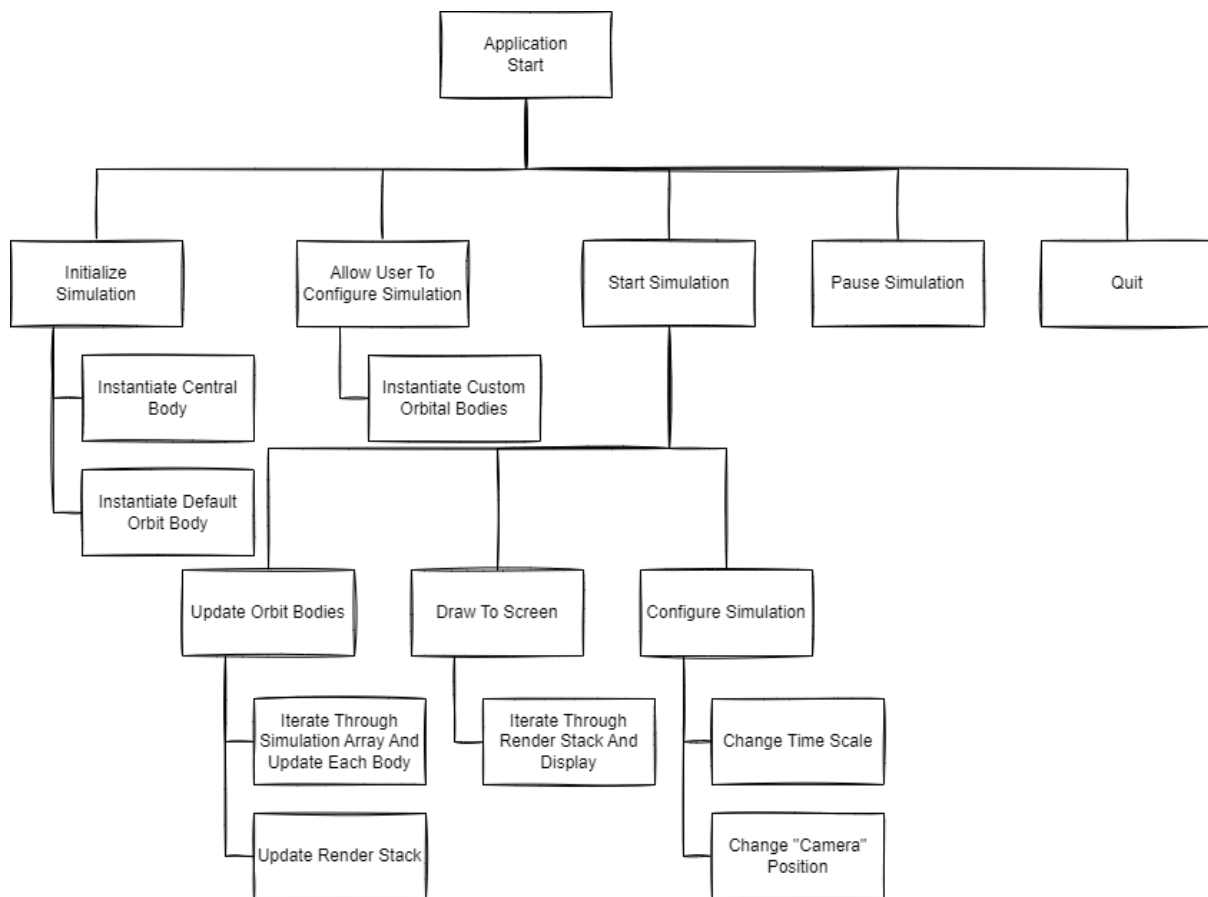
Figure 1: Process diagram detailing application flow

The above diagram shows how the user will interact with the application, featuring the typical steps taken when using the simulation. The user will create a new simulation, instantiating the central body and a default orbit body acting as an example. After configuration, starting the simulation will begin the mainloop for the simulation.

The programming methodology of the project will be Object-Oriented, this has been chosen due to the number of instantiated elements present in a simulation.

The user should be able to perform all of the above operations, and each of these should be implemented into the simulation as the most fundamental objectives to achieve what was outlined in the analysis.

Input, Process, Storage, Output (IPSO)

IPSO	Program section	Item
Input	Initializing Simulation	- ID (path) - Display Resolution - Central Object Mass - Central Object Scale
Input	Instantiating Orbiting Objects	- Mass

		-	Scale
		-	Initial Position
		-	Initial Velocity (*Can be calculated such that a circular orbit is produced*)
		-	Name
Input	Runtime Configuration	-	Change time-scale
		-	Camera Position / "zoom"
Process	Update Orbiting Bodies	-	Using RK4, calculate the next position of the body as a function of time.
		-	Calculate the force, acceleration, velocity (etc.) vectors to be drawn to the screen.
Process	Generate Vertices For Rendering	-	Calculate the new positions of the vertices defining the geometry of each orbiting body.
		-	Convert these 3D cartesian coordinates from "world" space to "screen" space so that they can be drawn to the screen.
Output	Draw Orbit Bodies	-	Draw the shapes representing the orbiting bodies to their respective positions.
Output	Display body information	-	Display the appropriate information pertaining to a selected orbiting body (if any).
Output	Draw attribute vectors	-	Represent body attributes (e.g. acceleration or velocity) as an arrow leading from the orbiting object.

Data dictionaries

Data Item	Data Type	Validation	Sample Data
Simulation ID	Int		1233052
Orbit Body ID	Int		4408723
Central Body ID	Int		5308921
Orbit Body Position	Vector3		(12000, 43000, 55000)
Orbit Body Velocity	Vector3		(1, 1, 1)
Initial Orbit Position	Vector3	X, y, z all floating point values and magnitude of position vector > 0.	(1000, 0, 0)
Orbit Body Mass	Double	Mass > 0 & Is Float	120000

Central Body Mass	Double	Mass > 0 & Is Float	26000000
Orbit Body Scale	Double	Scale > 0 & Is Float	1
Central Body Scale	Double	Scale > 0 & Is Float	1
Universal Time Scale	Double	100 > UTS > -100 & Is Float	10
Orbit Body Name	String	Is String	"Kevin"
Central Body Name	String	Is String	"Not Kevin"
Date Created	String		"12/11/2022"
Orbit Body Vertices	Vector3 Array		[(1, 1, 0), (1, 2, 0), (2, 2, 0)]
Orbit Body Edges	Vector Array		[(0, 1), (1, 2)]

Data structures

Vector 3

The simulation will be in 3 dimensions, its therefore necessary to implement a 3D vector structure in order to make calculations easier:

STRUCTURE Vector3
   Float x
   Float y
   Float z
END STRUCTURE

The above structure will also eventually contain various override methods to facilitate incorporation in mathematical equations. It will also need a "Debug" method that will return a string of its important attributes.

Edge

The Edge structure will contain two indices that refer to the position of a particular vertex in the object's vertex array. This way two vertices can be "Associated" and so a line can be drawn between them.

STRUCTURE Edge
   Integer a
   Integer b
END STRUCTURE

OrbitBodyData

This structure stores all the relevant information about an object in orbit such that it can be re-instantiated. This will be used for storing orbits in binary files as well as other debugging purposes.

STRUCTURE OrbitBodyData

String name
Vector3 center
Float scale
Vector3 velocity
Float mass
END STRUCTURE

OOP class design
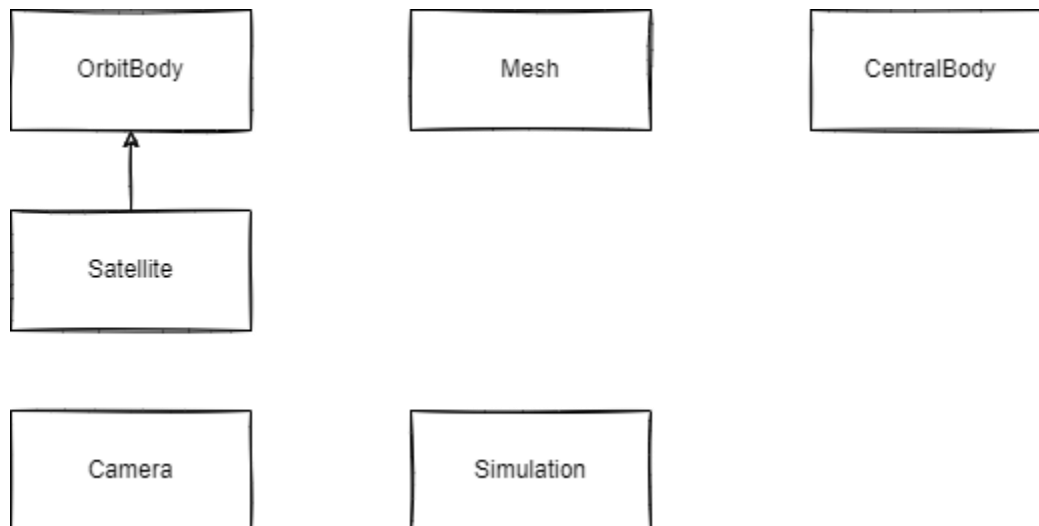
The above shows some of the classes that will be involved in the Technical Solution.

- **OrbitBody:** This object will be any planet or satellite within the orbit simulation. This class will contain all the relevant attributes necessary to calculate the position of the object as a function of time, data necessary for the display of the object and accessor methods for the above. This class will also contain a public method titled "Update" which can be called from the main-loop in order to update the object and calculate its new position.
- **Satellite:** This class will inherit from **OrbitBody**. It has one key distinction, being that it will have another attribute for an **OrbitBody** parent. It will also override the "Update" and "RK4" methods so that it can combine forces acting upon it from the **CentralBody** ("star") of the simulation and the **OrbitBody** parent.
- **Mesh:** This class[2] will contain vertex arrays and details pertaining to the lines connecting vertices. Every object that will be drawn by the renderer must have a mesh object.
- **CentralBody:** This is the object at the centre of the simulation. All objects orbit this body. It is static.
- **Camera:** This object handles conversion from world space to screen space. Data from the camera will be used in calculations relevant to pixel drawing.
- **Simulation:** The main class that will act as an encapsulating class, holding all the components within it. Instantiating a simulation will either involve creating a fresh one or using the parameters from an old version.

---

[2] This could ultimately be implemented as a structure.

Graphics

A crucial part of a simulation are the graphics that display the results of the physics calculations. A good representation of instantiated bodies and their orbits will be vital in creating an engaging and useful simulation.

For this project, I will be making my own graphics library called "Graphyte" that interacts directly with SDL in order to render pixels and images to the screen. Graphyte will display the results of the simulation, through 3D geometry and text, as well as forming the basis of user input through text fields and buttons.

Graphyte will handle the instantiation of GUI elements and their rendering by keeping them in a queue as a class attribute. Pixels to be drawn to the screen will be stored in a buffer with their color being determined by their position on the screen (forming a gradient color that I have grown fond of during prototyping).

Rendering 3D Objects

In order to represent geometry in 3D space, it is necessary to have a set of vertices and edges so that a wireframe representation of a 3D shape can be drawn. The shape chosen for planets in the project will be a 3D diamond with 6 vertices, while satellites will be a cube. The scale of the object (determining its dimensions) will be passed to a constructor.

When drawing the object, all points representing the geometry are converted from world space to screen space before being added to the pixel buffer within Graphyte. Accessed during a **draw** method, the pixel buffer is iterated through with each point being drawn to the screen.

User Interface

Graphyte will also have a number of classes for GUI, such as Text and Button. Text objects will have a string attribute containing the text they are to display, and buttons will contain pointers to a function to execute upon being clicked.

The UI will consist of a few key areas so that as much of the screen is preserved for viewing the simulation as possible. The general control panel for configuring the simulation, a panel in the top right corner for functions such as closing, saving or opening a simulation and an inspector in the bottom right corner that can be opened by clicking on an orbiting body. This inspector will serve as the main display of orbit information and will also contain input fields for different orbital parameters.
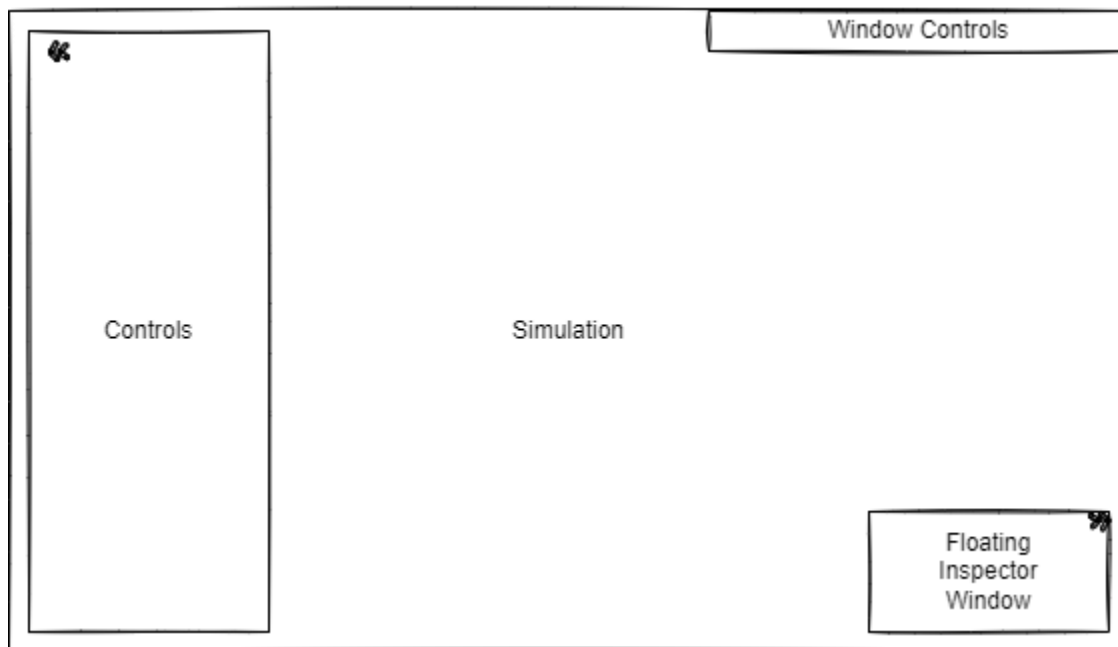
Figure 2: GUI Design

Items to be listed under controls:

- Performance Metrics (FPS, Vertices etc.)
- Clock (Displaying scaled time since start)
- Simulation Controls (Including central body mass, time scale and save destination)

Items in the orbit inspector:

- Name
- Orbit Period
- Mass
- Radius
- Velocity
- Acceleration
- *And accompanying input fields*

Buttons in controls:

- Save
- Reset
- Open
- Pause

Algorithms

Rotate

The purpose of this procedure is to rotate a set of vertices around the centre of the object (Centre of rotation). It takes 3 floating point parameters defining the rotation in the x, y and z axes. It accomplishes this through moving the points to the origin (by subtracting the centre position vector), applying a rotation through a 3D matrix transformation, then re-adding the centre's position vector to move the vertices to the correct distance from the centre. This method is duplicated within the Camera object in order to rotate orbits around the world origin.

```
PROCEDURE Rotate(Vector3 Rotation, Vector3 Point, Vector3 Centre)
      Point = Point – Centre

      float rad = 0
      float x, y, z
      rad = Rotation.x

      x = Point.x
      y = Point.y
      z = Point.z

      Point.y = (Cos(rad) * y) - (Sin(rad) * z)
      Point.z = (Sin(rad) * y) + (Cos(rad) * z)

      x = Point.x
      y = Point.y
      z = Point.z

      rad = Rotation.y
      Point.x = (Cos(rad) * x) + (Sin(rad) * z)
      Point.z = (-Sin(rad) * x) + (Cos(rad) * z)

      x = Point.x
      y = Point.y
      z = Point.z

      rad = Rotation.z
      Point.x = (Cos(rad) * x) - (Sin(rad) * y)
      Point.y = (Sin(rad) * x) + (Cos(rad) * y)

      Point = Point + Centre
ENDPROCEDURE
```

Runge-Kutta 4

The algorithm for the RK4 implementation is included in the Investigation section titled: "Runge-Kutta 4". It is repeated here for completeness.

```
Function ode(time [float], position [Vector3], velocity [Vector3])
          s = position
          a = -mu * r / (s.magnitude * s.magnitude * s.magnitude)
          return [velocity, a]
EndFunction
```

Function rk4_step(time [float], position [Vector3], velocity [Vector3], step_size [float])
        k1 = ode(time, position, velocity)
        *k2 = ode(time + 0.5 * step_size, position + 0.5 * k1 * step_size, velocity + 0.5 * k1 * step_size)*
        *k3 = ode(time + 0.5 * step_size, position + 0.5 * k2 * step_size, velocity + 0.5 * k1 * step_size)*
        *k4 = ode(time + step_size, position + k3 * step_size, velocity + k3 * step_size)*
        *return [position + step_size / 6.0 * (k1 + k2 * 2 + k3 * 3 + k4), velocity + step_size / 6.0 * (k1 + k2 * 2 + k3 * 3 + k4)]*
EndFunction

World space to camera space

In order to facilitate orbits in 3 dimensional space, it is necessary to distinguish between world space and screen space so that objects further away from the camera seem smaller. This is accomplished by dividing a point's x and y coordinates by their z coordinates (all relative to the camera) and then multiplying by the width and height of the screen so that the simulation fills the screen.

Function WorldSpaceToScreenSpace(vector3 world_position, float screen_height, float screen_width)
    Vector3 position = world_position – camera_position

    Vector3 screen_space_position
    *screen_space_position.x = (position.x / position.z) * screen_width*
    *screen_space_position.y = (position.y / position.z) * screen_height*
    screen_space_position.z = position.z

    Return screen_space_position
EndFunction

Line between two points

This is a crucial algorithm to be implemented, as it underpins most of the 3D geometry rendering in the project. The purpose of this procedure is to draw a line in screen-space between two points.

```
Procedure Line(Float x1, Float x2, Float y1, Float y2)
        Float dx = x2 − x1
        Float dy = y2 − y1
        Float length = square_root(dx * dx + dy * dy)
        Float angle = arctan(dy / dx)

        For( Int i in range(0, length))
                Pixel(x1 + cos(angle) * i, y1 + sin(angle) * i)
        EndFor
EndProcedure
```

Hashing Algorithm

This algorithm will be used for a hash table storing simulation data.

```
Function Hash(String name)
        String reverse = Reverse(name)
        String hash = Bitwise_String_XOR(name, reverse)
        Integer total = 0

        For(Integer i in Range(hash.length())
                total  = total + Integer(hash[i])
        EndFor

        Return (total % max_orbits)
EndFunction
```

The hashing algorithm XORs the name of the orbit with the reverse of itself, (XORing each bit in every character). The hash's character values in binary are then summed (mod max number of elements in storage so indexing wraps around). This simple hashing algorithm will be used with collision avoidance in order to handle simulation data storage.

Libraries

- **SDL:** The reasons surrounding the use of SDL2 [SDL_WIKI] has been outlined in the Objectives section of the Analysis. Fundamentally, access to low-level graphics functionality and I/O events will enable me to develop a performant, bespoke simulation.

Technical solution
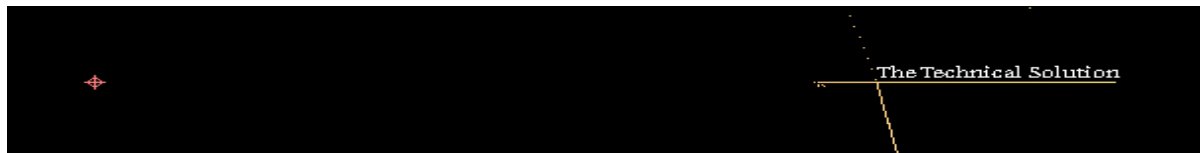
Orbyte is composed of 3 core systems: The Simulation, Graphyte (My custom graphics solution) and the Simulation Storage System. While the simulation handles all the realtime orbit simulation and user I/O, it interfaces with Graphyte in order to display GUI elements and render 3D planets and orbits. Once the user has finished configuring and observing their simulation, they may save it to a .orbyte binary file, so that they can open it another time exactly as they left it. Or students may be able to open pre-made simulations made by the teacher before their lesson.

Below is a UML Diagram providing an overview of the implementation's classes[3]. Each entity has a corresponding number for reference in the following table. Attributes and methods have been purposefully omitted from the diagram and included in a table for presentability.

---

[3] Note: Structures such as Mesh, Vector3, OrbitBodyData or SimulationData have not been included in this diagram. The table includes an objects key attributes & methods, important to its design and functionality, and excludes implementational variables and less vital procedures for conciseness. Pointers have been used frequently in the program, but will be listed as the data type of what they point to.
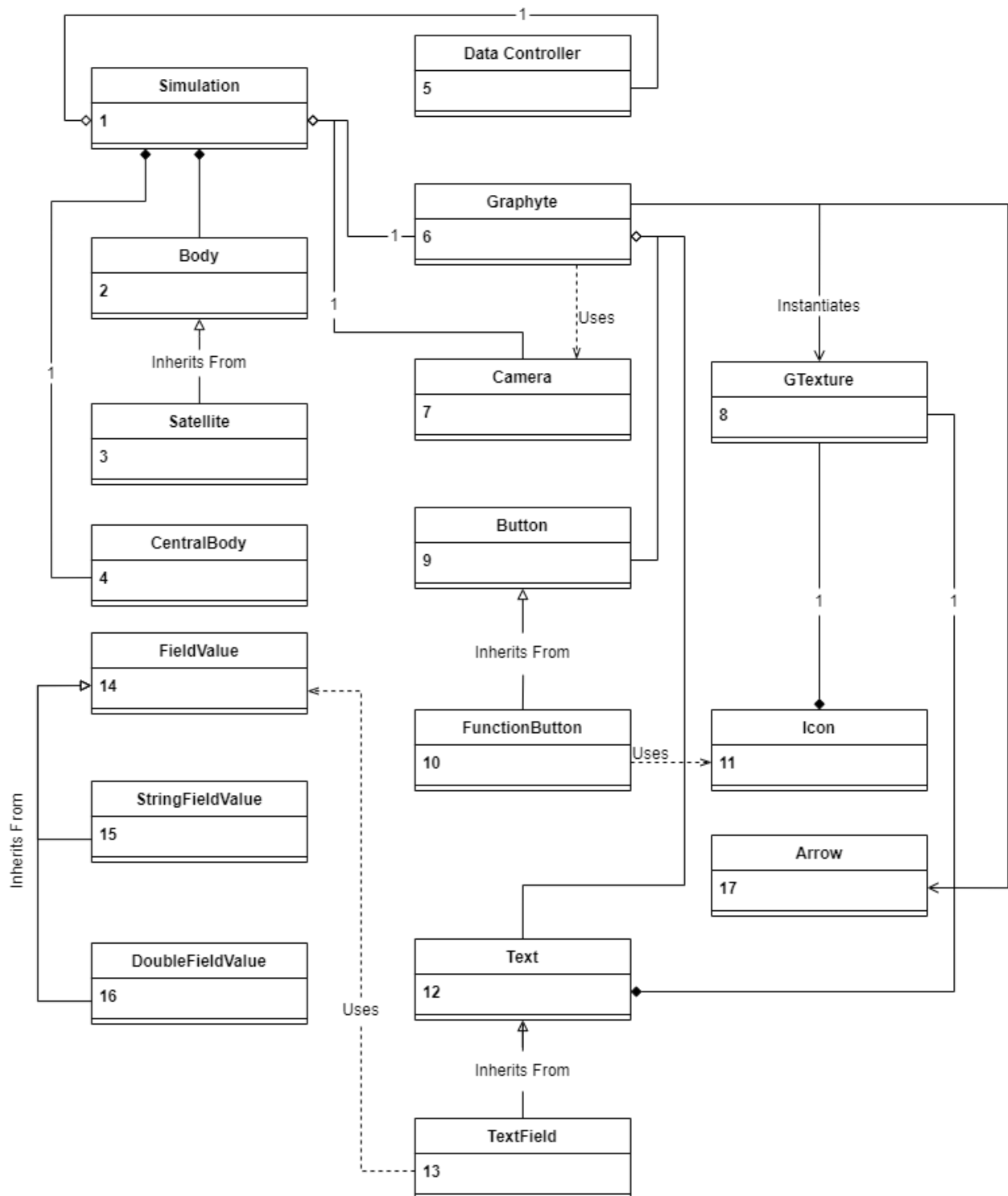
Figure 3: Technical Solution UML Diagram

Entity	Attributes	Methods
1: Simulation	- screen_width: DOUBLE - screen_height: DOUBLE - max_fps: INTEGER - time_scale: DOUBLE - gCamera: CAMERA - graphyte: GRAPHYTE - data_controller:	- init() - commit_to_text_field() - close_planet_inspectors() - click() - Update_Clock() - clean_orbit_queue() - add_specific_orbit()

	DATACONTROLLER - orbiting_bodies: VECTOR<BODY> - sun: CENTRALBODY - path_source: STRING	- add_orbit_body() - save() - open() + run()
2: Body	- satellites: VECTOR<SATELLITE> - graphyte: GRAPHYTE # mesh: MESH # trail_points: VECTOR<VECTOR3> # start_pos: VECTOR3 # start_vel: VECTOR3 # time_since_start: DOUBLE # position: VECTOR3 # radius: DOUBLE # velocity: VECTOR3 # angular_velocity: DOUBLE # acceleration: VECTOR3 # mu[4]: DOUBLE # mass: DOUBLE # scale: DOUBLE # gui: GUI_BLOCK + name: STRING	- Add_Satellite() - Create_Satellite() - Delete_Satellite() - Update_Satellites() - Draw_Satellites() # two_body_ode() # rk4_step() # Project_Circular_Orbit() # Generate_Vertices() # MoveToPos() # rotate() # CreateInspector() + Body() + free() + ShowBodyInspector() + HideBodyInspector() + GetOrbitBodyData() + DebugBody() + Reset() + Delete() + Update_Body() + Draw() + Draw_Arrows() + Calculate_Period()
3: Satellite (INHERITS FROM BODY)	- parentBody: BODY	- Generate_Vertices() **override** - Project_Circular_Orbit() **override** + Satellite() + Update_Body() **override**
4: Central Body[5]	- mesh: MESH + mass: DOUBLE + mu: DOUBLE + scale: DOUBLE + (**constant**) GravitationalConstant: DOUBLE + position: VECTOR3	- Generate_Vertices() + CentralBody() + Draw()
5: Data Controller		+ WriteDataToFile() + ReadDataFromFile()
6: Graphyte	- screen_width : DOUBLE - screen_height : DOUBLE	+ Init() + CreateText()

---

[4] "mu" is defined as the gravitational constant multiplied by the focus of the orbit: G x M_{large}

[5] This class does not inherit from Body because it is designed specifically to be static. As such it does not require any of the simulation methods or GUI and therefore it would not be consistent with OOP practice to derive it from Body. Any similarity between class attributes and methods is due to how graphics have been implemented and consistency with naming conventions, not a design oversight.

	- Renderer : SDL_RENDERER - Font : TTF_FONT - texts : VECTOR<TEXT> - icons : VECTOR<ICONS> - points : VECTOR<SDL_POINT> + active_text_field : TextField + text_fields : VECTOR<TEXTFIELD> + function_buttons : VECTOR<FUNCTIONBUTTON>	+ CreateIcon() + GetTextParams() + AddTextToRenderQueue() + AddIconToRenderQueue() + Get_Screen_Dimensions() + Get_Number_Of_Points() + pixel() + line() + draw() + free()
7: Camera	- camera_rotation : VECTOR3 + position : VECTOR3 + clipping_z : FLOAT	+ Camera() + RotateCamera() + rotate() + WorldSpaceToScreenSpace()
8: GTexture[6]	- MTexture : SDL_TEXTURE - renderer : SDL_RENDERER - font : TTF_FONT - mWidth : INTEGER - mHeight : INTEGER	+ GTexture() + GTexture(GTexture source)[7] + loadFromFile() + loadFromRenderedText() + reset_texture() + free() + render() + getWidth() + getHeight()
9: Button	- position : VECTOR3 - width : INTEGER - height : INTEGER - left_wall_offset : INTEGER - function : FUNCTION<VOID()> # enabled : BOOLEAN	# CallFunction() # AttachFunction() + Button() + SetDimensions() + SetPosition() + SetEnabled() + Clicked()
10: FunctionButton (INHERITS FROM BUTTON)	- icon : ICON	+ FunctionButton() + CheckForClick() + SetEnabled() + free()
11: Icon	- texture : GTEXTURE + pos_x : INTEGER + pos_y : INTEGER + path_to_image : STRING + visible : BOOLEAN + dimensions : VECTOR<INTEGER>	+ Icon() + Render() + SetPosition() + SetDimensions() + GetDimensions() + free()
12: Text	# texture : GTEXTURE + pos_x : INTEGER + pos_y : INTEGER + text : STRING	+ Text() + Text(Text t) + Set_Text() + Set_Position() + Set_Visibility()

---

[6] A "Texture" class is a way of encapsulating the rendering of more complex graphics. Images, fonts etc. would be loaded to a texture. Implementation heavily guided by this resource: https://lazyfoo.net/tutorials/SDL/ A series of tutorials regarding creating an application using SDL.
[7] Copy constructor used for debugging. Adds no functionality in the class but was crucial in fixing rendering bugs.

	+ visible : BOOLEAN	+ GetTexture() + GetPosition() + GetDimensions() + Render() + Debug() + free()
13: TextField (INHERITS FROM TEXT)	- text_color : SDL_COLOR - input_text : STRING - enabled : BOOLEAN - button : BUTTON - fvalue : FIELDVALUE	- Update_Text() - update_button_dimensions() - write_value() + TextField() + Set_Position() **override** + Set_Visibility() **override** + Backspace() + Add_Character() + CheckForClick() + Commit() + Enable() + Disable()
14: FieldValue		+ ReadField()
15: StringFieldValue	- value : STRING - read_f : FUNCTION<VOID()> - regex : STRING	- ValidateValue() + StringFieldValue() + ReadField() **override**
16: DoubleFieldValue	- value : DOUBLE - read_f : FUNCTION<VOID()>	- ValidateValue() + DoubleFieldValue() + ReadField() **override**
17: Arrow		+ Draw()

The Simulation

Orbyte consists of a realtime orbit simulation that has undergone constant refinement throughout the iterative development process. Within this simulation is a central body that is orbited by many orbiting bodies, and each of these bodies may also have many satellites. The user is able to dynamically instantiate and delete orbits during runtime, as well as configuring simulation parameters and seeing how they affect the orbits currently being simulated.

Input Handling

Input is handled by a large switch statement within the main update method of the application under the **Simulation** class.

```cpp
//Handle input events
while (SDL_PollEvent(&sdl_event) != 0)
{
    switch (sdl_event.type) //Switch on type of input
    {
    default:
        break;

    case SDL_QUIT: //Exit
        quit = true;
        break;

    case SDL_TEXTINPUT: //Typing
        printf("User is typing: %s\n", sdl_event.text.text);
        if (graphyte.active_text_field != NULL)
        {
            graphyte.active_text_field->Add_Character(sdl_event.text.text);
        }
        break;

    case SDL_KEYDOWN: //Keypress
        switch (sdl_event.key.keysym.sym)
        {
        case SDLK_BACKSPACE:
            printf("User Pressed the Backspace\n");
            if (graphyte.active_text_field != NULL)
            {
                graphyte.active_text_field->Backspace();
            }
            break;

        case SDLK_RETURN:
            commit_to_text_field();
            break;

        case SDLK_UP:
            //Rotate Up
            gCamera.RotateCamera({ 0.01, 0, 0 });
            break;

        case SDLK_DOWN:
            //Rotate Down
            gCamera.RotateCamera({ -0.01, 0, 0 });
            break;

        case SDLK_LEFT:
            //Rotate Left
            printf("\n\nPressed The Left Arrow Key\n");
            gCamera.RotateCamera({ 0, -0.01, 0 });
            break;

        case SDLK_RIGHT:
            //Rotate Right
            printf("\n\nPressed The Right Arrow Key\n");
            gCamera.RotateCamera({ 0, 0.01, 0 });
            break;
        }
        break;
```

```cpp
    case SDL_MOUSEBUTTONDOWN: //Mouseclick
        if (sdl_event.button.button == SDL_BUTTON_LEFT) //Left click
        {
            int mX = 0;
            int mY = 0;
            SDL_GetMouseState(&mX, &mY);
            click(mX - SCREEN_WIDTH / 2, -mY + SCREEN_HEIGHT / 2, true); //Adjust for screen dimensions
        }
        else if (sdl_event.button.button == SDL_BUTTON_RIGHT) //Right click
        {
            int mX = 0;
            int mY = 0;
            SDL_GetMouseState(&mX, &mY);
            click(mX - SCREEN_WIDTH / 2, -mY + SCREEN_HEIGHT / 2, false);
        }
        break;

    case SDL_MOUSEWHEEL:
        if (sdl_event.wheel.y > 0) //Scroll up
        {
            //Zoom in
            gCamera.position.z *= 1.4;
            std::cout << "Moved Camera to new pos: " << gCamera.position.z << "\n";
        }
        if (sdl_event.wheel.y < 0) //Scroll down
        {
            //zoom out
            gCamera.position.z /= 1.4;
            std::cout << "Moved Camera to new pos: " << gCamera.position.z << "\n";
        }
        break;
    }
}
```

Simulation Clock

For a physics simulation, it is necessary to control the rate of calculations made so that the progression of orbits can be simulated real-time or using a constant time-scale. This is accomplished by using the time since the last frame within the physics equations. Without this factor, simulations would execute at different speeds on different hardware. By utilizing a clock, consistent performance[8] across all hardware is ensured.

```
//Current time start time
Uint32 startTime = 0;
Uint32 deltaTime = 0; // delta time in milliseconds
```

Delta time is calculated from the time since application start and last frame time. This delta is then used in all simulation calculations. Delta time is stored in milliseconds as an unsigned integer.

```
Uint32 Update_Clock()
{
    Uint32 current_time = SDL_GetTicks(); //milliseconds
    Uint32 delta = current_time - startTime;
    startTime = current_time;
    return delta;
}
```

Delta Time is also used to pad out the duration of each frame so that the simulation runs at a maximum FPS. As default, the maximum frames per second is 500 (stored as a constant integer). Future versions of Orbyte could include a settings menu that may allow FPS to be capped at a user-defined limit for performance.

```
//DELAY UNTIL END
deltaTime = Update_Clock(); // get new delta
float interval = (float)1000 / MAX_FPS; // Intended interval (capped FPS)
if (deltaTime < (Uint32)interval) // If simulation is updating too quickly
{
    Uint32 delay = (Uint32)interval - deltaTime;
    if (delay > 0)
    {
        SDL_Delay(delay); // Delay to pad out frame duration and limit FPS
        deltaTime += delay;
    }
}
```

The clock is also displayed on the user interface as "days" since the simulation began.

---

[8] Performance and simulation optimisation is discussed further on.
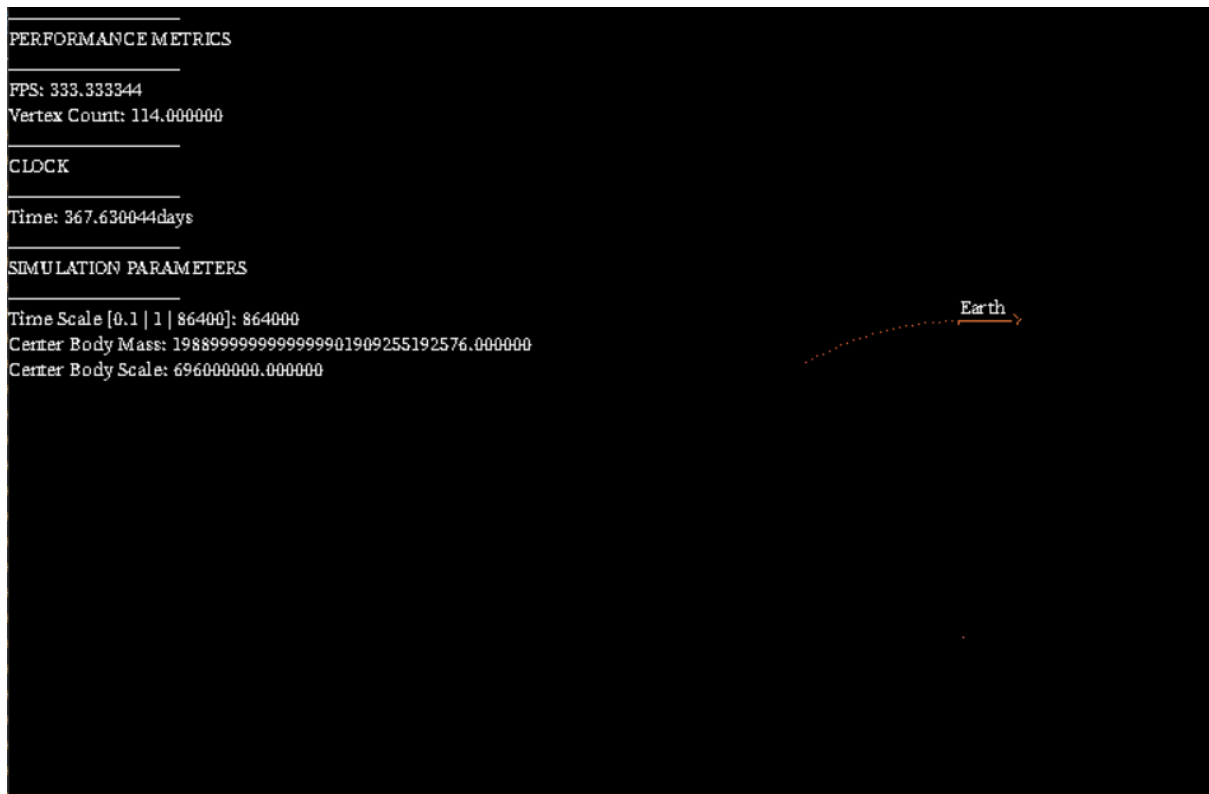
Figure 4 : Clock in Simulation GUI

Orbit Queue

The orbit "Queue" is implemented as a vector of pointers to Body objects. The original design planned to dequeue bodies sequentially in order to update and render them. However, due to the frequency of the necessary access to orbits in the queue and the need to iterate through a definite set of bodies, all the orbits simulated at any time in the application is stored in the "*orbiting_bodies*" vector[9].

```cpp
clean_orbit_queue(); // Check if any orbits in the vector are scheduled for deletion.

for (Body* b : orbiting_bodies)
{
    b->Update_Body(deltaTime, time_scale); // Update body
    b->Draw(graphyte, gCamera); // Draw the body
}
```

Whenever a new orbit is added to the simulation, it is "enqueued" or pushed to the back of the vector.

```cpp
// Add orbit with given OrbitBodyData
void add_specific_orbit(OrbitBodyData data)
{
    orbiting_bodies.push_back(new Body(data.name, data.center, data.mass, data.scale, data.velocity, Sun.mu, graphyte, false));
}

// Add general orbit with generic parameters
void add_orbit_body()
{
    orbiting_bodies.push_back(new Body("New Orbit", { 0, 5.8E10, 0 }, 3.285E23, 2.44E6, { 47000, 0, 0 }, Sun.mu, graphyte, false));
}
```

---

[9] Treated as a "queue" in so far as it operates as first in, first out. It is not necessary to access a specific element by index per sae, however by removing the need to re-enqueue objects every frame, significant performance has been preserved.

Orbit Body Class
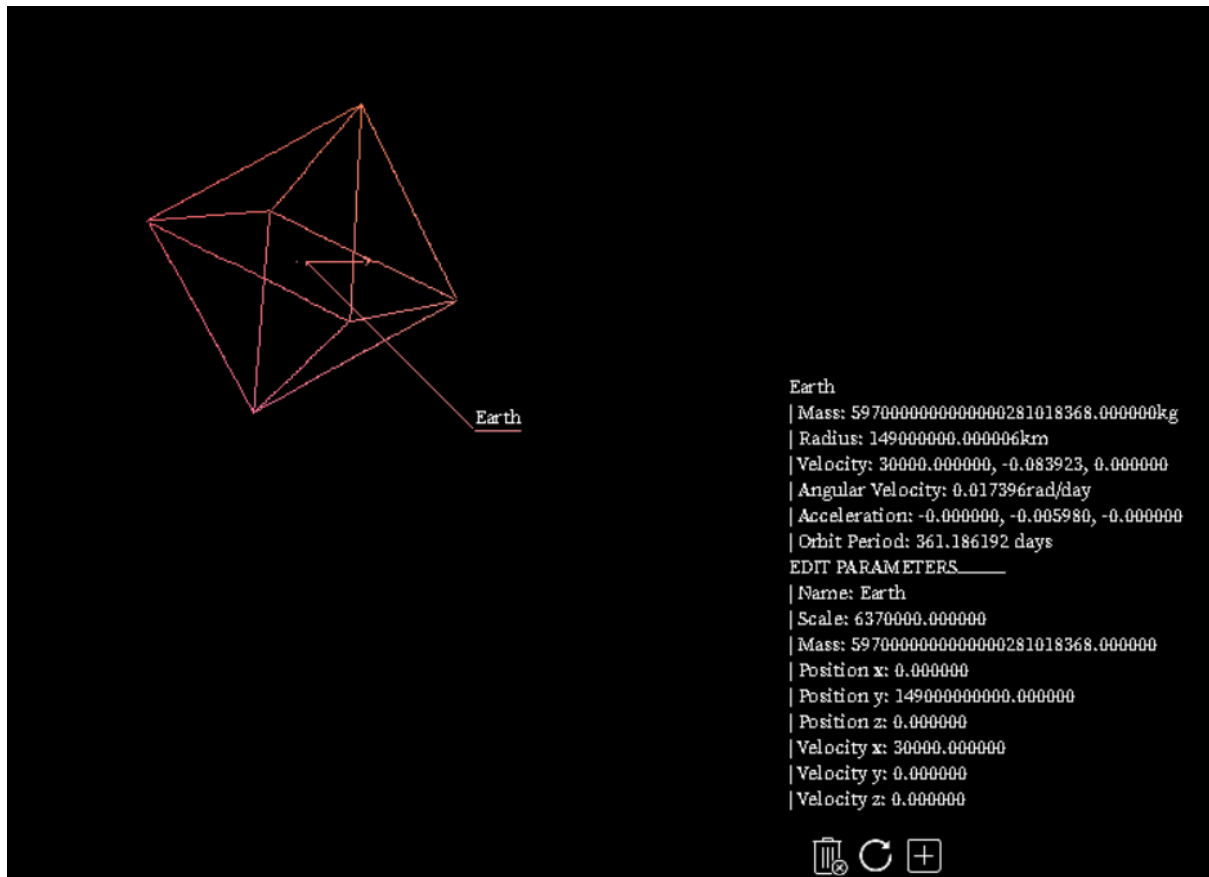
Figure 5 : Example orbit body with inspector enabled.

This class is instantiated to represent orbiting bodies. It contains a constructor that initialises all the GUI for the orbiting body, and other significant methods such as the Update method that is called every frame.

```
Body(std::string _name, vector3 _center, double _mass, double _scale, vector3 _velocity, double _mu, Graphyte& g, bool override_velocity = false):
    graphyte(g),
    ScaleFV(&scale, [this]() { this->RegenerateVertices(); }), MassFV(&mass), NameFV(&name, [this]() { this->Rename(); }), // Scale Dield Value. When written to, recalculate geometry
    PosXFV(&this->position.x, [this]() { this->RecenterBody(); }), PosYFV(&this->position.y, [this]() { this->RecenterBody(); }), PosZFV(&this->position.z, [this]() { this->RecenterBody(); }),
    VelXFV(&this->velocity.x, [this]() { this->SetStartVelocity(); }), VelYFV(&this->velocity.y, [this]() { this->SetStartVelocity(); }), VelZFV(&this->velocity.z, [this]() { this->SetStartVelocity(); })
{
    position = _center;
    start_pos = position;
    radius = Magnitude(position);

    name_label = g.CreateText(_name, 16); // Create floating text label that follows orbit body
    name_label->pos_x = 100; //Test values (overwritten later)
    name_label->pos_y = 100;

    mu = _mu; //Setting attributes
    name = _name;
    if (override_velocity)
    {
        Project_Circular_Orbit(_velocity); // Force a circular orbit. Not reccommended as will override given parameters. [NO LONGER SUPPORTED]
    }
    velocity = _velocity;
    start_vel = velocity;

    scale = _scale;
    mass = _mass;

    std::cout << "Instantiated Orbiting Body with initial position: " << start_pos.Debug() << " and velocity: " << velocity.Debug() << "\n";

    mesh.vertices = Generate_Vertices(scale); // Generate body geometry

    CreateInspector(g); // Create GUI for body
}
```

The Body class contains the methods for the Ordinary Differential Equation solver, instantiation of satellites and handles its own "inspector."

The Update Method

```cpp
virtual int Update_Body(float delta, float time_scale)
{
    if (time_scale == 0) // If paused, don't update.
    {
        return 0;
    }

    Update_Satellites(delta, time_scale); // Call Update Method of all child satellites

    rotate_about_centre({0.01, 0.01, 0.01}); // Gradual rotation about body origin to mimic a planet's rotation about its axis

    vector3 this_pos = position;
    float t = (delta / 1000); //time in seconds
    std::vector<vector3> sim_step = rk4_step(time_since_start, this_pos, velocity, t * time_scale); // Get RK4 result into a sim_step buffer.
    this_pos = sim_step[0];
    //if (position.z > 0) { std::cout << position.Debug() << "\n"; std::cout << velocity.Debug() << "\n"; }
    MoveToPos(this_pos); // Shift vertices to new position
    angular_velocity = Magnitude(velocity) / Magnitude(position); // angular velocity = tangential velocity / radius
    time_since_start += t * time_scale;

    // Add a "breadcrumb" or trail point if a certain distance away from last
    if (Magnitude(this_pos - last_trail_point) > (0.5 * radius) / 24)
    {
        trail_points.emplace_back(this_pos);
        last_trail_point = this_pos;
    }

    // Remove trail point to only show most recent
    if (trail_points.size() > 24)
    {
        trail_points.erase(trail_points.begin());
    }

    velocity = sim_step[1]; // Get result from RK4 buffer
    acceleration = sim_step[2];

    update_inspector(); // Update GUI

    return 0; // Successful update.
}
```

Taking the change in time since the last frame (clamped in the main .cpp file) and a time scale as its parameters, the update method governs the orbiting body's behaviour. It calls the private method **rk4_step** and **MoveToPos** most notably. These define the motion of the body frame-to-frame.

The RK4 Step

```cpp
std::vector<vector3> rk4_step(float _time, vector3 _position, vector3 _velocity, float _dt = 1)
{
    //structure of the vectors: [pos, velocity]
    std::vector<vector3> rk1 = two_body_ode(_time, _position, _velocity);
    std::vector<vector3> rk2 = two_body_ode(_time + (0.5 * _dt), _position + (rk1[0] * 0.5f * _dt), _velocity + (rk1[1] * 0.5f * _dt));
    std::vector<vector3> rk3 = two_body_ode(_time + (0.5 * _dt), _position + (rk2[0] * 0.5f * _dt), _velocity + (rk2[1] * 0.5f * _dt));
    std::vector<vector3> rk4 = two_body_ode(_time + _dt, _position + (rk3[0] * _dt), _velocity + (rk3[1] * _dt));

    vector3 result_pos = _position + (rk1[0] + (rk2[0] * 2.0f) + (rk3[0] * 2.0f) + rk4[0]) * (_dt / 6.0f);
    vector3 result_vel = _velocity + (rk1[1] + rk2[1] * 2 + rk3[1] * 2 + rk4[1]) * (_dt / 6);
    return { result_pos, result_vel, rk1[1]};
}
```

As explained previously in this document, the RK4 step takes **time, position, velocity and delta time** as parameters and uses them to calculate the orbit body's new position and velocity. Each rk step is calculated using the two_body_ode method.

Solving Ordinary Differential equations

```cpp
std::vector<vector3> two_body_ode(float t, vector3 _r, vector3 _v)
{
    vector3 r = _r; //displacement
    vector3 v = _v; //velocity
    //std::cout << "R.z" << r.z << "\n";
    vector3 nr = Normalize(r);
    //std::cout << "NR.z" << nr.z << "\n";
    if (nr.z == 0) { nr.z = 1; r.z = 0; }
    if (nr.y == 0) { nr.y = 1; r.y = 0; }
    if (nr.x == 0) { nr.x = 1; r.x = 0; }
    double mag = Magnitude(r);
    vector3 a = {
        (-mu * r.x) / (pow(mag, 3)),
        (-mu * r.y) / (pow(mag, 3)),
        (-mu * r.z) / (pow(mag, 3))
    };
    //std::cout << "rk_result: " << (pow(nr.z, 3)) << "\n";
    return { v, a };
}
```

This method takes a position and velocity and returns a velocity and acceleration. This underpins the physics simulation involved in the project and is this method is common amongst all orbiting bodies, including satellites.

Calculating Orbit Period

In order to calculate the orbit period of a body, the radius of the orbit and current relative velocity (to the object it is orbiting) is used.

```cpp
/// <summary>
/// The amount of time in seconds for the body to complete 1 orbit
/// </summary>
/// <returns>Time period</returns>
virtual double Calculate_Period()
{
    double length_of_orbit = 2 * 3.14159265359 * radius; //// 2 * pi * R
    double t = length_of_orbit / Magnitude(velocity);
    return t;
}
```

Orbit Body Geometry

```cpp
virtual std::vector<vector3> Generate_Vertices(double scale)
{
    std::vector<vector3> _vertices{
        {1, 0, 0},
        {-1, 0, 0},

        {0, 1, 0}, //2
        {0, -1, 0},

        {0, 0, 1}, //4
        {0, 0, -1}
    };

    for (auto& v : _vertices)
    {
        v.x *= scale;
        v.x += position.x;
        v.y *= scale;
        v.y += position.y;
        v.z *= scale;
        v.z += position.z;
    }

    //Edges Now
    std::vector<edge> _edges{
        {0, 3},
        {0, 2},
        {0, 4},
        {0,5},

        {1, 2},
        {1,3},
        {1,4},
        {1,5},

        {2,4},
        {2,5},
        {3,4},
        {3,5}
    };
    mesh.edges = _edges;

    printf("\n Generated vertices");

    return _vertices;
}
```

In order to represent a 3D object, information such as number and location of vertices and edges are necessary. That information is generated in this method and will be used in Graphyte to render the body.

Orbit Body Geometry
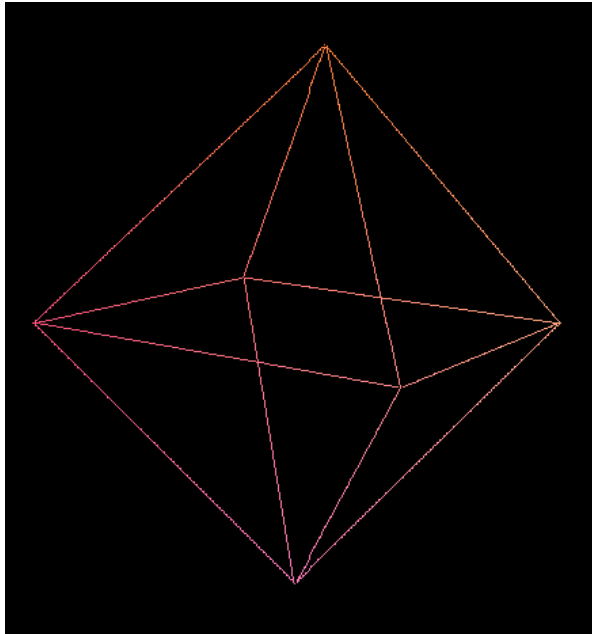
Figure 6: Orbit Body Geometry

The image (left) shows the geometry generated from the data created with this function and drawn with Graphyte.

Future versions of Orbyte could build upon the rudimentary geometry of bodies by facilitating the importing of mesh data from 3D object files (such as .fbx files).

Rotate

```cpp
void rotate(float rot_x = 1, float rot_y = 1, float rot_z = 1)
{
    for (auto& p : vertices)
    {
        vector3 point = p;
        //centroid adjustments
        point = point - position;

        //Rotate point
        float rad = 0;

        rad = rot_x;
        point.y = std::cos(rad) * point.y - std::sin(rad) * point.z;
        point.z = std::sin(rad) * point.y + std::cos(rad) * point.z;

        rad = rot_y;
        point.x = std::cos(rad) * point.x + std::sin(rad) * point.z;
        point.z = -std::sin(rad) * point.x + std::cos(rad) * point.z;

        rad = rot_z;
        point.x = std::cos(rad) * point.x - std::sin(rad) * point.y;
        point.y = std::sin(rad) * point.x + std::cos(rad) * point.y;

        //centroid adjustments
        point = point + position;

        p = point;
    }
}
```

In order to rotate objects in 3D space, a matrix transformation must be applied to each vertex that defines the geometry. The rotate method accomplishes this. This is a protected method that shifts the geometry to the origin (via a centroid adjustment) and rotates the shape by a given number of radians before shifting it back.

Accessor Methods

Following object-oriented programming conventions, numerous accessor methods have been implemented in this class and others in the project so that private and protected variables can be read.

```cpp
std::vector<vector3> Get_Vertices()
{
    //Return vertices
    return vertices;
}

std::vector<edge> Get_Edges()
{
    return edges;
}

std::vector<vector3> Get_Trail_Points()
{
    return trail_points;
}

vector3 Get_Tangential_Velocity()
{
    return velocity;
}

vector3 Get_Position()
{
    return position;
}
```

Satellites

Satellite inherits from **Body** and overrides several important methods.

```cpp
class Satellite : public Body
{
private:
    Body* parentBody; // Pointer to parent, e.g. Moon -> Earth
    //Satellites have different geometry! Cube.
    std::vector<vector3> Generate_Vertices(double scale) override { ... }
    //Override Circular Orbit Projection [NO LONGER SUPPORTED]
    void Project_Circular_Orbit(vector3& _velocity) override { ... }

protected:
    //Override Inspector Values
    void update_inspector() override { ... }
    //Override Period Calculation
    double Calculate_Period() override { ... }

public:
    //Constructor
    Satellite(std::string _name, Body* _parentBody, vector3 center, double _mass, double _scale, vector3 _velocity, Graphyte& g, bool override_velocity = false)
    //Override Update
    int Update_Body(float delta, float time_scale, std::vector<Body*>* bodies_in_system) override { ... }
};
```

They are instantiated by a "parent" **Body** via a user interaction with a **FunctionButton** accessible in the Body's inspector.

```cpp
inspector_satellite = new FunctionButton([this]() { this->Create_Satellite(); }, {(screen_dimensions.x / 2) - 215, -(screen_dimensions.y / 2) + 30, 0}, {25, 25, 0}, g, "icons/add.png");
g.function_buttons.emplace_back(inspector_satellite);
```

```
Earth
| Mass: 5970000000000000281018368.000000kg
| Radius: 149000000.000004km
| Velocity: 30000.000000, -0.061721, 0.000000
| Angular Velocity: 0.017396rad/day
| Acceleration: -0.000000, -0.005980, -0.000000
| Orbit Period: 363.015589 days
EDIT PARAMETERS_____
| Name: Earth
| Scale: 6370000.000000
| Mass: 5970000000000000281018368.000000
| Position x: 0.000000
| Position y: 149000000000.000000
| Position z: 0.000000
| Velocity x: 30000.000000
| Velocity y: 0.000000
| Velocity z: 0.000000
```

Figure 7 (LEFT) : Orbit Inspector with circled "add satellite" button

The image (left) shows a first-generation orbit body's inspector with the add-satellite button circled.

Polymorphism

Satellite inherits from the Body class and directly overrides several virtual methods inherited from the parent class. This, while fundamentally preserving the functionality of the orbit body, changes how: the geometry is generated, inspector is updated, period is calculated and slightly changes how the body is updated.

The most significant visual difference between the 1st generation[10] orbits and 2nd generation orbits are the geometries generated in the virtual **Generate_Vertices** method.

```cpp
//Satellites have different geometry! Cube.
std::vector<vector3> Generate_Vertices(double scale) override {
    //Polymorphism!
    std::vector<vector3> _vertices{
        {0, 1, 1},
        {1, 0, 1},
        {0, -1, 1},
        {-1, 0, 1},

        {0, 1, -1},
        {1, 0, -1},
        {0, -1, -1},
        {-1, 0, -1},
    };

    for (auto& v : _vertices)
    {
        v.x *= scale;
        v.x += position.x;
        v.y *= scale;
        v.y += position.y;
        v.z *= scale;
        v.z += position.z;
    }

    //Edges Now
    std::vector<edge> _edges{
        {0,1},
        {0, 3},
        {2, 1},
        {2, 3},

        {4, 5},
        {4, 7},
        {6, 5},
        {6, 7},

        {0, 4},
        {1, 5},
        {2, 6},
        {3, 7}
    };
    mesh.edges = _edges;

    return _vertices;
};
```

The above method generates the vertices and edges forming a cube. The differences between the parent and child class can be seen below:

---

[10] Remembering that "generation" refers to the hierarchical nature of orbit instantiation. 1st gen refers to objects directly orbiting the centre body. 2nd gen, the satellites, 3rd gen the satellites of satellites etc.

Figure 8 : 1st (top) and 2nd (bottom) gen orbit geometries (NOT TO SCALE)
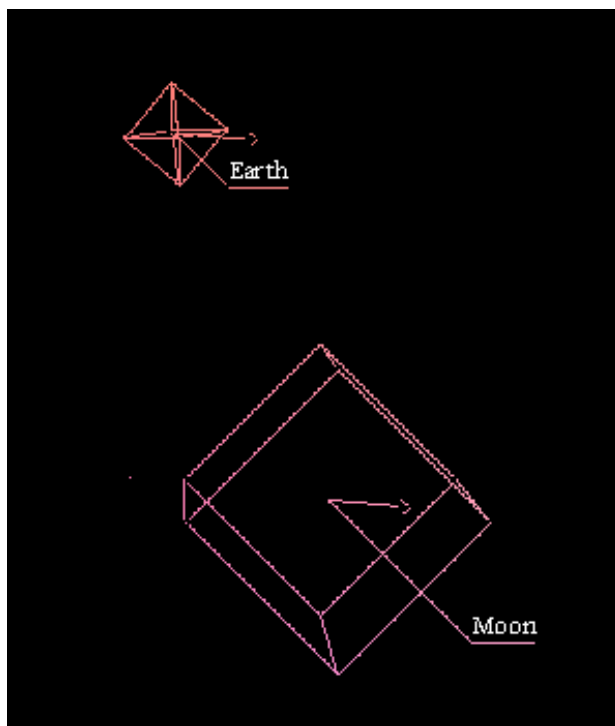
The other significant override within the child class **Satellite** is the **Calculate_Period** method:

```
//Override Period Calculation
double Calculate_Period() override
{
    //double T = 2 * 3.14159265359 * sqrt((pow(radius, 3) / mu));
    double length_of_orbit = 2 * 3.14159265359 * (Magnitude(parentBody->Get_Position() - position)); //Relative Position
    double t = length_of_orbit / Magnitude(velocity - parentBody->Get_Tangential_Velocity());
    return t;
}
```

Within this method, it is necessary to calculate the **length_of_orbit** with the relative position of the satellite to the **parent_body**. This ensures that the calculated orbit period is for the satellite around the parent body instead of around the centre body.

Vector3

A vital structure composed of 3 doubles: x, y and z. This struct overloads many standard c++ arithmetic operators to implement vector addition, subtraction and scalar / vector multiplication. Methods including Normalize, Distance, Scalar Product and Magnitude are also part of the header file.

```cpp
struct vector3
{
    double x, y, z;

    vector3 operator*(double right)
    {
        vector3 result = { x * right, y * right, z * right };
        return result;
    }

    vector3 operator+(double right)
    {
        vector3 result = { x + right, y + right, z + right };
        return result;
    }

    vector3 operator-(double right)
    {
        vector3 result = { x - right, y - right, z - right };
        return result;
    }

    vector3 operator+(vector3 v)
    {
        vector3 result = { x + v.x, y + v.y, z + v.z };
        return result;
    }

    vector3 operator-(vector3 v)
    {
        vector3 result = { x - v.x, y - v.y, z - v.z };
        return result;
    }

    double operator*(vector3 v)
    {
        double result = (x*v.x) + (y*v.y) + (z*v.z);
        return result;
    }

    vector3* operator=(const vector3& v)
    {
        x = v.x;
        y = v.y;
        z = v.z;
        return this;
    }
}
```

A debug method also converts the vector3 data to a more readable string format:

```cpp
std::string Debug()
{
    return std::to_string(x) + ", " + std::to_string(y) + ", " + std::to_string(z);
}
```

The "vec3" header file contains 4 useful methods, used in a variety of circumstances throughout the program.

```cpp
// Length of a vector3. Thank you Pythagoras.
double Magnitude(vector3 vec)
{
    return sqrt((vec.x * vec.x) + (vec.y * vec.y) + (vec.z * vec.z));
}

// Distance between two points in 3D space
double Distance(vector3 vecFrom, vector3 vecTo)
{
    vector3 a = vecTo - vecFrom;
    double distance = Magnitude(a);

    return distance;
}

// "Dot Product" of two vectors.
double Scalar_Product(vector3 a, vector3 b)
{
    return (a.x * b.x) + (a.y * b.y) + (a.z * b.z);
}

// Vector3 in same direction but with magnitude 1
vector3 Normalize(vector3 vec)
{
    double mag = Magnitude(vec);
    vector3 norm = {
        vec.x / mag,
        vec.y / mag,
        vec.z / mag
    };
    return norm;
}
```

Graphics

Orbyte is an educational simulation. It's bespoke nature warrants more specialized graphics and implementation choices to maximise performance so that the greatest number of orbiting entities can be simulated. For this reason, instead of using a pre-existing graphics library, I decided to write my own graphics library built exclusively for Orbyte on the foundations laid by SDL2. This permits lower-level access to pixel and texture rendering than otherwise possible, giving me greater control over its performance, behaviour and features throughout development.

Graphyte

The graphics library is called Graphyte and it oversees everything from rendering orbiting objects to drawing True-Type-Fonts and handling GUI input. [**WIP**]

```cpp
//Draw everything to the screen. Called AFTER all points added to the render queue
void draw()
{
    SDL_SetRenderDrawColor(Renderer, 0, 0, 0, 255);
    SDL_RenderClear(Renderer);
    int count = 0;

    //pixel(100, 100);

    for (auto& point : points)
    {
        SDL_SetRenderDrawColor(Renderer, 255, ((float)point.x / (float)SCREEN_WIDTH) * 255, ((float)point.y / (float)SCREEN_HEIGHT) * 255, 255);
        SDL_RenderDrawPoint(Renderer, point.x, point.y);
        count++;
    }

    //Make sure you render GUI!
    for (Text* t : texts)
    {
        t->Render({ SCREEN_WIDTH, SCREEN_HEIGHT, 0 });
    }

    for (Icon* i : icons)
    {
        i->Render({ SCREEN_WIDTH, SCREEN_HEIGHT, 0 });

    }

    SDL_RenderPresent(Renderer);
    points.clear();
}
```

The draw method iterates through the points, texts and icons present in the simulation and renders them all to the screen via their respective methods. These methods are accessed via a reference to a particular point and pointers to texts and icons.

Textures

Heavily influenced by "Lazy Foo' Productions SDL2 Game Programming tutorial" [LAZY_FOO], the texture class is used for loading fonts and rendering text to the screen, as well as displaying icons loaded from bitmaps. Stages of development yielded some difficulties handling textures. Some issues included textures instantiated by constructors in other classes falling out of scope and hence calling the destructor of the texture class. This has been remedied through better programming style and the addition of a copy constructor in the texture class that served as a debugging notifier.

```
class GTexture
{
private:
    //The actual hardware texture
    SDL_Texture* mTexture;

    //The renderer
    SDL_Renderer* renderer;

    //The font
    TTF_Font* font;

    //Image dimensions
    int mWidth;
    int mHeight;

public:
    //Constructor
    GTexture(SDL_Renderer* _renderer = NULL, TTF_Font* _font = NULL){ ... }

    GTexture(const GTexture& source){ ... }

    //Deallocates memory
    ~GTexture(){ ... }

    //Loads image at specified path
    bool loadFromFile(std::string path){ ... }

    //Creates image from font string
    bool loadFromRenderedText(std::string textureText, SDL_Color textColor){ ... }

    void reset_texture(){ ... }

    //Deallocates texture
    void free(){ ... }

    //Renders texture at given point
    void render(int x, int y, int override_width = NULL, int override_height = NULL, SDL_Rect* clip = NULL, double angle = 0.0, SDL_Point* center = NULL, SDL_RendererFlip flip = SDL_FLIP_NONE){ ... }

    //Accessor Methods for retrieving dimensions
    int getWidth(){ ... }

    int getHeight(){ ... }
};
```

Text

The text class primarily consists of a texture reference and string to be displayed. The texture is loaded from a true type font file with the characters passed as an argument to the "loadFromRenderedText" function. This class contains get & set methods for the text, as well as attributes and accessor / mutator methods pertaining to the position and dimensions of the text on screen.

Arrow

A crucial[11] part of the GUI includes arrows to represent vectors in 3D space, most prominently the velocity and acceleration of objects in orbit.

```
void Draw(vector3 position, vector3 direction, double magnitude, int heads, Graphyte& graphyte)
{
    //These are all 2D vectors.
    vector3 start = position;
    vector3 end = position + (direction * magnitude);
    vector3 perp_dir = vector3{ direction.y, -direction.x, 0 };
    double arrow_head_size = magnitude / 10;

    for (int i = 0; i < heads; i++)
    {
        vector3 ah1 = end + (direction * arrow_head_size);
        vector3 ah2 = end + (perp_dir * arrow_head_size);
        vector3 ah3 = end + (perp_dir * -arrow_head_size);

        graphyte.line(start.x, start.y, end.x, end.y);
        graphyte.line(ah1.x, ah1.y, ah2.x, ah2.y);
        /*graphyte.line(ah2.x, ah2.y, ah3.x, ah3.y);*/
        graphyte.line(ah3.x, ah3.y, ah1.x, ah1.y);

        end = ah1;
    }
}
```

This method takes the initial position, normalized direction and magnitude as parameters, as well as number of heads on the arrow (convention suggests 2 heads to denote acceleration). Vector math and vertex placement then forms the arrow.

---

[11] Arrows were also used extensively in debugging. They proved extremely useful when determining the cause of peculiar satellite behaviour by configuring an arrow to point towards the centre of the object it was *supposed* to be orbiting.

Button

Buttons underpin any GUI functionality in any application, and in Orbyte they will be used to enable user interaction, such as adding new orbiting planets and editing text fields. Buttons have a function pointer attribute for when they are clicked, which can be set at bound and rebound dynamically rather than at buttons instantiation.

```cpp
bool Clicked(int x, int y)
{
    if (!enabled)
    {
        //Disabled button should not be clicked!
        return false;
    }

    // (0<AM.AB<AB.AB) ^ (0<AM.AD<AD.AD) Where M is a point we're checking
    // Find each vertex using left_wall_offset, position and height of the button.
    vector3 A = { position.x - left_wall_offset, position.y + height / 2, 0 };
    vector3 B = { position.x - left_wall_offset + width, position.y + height / 2, 0 };
    vector3 D = { position.x - left_wall_offset, position.y - height / 2, 0 };
    vector3 C = { position.x - left_wall_offset + width, position.y - height / 2, 0 };

    // M is the location of the click
    vector3 M = { x, y, 0 };

    vector3 AM = M - A;
    vector3 AB = B - A;
    vector3 BC = C - B;
    vector3 BM = M - A;

    bool in_area = 0 <= AB*AM && AB*AM <= AB*AB && 0 <= BC*BM && BC*BM <= BC*BC;
    // Return if click in button bounds.
    return in_area;
}
```

2D vector calculations determine whether any given point in the xy plane is within the bounds of the button as defined by the button's dimensions. This process involved the scalar product of vectors.

Function Buttons

Inheriting from Button, function buttons are used with icons in order to call a specific function assigned at instantiation.

```cpp
FunctionButton(std::function<void()> f, vector3 pos, vector3 dimensions, Graphyte& g, std::string path_to_icon) : Button(pos, dimensions)
{
    std::cout << "\n Instantiated a function button. Method present?: " << (bool)f << "\n";
    if (path_to_icon != "") //Never accessed by user therefor no need for complex regular expressions or validation.
    {
        icon = g.CreateIcon(path_to_icon, {(int)dimensions.x, (int)dimensions.y, 0});
        icon->SetPosition(pos);
    }
    AttachFunction(f);
}
```

Regular buttons do not have an icon by default.

```cpp
void SetEnabled(bool _enabled) override // Set both button and icon to be enabled or disabled
{
    Button::SetEnabled(_enabled);
    if (icon)
    {
        icon->visible = _enabled;
    }
}
```

FunctionButtons override the SetEnabled method so that their icon can be enabled or disabled as well.

Field Value

Input fields need to have a pointer to the variable their new contents should change. E.g. if an edition to the time scale field is made, the time_scale variable should be changed. This is accomplished through the Field Value class, inheritance and polymorphism through overriding methods (See Double Field Value or String Field Value).

```cpp
class FieldValue
{
public:
    virtual void ReadField(std::string content) = 0;
};
```

Double Field Value

Inherits from field value and overrides the ReadField method to include its own validation and write to its double pointer attribute.

```cpp
void ReadField(std::string content) override
{
    if (ValidateValue(content))
    {
        std::cout << "\n Valid field content";
        if (value != NULL)
        {
            double new_value = atof(content.c_str());
            if (new_value != *value)
            {
                *value = new_value;
                std::cout << "\n Successfully wrote to value from input field!  \n" << new_value;

                if (read_f != NULL)
                {
                    read_f();
                }
            }
        }
    }
}
```

The ReadField method calls ValidateValue which is a private function returning a Boolean if the value is valid.

```cpp
private:
    double* value = NULL; // This is the pointer to the variable the input field is associated with. E.g. time step or object mass
    std::function<void()> read_f = NULL; // Function to call if successfully read

    bool ValidateValue(std::string content)
    {
        try
        {
            std::regex dbl_regex("(-+)?([0-9]+(\.[0-9]+)?)(E[0-9]+)?"); // Regex
            if (std::regex_match(content, dbl_regex))
            {
                // Success
                double test_validity = atof(content.c_str());
                std::cout << content << "=>" << test_validity;
                return true;
            }
            else {
                throw(content); // Issue
            }
        }
        catch (std::string bad)
        {
            std::cout << "\n Bad input recieved: " << bad; // Its bad
            return false; // Failure
        }
    }
```

ValidateValue initially uses a regex expression to ensure the content string is a valid signed or unsigned number before attempting to convert it to a double within a try-catch statement. Both **Double Field Value** and **String Field Value** have pointers to functions (defaulted to NULL) that can be set so that a function is called whenever the value of the field is changed successfully.

```cpp
void ReadField(std::string content) override
{
    if (ValidateValue(content)) // Check if valid
    {
        std::cout << "\n Valid field content"; // Passed validation
        if (value != NULL) // If pointer to value to write to is not null
        {
            double new_value = atof(content.c_str()); // Conversion
            if (new_value != *value) // Check for redundant set
            {
                *value = new_value;
                std::cout << "\n Successfully wrote to value from input field!  \n" << new_value; // Debug

                if (read_f != NULL)
                {
                    read_f(); // Call attached method if it exists.
                }
            }
        }
    }
}
```

This functionality is vital as it allows editions made to GUI input fields to call appropriate methods. These methods range from recalculating the geometry of an orbit body given a new parameter to recalculating simulation variables for all orbits.

```cpp
DoubleFieldValue CentreMassFV(&Sun.mass, [this]() { this->recalculate_center_body_mu(); });
```

String Field Value

Inherits from field value and overrides the ReadField method to include its own validation and write to its string pointer attribute. Contains a regular expression that can be customized upon instantiation.

```cpp
StringFieldValue(std::string* write_to, std::function<void()> f = NULL, std::string _regex = "(([A-Z]|[a-z])|[ ])+")
{
    value = write_to;
    regex = _regex;
    if (f)
    {
        read_f = f;
    }
}
```

This ability to override the default regular expression proves especially useful for the "path input field" where the user can enter the path to a .orbyte file. Used to validate input before searching for the file.

```cpp
StringFieldValue path_to_file(&path_source, NULL, "(([A-Z]|[a-z]|(_))|[ ])+\.orbyte"); //Custom regex
```

Below shows the validation method.

```cpp
bool ValidateValue(std::string content)
{
    try
    {
        std::regex dbl_regex(regex); // Create regex object from string defined in constructor.
        if (std::regex_match(content, dbl_regex))
        {
            return true; // Success
        }
        else {
            throw(content); // Content has failed the validation => Catch
        }
    }
    catch (std::string bad)
    {
        std::cout << "\n Bad input recieved: " << bad; // Its bad
        return false; // Failure
    }
}
```

Option attached method functionality is identical to that specified in **Double Field Value**.

Text Field

This class forms a critical part of the user interface as it allows users to edit parameters of the simulation and certain exposed attributes of orbiting bodies. Text fields inherit from the Text class and have a reference to a button to toggle editing.

```cpp
TextField(vector3 position, FieldValue& writeto, Graphyte& g, std::string default_text = "This Is An Input Field") :
    Text(*g.GetTextParams(default_text, 16, text_color), fvalue(writeto)
{
    vector3 dimensions = { Get_Texture().getWidth(), Get_Texture().getHeight(), 0 };
    input_text = default_text;
    button = new Button(position, dimensions);
    Set_Position({ position.x, position.y, 10 });
    g.AddTextToRenderQueue(this); //Beautiful
}
```

The constructor method calls the parent constructor before defining the button dimensions and position of the text field. The text field is then added to Graphyte's render queue for text elements.

Camera

The camera class handles the projection of objects in 3D world space to 2D screen space. All of what is seen on screen has been adjusted by the camera via the world-space to screen-space method. The camera also has a clipping plane so that objects behind the camera are not rendered.

```cpp
vector3 WorldSpaceToScreenSpace(vector3 world_pos, float screen_height, float screen_width)
{
    //manipulate world_pos here such that it is rotated around centre of universe
    vector3 rotated_world_pos = rotate(camera_rotation, world_pos, { 0, 0, 0 });

    vector3 pos = rotated_world_pos - position;
    //std::cout << "WORLD POS: " << world_pos.x << " | CAMERA POS: " << position.x << " | => " << pos.x;
    if (pos.z < clipping_z)
    {
        //DONT DRAW IT
        //printf("Culled a vertex hopefully \n");
        return { 0, 0, -1 };
    }
    else {
        //Why ignore the screen_width? Because the screen is wide and we don't want to stretch the projection
        vector3 Screen_Space_Pos = {
        (pos.x / pos.z) * screen_height,
        (pos.y / pos.z) * screen_height,
        pos.z
        };
        return Screen_Space_Pos;
    }
}
```

The "**rotated_world_pos**" vector3 is used in rotating every point to be rendered in the simulation around the world origin. Rotation is controller by the user's arrow keys:

```
case SDLK_UP:
    //Rotate Up
    gCamera.RotateCamera({ 0.01, 0, 0 });
    break;

case SDLK_DOWN:
    //Rotate Down
    gCamera.RotateCamera({ -0.01, 0, 0 });
    break;

case SDLK_LEFT:
    //Rotate Left
    gCamera.RotateCamera({ 0, -0.01, 0 });
    break;

case SDLK_RIGHT:
    //Rotate Right
    gCamera.RotateCamera({ 0, 0.01, 0 });
    break;
}
```

As the keys are pressed, the **camera_rotation** attribute of **Camera** are changed via the **RotateCamera** method. This way, the user is seemingly able to manoeuvre the camera about the world origin, facilitating 3D interaction, whereas in reality, every pixel is rotated as necessary before being drawn. This is done so that none of the simulation's positions or velocities need to be augmented when rotating every object around the origin. Rotating within the **WorldSpaceToScreenSpace** method preserves the structure of the system and integrity of the simulation.

Simulation Storage Solution

The storage system for Orbyte, originally intended to be implemented as a database, has instead been developed to be a binary file storage system. Every simulation can be saved to a .orbyte file at a given path[12].

Binary files were implemented instead of a database as they are, in this use case, more memory efficient (due to my own format of file) and faster to read and write from as there is no external library being used to interface with a database. It is also never necessary to partially access saved variables of a simulation, as you would be able to if using a table. Simulations are always written or read in one go. There is therefore no need for the storage solution to involve a database, however the OrbitBodyData structure has been designed with the principles of good table design in mind, where every fact stored in OrbitBodyData is about the body and only about the body.

OrbitBodyData

This structure is a representational abstraction of an Orbit Body. The Body Class contains a method: "GetOrbitBodyData" which returns this structure. The definition of OrbitBodyData is shown below.

---

[12] Path to .orbyte files for reading or writing are entered in the global path input

```
struct OrbitBodyData
{
    //Information for storage
    std::string name; // Name of Body
    vector3 center; // Position of Body
    double mass; // Mass of Body
    double scale; // Scale of Body
    vector3 velocity; // Velocity of Body

    uint8_t bytes_for_name; //So the first 8 bits of the file will tell us how many bytes the name contains. Name being the only var with "unlimited length"
    OrbitBodyData(std::string _name = "", vector3 _center = { 0, 0, 0 }, double _mass = 1, double _scale = 1, vector3 _velocity = {0, 0, 0})
    {
        name = _name;
        center = _center;
        scale = _scale;
        velocity = _velocity;
        mass = _mass;
        bytes_for_name = (uint8_t)name.length();
    }
};
```

This representation is then used directly in writing to and reading from .orbyte files. It strips down the Body attributes to only those that are essential for instantiating it again at the point when it was saved.

Information about "parent" bodies are not stored in OrbitBodyData. For first order orbiting bodies

OrbitBodyCollection

This structure stores a collection of OrbitBodyData in a hash table. OrbitBodyCollection is used in SimulationData as a way of encapsulating all the Orbit Bodies in a simulation.

```
//A Hash table of orbit objects
struct OrbitBodyCollection
{
private:
    //101 has been chosen as it is prime and not too close to a power of two
    // => Maximum of 101 bodies in storage!
    std::vector<OrbitBodyData> data = std::vector<OrbitBodyData>(101);

    int Hash(std::string name) { ... }

    void TryWrite(OrbitBodyData d, int index) { ... }

    OrbitBodyData TryRead(std::string name, int index) { ... }

public:
    void AddBodyData(OrbitBodyData new_data) { ... }

    OrbitBodyData GetBodyData(std::string name) { ... }

    std::vector<OrbitBodyData> GetAllOrbits() { ... }
};
```

An OrbitBodyCollection **can** store a maximum of 101 OrbitBodyData. One slot is purposefully kept empty so that the read method does not recurse infinitely (see COLLISION AVOIDANCE), so ultimately the structure only stores 100 valid Orbits.

The TryWrite method is a recursive algorithm used to write to the hash table, with open addressing collision avoidance. TryRead is also recursive; it reads from the hash table given the index to read from, and compares it to the name it is searching for.

Hash Table

```cpp
//101 has been chosen as it is prime and not too close to a power of two
// => Maximum of 101 bodies in storage!
std::vector<OrbitBodyData> data = std::vector<OrbitBodyData>(101);

int Hash(std::string name)
{
    // Reverse the string. A palindrome could XOR itself
    std::string reverse = name;
    reverse_string(reverse, reverse.length() - 1, 0);

    // XOR to produce Hash
    std::string hash = bitwise_string_xor(name, reverse);
    int total = 0;

    for (int i = 0; i < hash.length(); i++)
    {
        total += int(hash.at(i));
    }

    // Return Index To Write
    return (total % data.size());
}
```

The index within the data vector to write to is gotten through a hashing algorithm. This function returns an integer and the OrbitBodyData is then stored in that location.

Collision Avoidance

Open addressing is used when writing to the hash table so that if a collision results from the hashing algorithm generating two identical keys, the next open available address will be written to.

```cpp
void TryWrite(OrbitBodyData d, int index)
{
    // Saving one empty space so that Reading does not recurse infinitely
    // Max bodies can store: 100
    if (count < data.size() - 1)
    {
        if (data[index].name == "")
        {
            // If address is empty. Bodies Cannot have no name!
            data[index] = d;
            count++;
            return;
        }
        else
        {
            //Some recursion for collision avoidance with open addressing
            TryWrite(d, (index + 1) % data.size());
        }
    }
    else {
        std::cout << "\nErr. Cannot write to OrbitBodyCollection => Hash Table is full!\n";
    }
}
```

If a non-empty address is to be written to, the method recurses and passes an incremented index to write to with the same data (mod the size of the hash table so that it wraps around to zero if necessary).

When reading, another recursive method is used: TryRead. The same "recurse & increment" strategy is used:

```cpp
OrbitBodyData TryRead(std::string name, int index)
{
    if (data[index].name == "")
    {
        //Unsuccessful Read
        return data[index]; //Equivalent to NULL
    }
    if (data[index].name == name)
    {
        //Successful Read
        return data[index];
    }
    else {
        //Increment & read again
        return TryRead(name, (index + 1) % data.size()); //Careful with recursion depth!
    }
}
```

The integer **index** passed to both methods is generated from the hashing algorithm shown under HASH TABLE.

```cpp
void AddBodyData(OrbitBodyData new_data)
{
    // Try write data to address generated by Hash() method
    data[Hash(new_data.name)] = new_data;

    // debug
    std::cout << "\n Adding body data with hash: " << Hash(new_data.name)
        << "\n" << "Testing fetch [If blank, problem!]: "
        << GetBodyData(new_data.name).name << "\n";
}

OrbitBodyData GetBodyData(std::string name)
{
    // Return OrbitBodyData stored in the Hash table
    // at address given by Hash(name)
    return TryRead(name, Hash(name));
}
```

SimulationData

The highest level of encapsulation in the storage solution: The **SimulationData** structure represents an entire simulation in terms of a few key attributes.

```cpp
struct SimulationData
{
    double cb_mass = 0; // Mass of center body
    double cb_scale = 0; // Scale of center body
    OrbitBodyCollection obc; // All orbits
    vector3 c_pos; // Camera position
};
```

An instance of this type is passed to the **WriteDataToFile** method in the **DataController** class, describing the entire simulation state to be saved with only 4 fields.

DataController

This class handles all writing to and reading from .orbyte files.

```cpp
class DataController
{
public:
    // Write simulation data to a .orbyte file at specified path.
    int WriteDataToFile(SimulationData sd, std::vector<std::string> bodies_to_save, std::string path) { ... }

    // Read simulation data from a .orbyte file at given path.
    SimulationData ReadDataFromFile(std::string path) { ... }
};
```

The class contains two public methods that specify the format that the binary files are written in. The details of these methods are covered in the BINARY FILE section.

Usage

The below methods are from the **Simulation** class and show usage of the DataController.

```cpp
void save()
{
    OrbitBodyCollection obc; // Collection of orbits
    std::vector<std::string> to_save; // Name of orbits to save

    // Check that body is not about to be deleted before saving.
    for (Body* b : orbiting_bodies)
    {
        if (!b->to_delete)
        {
            to_save.push_back(b->name); // Add name
            obc.AddBodyData(b->GetOrbitBodyData()); // Add orbit data
        }
    }

    // Encapsulate all simulation data
    SimulationData sd = { Sun.mass, Sun.scale, obc, gCamera.position };

    // Write to .orbyte file
    data_controller.WriteDataToFile(sd, to_save, path_source);
}
```

In saving, SimulationData is created from all the essential attributes of the simulation before being given to the WriteDataToFile method of the DataController.

```cpp
void open()
{
    std::cout << "\nOpening File";
    // New Simulation Data
    SimulationData new_sd = data_controller.ReadDataFromFile(path_source);
    // Pause simulation
    time_scale = 0;

    // Load all parameters from SimulationData
    Sun.mass = new_sd.cb_mass;
    std::cout << "\n Sun mass: " << Sun.mass;
    Sun.scale = new_sd.cb_scale;
    gCamera.position = new_sd.c_pos;

    // Clear Old Orbits
    for (Body* old_orbit : orbiting_bodies)
    {
        old_orbit->Delete();
    }

    OrbitBodyCollection obc = new_sd.obc;
    std::vector<OrbitBodyData> orbits = obc.GetAllOrbits();
    // Instantiate Orbits
    for (OrbitBodyData orbit : orbits)
    {
        add_specific_orbit(orbit);
    }
}
```

When opening, the DataController's method: **ReadDataFromFile** returns a new SimulationData type instance that is then used to set up the simulation as it was when it was saved.

Binary File

The .orbyte file type was developed so that I could have better control over how data was being stored. It is also more memory efficient than utilising a database solution. Read-Write speeds are fast enough to have negligible impact on the simulation when saving or loading.

Format

```cpp
int WriteDataToFile(SimulationData sd, std::vector<std::string> bodies_to_save, std::string path)
{
    double no_orbits = bodies_to_save.size();

    std::cout << "Writing data to file: " << path << "\n";

    std::ofstream out(path, std::ios::binary | std::ios::out);;
    if (!out)
    {
        std::cout << "\nERR. Writing to file failed.";
        return -1;
    }
    std::cout << (char*)&sd;
    uint8_t my_size = sizeof(sd);

    out.write((char*)&sd.cb_mass, sizeof(double)); // Center Body Mass
    out.write((char*)&sd.cb_scale, sizeof(double)); // Center Body Scale
    out.write((char*)&sd.c_pos, sizeof(vector3)); // Camera Position
    out.write((char*)&no_orbits, sizeof(double)); // Number OF Orbits

    for (int i = 0; i < bodies_to_save.size(); i++)
    {
        //Access Data via hashing algorithm & hash table
        OrbitBodyData data = sd.obc.GetBodyData(bodies_to_save[i]);

        double bfn = data.bytes_for_name; // Number of characters in name
        out.write((char*)&bfn, sizeof(double));

        std::string n = data.name;
        for (char n_char : n) //Write Name
        {
            out.write((char*)&n_char, sizeof(char));
        }

        vector3 c = data.center; // Body position
        out.write((char*)&c, sizeof(vector3));

        double m = data.mass; // Body Mass
        out.write((char*)&m, sizeof(double));

        double s = data.scale; // Body Scale
        out.write((char*)&s, sizeof(double));

        vector3 v = data.velocity; // Body Velocity
        out.write((char*)&v, sizeof(vector3));
    }
    out.close();

    std::cout << "\nBytes: " << my_size + 1;
    return 0;
}
```

Data is written to a .orbyte file using *ofstream*. As the number of bytes per variable is known, it is not necessary to separate the data when writing, as reading can divide the input bit stream into appropriate bytes and read the variables one after another.

Data in storage (in order):
1. Centre Body Mass (64 bits : double)
2. Center Body Scale (64 bits : double)
3. Camera Position (64 * 3 bits : vector3)
4. Number of Orbits (64 bits : double)
5. Body Data:
    a. Bytes For Name (64 bits : double)
    b. Name:
        i. Character (8 bits : char)
    c. Position (64 * 3 bits : vector3)
    d. Mass (64 bits : double)

        e. Scale (64 bits : double)

        f. Velocity (64 * 3 : vector3)

The only two items to be stored with variable size are the name of an orbit body and number of orbit bodies. This is handled by storing the number of bodies before the data for each orbit is added to the file, so that it can be written and read iteratively. Similarly, the number of characters in a name is stored before the name itself.

Below is the read method in DataController.

```cpp
// Read simulation data from a .orbyte file at given path.
SimulationData ReadDataFromFile(std::string path)
{
    SimulationData sd; // New Simulation Data
    double no_orbits; // Number of orbits to read

    std::ifstream in(path, std::ios::binary | std::ios::in);
    if (!in)
    {
        std::cout << "\nERR. Reading from file failed.";
        return sd; //But center mass will be 0, so known error.
    }
    // Mass
    in.read((char*)&sd.cb_mass, sizeof(double)); std::cout << "\nReading mass: " << sd.cb_mass;
    // Scale
    in.read((char*)&sd.cb_scale, sizeof(double)); std::cout << "\nReading scale: " << sd.cb_scale;
    // Camera Position
    in.read((char*)&sd.c_pos, sizeof(vector3)); std::cout << "\nReading c_pos: " << sd.c_pos.Debug();
    // Number of Orbits
    in.read((char*)&no_orbits, sizeof(double)); std::cout << "\nReading No. Orbits: " << no_orbits;
```

As previously explained, due to all .orbyte files conforming to a specified format, reading from the binary files is simple and unless tampered with externally, will always be successful provided writing was copacetic.

```cpp
    // Iterate through orbits
    for (int i = 0; i < no_orbits; i++)
    {
        OrbitBodyData data; // New Data

        double bfn; // Bytes for Name
        in.read((char*)&bfn, sizeof(double));
        data.bytes_for_name = bfn;
        std::cout << "\nReading No. bytes: " << (double)data.bytes_for_name;

        std::string name = ""; // Reading name
        for (int j = 0; j < data.bytes_for_name; j++)
        {
            char c; // Character in name
            in.read((char*)&c, sizeof(char));
            name += c;
        }
        data.name = name; std::cout << "\nReading name: " << name;
        // Position
        in.read((char*)&data.center, sizeof(vector3));
        // Mass
        in.read((char*)&data.mass, sizeof(double));
        // Scale
        in.read((char*)&data.scale, sizeof(double));
        // Velocity
        in.read((char*)&data.velocity, sizeof(vector3));
        // Add to OrbitBodyCollection
        sd.obc.AddBodyData(data);
    }

    in.close();


    if (!in.good())
    {
        std::cout << "\n\nErr. Reading from .orbyte file failed!\n\n";
    }

    return sd;
}
```

SimulationData has been successfully read from a .orbyte binary file if this method executes properly.

Example : Saving Earth

only_earth.orbyte

Type of file: ORBYTE File (.orbyte)

Opens with: Notepad   Change...

Location: ██████████████

Size: 125 bytes (125 bytes)

Figure 9 : Example .orbyte file with file size
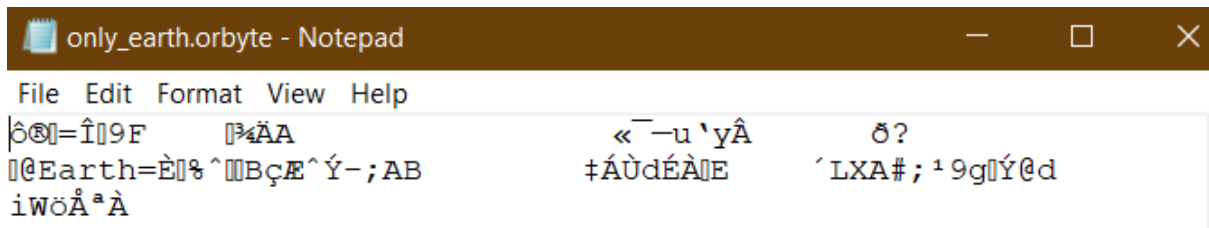
Figure 10 : Example .orbyte file viewed with notepad.

The above example contains an "empty" simulation with just the earth and the sun. The state of the simulation before being saved is shown below.
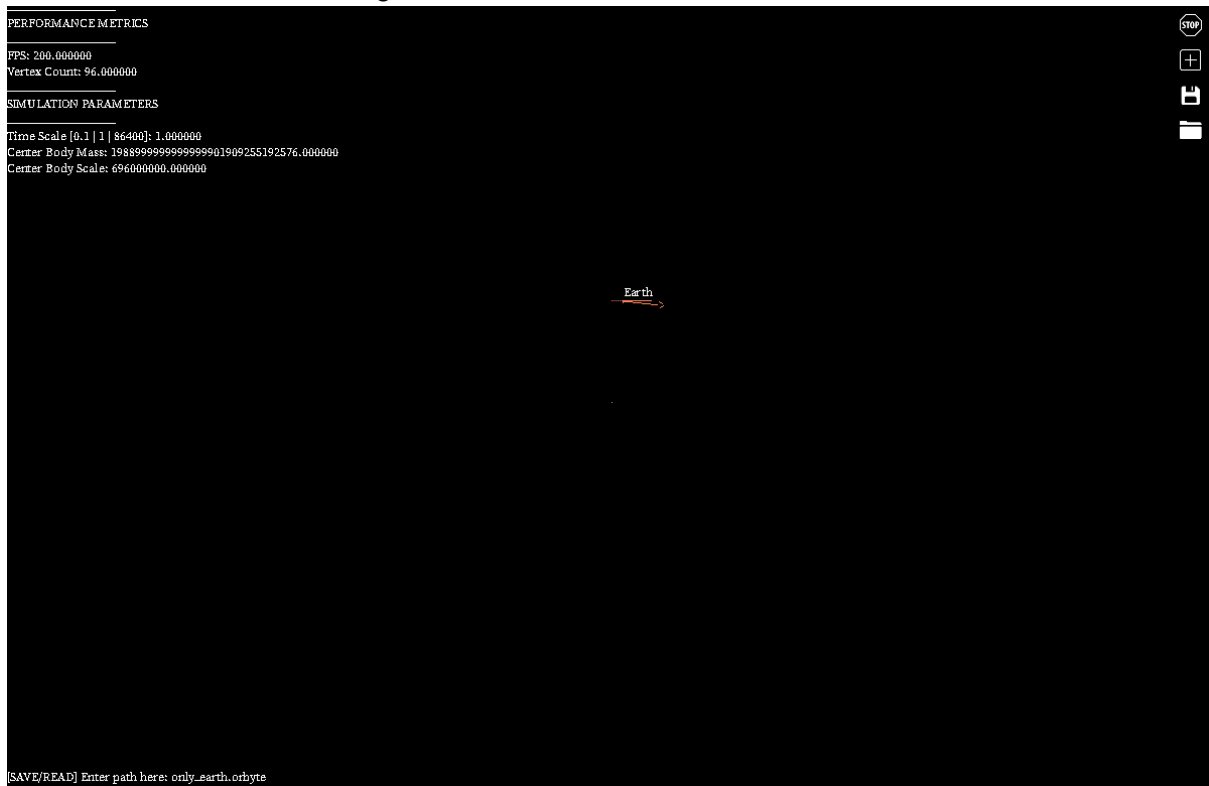
Figure 11 : Current state of simulation when saving.

In the above image, it is also possible to see the "Enter Path Here" input field that serves as a read/write path for .orbyte files.

Read/Write Path


```
/*
    PATH TO OPEN FROM FILE
*/
GUI_Block path_gui;
path_gui.position = { (-SCREEN_WIDTH / 2), (-SCREEN_HEIGHT / 2) + 36, 0 };
Text* path_input_prompt = graphyte.CreateText("[SAVE/READ] Enter path here: ", 12);
path_gui.Add_Stacked_Element(path_input_prompt);
StringFieldValue path_to_file(&path_source, NULL, "(([A-Z]|[a-z]|(_))|[ ])+\.orbyte"); //Custom regex
TextField* path_input = new TextField({ ( - SCREEN_WIDTH / 2), (-SCREEN_HEIGHT / 2), 0}, path_to_file, graphyte, "solar_system.orbyte");
path_source = "solar_system.orbyte"; //default
graphyte.text_fields.push_back(path_input);
path_gui.Add_Inline_Element(path_input);
```

"(([A-Z]|[a-z]|(_))|[ ])+\.orbyte"

The read/write path GUI element, used for opening and saving .orbyte files, is validated by a custom Regex string facilitated by the **StringFieldValue** type.

Testing

Testing strategy

In order to test Orbyte in its current state, a combination of 3 testing methods will be used.

- Black Box Testing: Using the application as intended via the Graphical User Interface, giving standard, boundary and erroneous data to each field within the User Interface and comparing result to the expected outcome.
- (Integrated) Module Testing: Largely done using debug output messages to the console while the application is in use.
- End User Testing: Using Orbyte to complete an A-Level physics practical done in class with a Physics teacher.

Additionally, stress testing data will be included to demonstrate the performance of Orbyte given a number of orbits being simulated. This information will be discussed in the **EVALUATION** section.

Testing plan

Key

"STND": Standard (correct) data
"INCR": Incorrect data
"BNDR": Boundary data[13]

"AE": As Expected
"ERR": Error / Incorrect result

"PR": Considered Pre-requisite.

Plan

Test Number	1
Objective / Version Number:	0.0.1 - 0.0.3
Description:	Testing to ensure that key presses are registered by Orbyte. (Anything preceding 0.0.3 considered prerequisite for this test).
Test Data:	I : Left Arrow Key [STND] II : Right Arrow Key [STND] III : Space Bar [STND] IV : "G" Key [INCR]
Expected Result:	I : Debug Message II : Debug Message III : Debug Message IV : No Message

---

[13] "Boundary Data" will be interpreted loosely from test to test. In some situation the conventional idea of giving data on the edge of a range is inapplicable. Technically for many input fields the boundary data would be the maximum representable double. The simulation was explicitly designed to allow any VALID data.

Actual Result:	I : AE II : AE III : AE IV : AE
Action needed?	None

Test Number	2
Objective / Version Number:	0.6.2.0[14] [PR: 0.4.2.0, 0.4.1.0, 0.3, 0.2, 0.1.3.1, 0.1.3.0, 0.1.2.0, 0.1.1.1, 0.1.1.0]
Description:	Instantiate Orbit Body in simulation with default parameters (through code). This is to test a combination of modules and the **Body** class.
Test Data:	```Body earth = Body("Earth", {0, 1.49E11, 0}, 5.97E24, 6.37E6, { 30000, 0, 0 }, Sun.mu, graphyte, false);

//Name: "Earth"
//Radius of orbit: 1.49E11m
//Mass: 5.97E24kg
//Scale: 6.37E6m
//T_Velocity: 30000ms^-1
//Mu: Sun's mu (See Analysis)``` |
Expected Result:	Orbit body instantiated with name: "Earth" Displayed vertically above the sun. The period of the orbit should be close to 365 days.
Actual Result:	*See corresponding supporting evidence in subsequent section.* Orbit Body instantiated in correct place, confirming that Body constructor works as expected. [AE] Graphyte correctly draws the 3D geometry. [AE] Period of orbit predicted within acceptable range. [ERR : I | *Technically within acceptable range, however nonetheless default parameters will be changed for more accuracy.*]
Action needed?	ERR : I → Change default orbit constructor parameters for earth to have more accurate values.

Test Number	3
Objective / Version Number:	0.6.3.0 [PR: 0.5, 0.4, 0.3, 0.2, 0.1, 0.0.2, 0.0.1]
Description:	Second generation orbit body instantiated with name "Moon". Checking to see that the

---

[14] Though many version points have been skipped, they are all being tested. These tests are targeted at the current version of Orbyte in its entirety, not individual components in the past. The iterative development process and testing that occurred throughout is discussed later.

	object is instantiated near the parent ("Earth"), the period of orbit is near 27 days, and that the geometry is unique.
Test Data:	Name: "Moon" Radius: 3.844E8m Mass: 7.3E22kg Scale: 1.7E5m Velocity: 1024m/s
Expected Result:	3D object (Cube) instantiated in orbit near the Earth. Should have a period of near 27 days.
Actual Result:	*See corresponding supporting evidence in subsequent section.* Moon instantiated correctly. [AE] Graphyte correctly renders the mesh wireframe for satellites (Cube). [AE] Predicted period equal to accepted values for period of moon's orbit around earth: 27.3 days. [AE]
Action needed?	None.

Test Number	4
Objective / Version Number:	0.7.2.1 [PR: 0.6.2.0, 0.4, 0.3, 0.2, 0.1, 0.0]
Description:	Test **FunctionButton** by adding a GUI button that dynamically instantiates a new orbit body.
Test Data:	Button press.
Expected Result:	Body instantiated with default parameters as 1st Generation orbit about the centre body.
Actual Result:	*See corresponding supporting evidence in subsequent section.* Body instantiated with default parameters, orbiting the central body. [AE]
Action needed?	None.

Test Number	5
Objective / Version Number:	0.7.2.2 [PR: 0.3, 0.2]
Description:	Test the simulation "pause" button. Should set the timescale to 0 so that all bodies do not orbit, clock does not advance but UI etc. is still usable.
Test Data:	Button press.
Expected Result:	Simulation time-scale is set to 0. All values within body inspectors should not change and simulation GUI should still be operational.
Actual Result:	*See corresponding supporting evidence in subsequent section.*

	Simulation time-scale set to 0 [AE], however text field showing current time scale not updated. [ERR: I] Simulation clock does not progress, due to time scale value being successfully set to 0. [AE] GUI still functional [AE].
Action needed?	ERR : 1 → Change "toggle_pause" method within Simulation so that it updates the text field.

Test Number	6
Objective / Version Number:	0.8.2.0 [PR: 0.7, 0.4, 0.3, 0.2, 0.1]
Description:	Test ability to click on an orbit body and have the inspector appear for that body. Clicking anywhere else should collapse the inspector.
Test Data:	Button press on one of many orbiting bodies.
Expected Result:	Orbit Body inspector should appear with the attributes of the body selected. The name displayed will correspond to that selected. The inspector will continue to update as the body orbits.
Actual Result:	*See corresponding supporting evidence in subsequent section.* Clicking on an orbit body shows its inspector; clicking in empty space hides the inspector. [AE]
Action needed?	None.

Test Number	7
Objective / Version Number:	0.8.3.0 [PR: 0.7, 0.6, 0.3, 0.1]
Description:	Test "Time Scale" Input field within the Simulation Parameters section of the GUI. Test multiple different values including negative real numbers, and check console & simulation for confirmation. This tests the simulation's ability to cope under different time scales and the DoubleFieldValue validation method.
Test Data:	I : 1 [STND] II : -1 [STND] III : 0.1 [STND] IV : 86400 [STND] V : -86400 [STND] VI : 0 [BNDR] VII : 8.64E7 [BNDR] VIII : -8.64E7 [BNDR]

	IX : "4EB9" [INCR] X : "8&9" [INCR]
Expected Result:	I : Simulation time scale becomes 1 second per second II : -1 second per second (reverse) III : 0.1 seconds per second (slow) IV : 1 day per second (fast) V : -1 day per second (fast reverse) VI : Paused VII : Really fast simulation progression. VIII : Really fast simulation regression. IX : Rejection & Value NOT updated X : Rejection & Value NOT updated
Actual Result:	*See corresponding supporting evidence in subsequent section.* I : AE II : AE III : AE IV : AE V : AE VI : AE VII : AE VIII : AE IX : AE X : [ERR : I] "8&9" incorrectly accepted as a valid input.
Action needed?	ERR : I → Amend DoubleFieldValue Validate method. Correct regular expression filtering inputs.

Test Number	8
Objective / Version Number:	0.8.3.0 [PR: 0.7, 0.6, 0.3, 0.1]
Description:	Test "Centre Body Mass" Input field under Simulation Parameters section of GUI. This tests the input field as well as the ability of bodies in orbit to adjust their physics simulation to the new value for "mu", due to the updated mass of the body they orbit.
Test Data:	I : "1.989E30" [STND] II : "5.972E24" [STND] III : "-1.989E30" [STND] IV : 0 [BNDR] V : "3E32" [BNDR] VI : "1.0.0.0" [INCR] VII : "1$$./" [INCR]
Expected Result:	I : Mass set to value & orbits change. II : Mass set to value & orbits change. III : Mass set to value & orbits change. IV : Mass set to 0 and orbits no longer affected by centre body. V : All orbits dragged towards centre.

	VI : Rejection & Value not updated. VII : Rejection & Value not updated.
Actual Result:	*See corresponding supporting evidence in subsequent section.* I : AE II : AE III : AE (albeit interesting to watch) IV : AE V : AE VI : AE VII : AE
Action needed?	None.

Test Number	9
Objective / Version Number:	0.8.3.0 [PR: 0.7, 0.6]
Description:	Test "Centre Body Scale" Input field. This field is purely cosmetic, changing the diameter of the geometry within the centre body's mesh. Tests the Input field and the methods surrounding the rendering & updating of meshes.
Test Data:	I : "12E9" [STND] II : "3.99E30" [BNDR] III : 0 [BNDR] IV: "0.00HHGTTG" [INCR]
Expected Result:	I : Large mesh seen. II : VERY large mesh seen. III : Mesh rendered as a point. IV : Rejection; Value & Geometry not updated.
Actual Result:	*See corresponding supporting evidence in subsequent section.* I : AE II : AE III : AE IV : AE
Action needed?	None.

Test Number	10
Objective / Version Number:	0.8.3.1 [PR: 0.0 – 0.8.2.0]
Description:	Testing name input field within Orbit Inspector. Tests StringFieldValue & Graphyte system.
Test Data:	I : "Earth" [STND] II : "16" [STND] III : "" [INCR]
Expected Result:	I : Name of orbit changed to "Earth"

	II : Name of orbit changed to "16" III : Rejected & Name of orbit not changed.
Actual Result:	*See corresponding supporting evidence in subsequent section.* I : AE II : Rejected [ERR : I] III : Accepted [ERR : II]
Action needed?	ERR : I → Amend StringFieldValue validation regular expression to include digits. ERR : II → Amend StringFieldValue validation regular expression to not allow empty strings.

Test Number	11
Objective / Version Number:	0.8.3.1 [PR: 0.0 – 0.8.2.0]
Description:	Testing scale input field within Orbit Inspector. Changing scale of body in orbit.
Test Data:	I : "6.37E10" [STND] II : 0 [BNDR] III : "" [INCR]
Expected Result:	I : Scale of orbit body changed, and geometry recalculated. II : Orbit body rendered as a single point. III : Rejection & Geometry unchanged.
Actual Result:	*See corresponding supporting evidence in subsequent section.* I : AE II : AE III : AE
Action needed?	None.

Test Number	12
Objective / Version Number:	0.8.3.1 [PR: 0.0 – 0.8.2.0]
Description:	Testing mass input field within Orbit Inspector. Dynamically changing mass of body in orbit. *To see the effects of changing the mass of a 1st Generation orbit (The earth). A 2nd Generation orbit (The moon) will be instantiated. This has the added benefit of verifying that the polymorphism within the child class **Satellite** works correctly.*
Test Data:	I : "5.972E26" [STND] II : 0 [BNDR] III : "Number" [INCR]
Expected Result:	I : Mass of Earth changed. 2nd Gen. Orbit to experience greater acceleration towards Earth.

	II : Mass of Earth changed. 2nd Gen. Orbit to experience no force due to Earth, continue with tangential velocity at that time. III : Rejection & Mass / 2nd Gen. Orbit unchanged.
Actual Result:	*See corresponding supporting evidence in subsequent section.* I : AE II : AE III : AE
Action needed?	None. (Though acceleration arrows are going to be increased in size so that they are easier to see).

Test Number	13
Objective / Version Number:	0.8.3.1 [PR: 0.0 – 0.8.2.0]
Description:	Testing position input field within Orbit Inspector. Changing position of body in orbit.
Test Data:	I : (0, -1.49E10, 0) [STND] II : (0, 0, 0) [BNDR] III : (1.E19, 0, 0) [INCR]
Expected Result:	I : Body in orbit moved to specified position. II : Body in orbit moved to position of centre body. Strange behaviour expected. III : Rejection & Position unchanged.
Actual Result:	*See corresponding supporting evidence in subsequent section.* I : AE II : AE (Strange behaviour when an orbit hits the origin. *Designed this way to avoid division by 0 problems*). III : AE
Action needed?	None.

Test Number	14
Objective / Version Number:	0.8.3.1 [PR: 0.0 – 0.8.2.0]
Description:	Testing velocity input field within Orbit Inspector. Changing velocity of body in orbit.
Test Data:	I : (0, 0, 30000) [STND] II : (0, 0, 0) [BNDR] III : ("A", "B", "C") [INCR]
Expected Result:	I : Body velocity changed to only be in the position Z axis. II : Body no longer has angular velocity; will move towards centre body. III : Rejection & Velocity unchanged.
Actual Result:	*See corresponding supporting evidence in subsequent section.* I : AE

	II : AE
	III : AE
Action needed?	None.

Test Number	15
Objective / Version Number:	0.8.3.2 [PR: 0.0 – 0.8.2.0]
Description:	Testing FunctionButton that instantiates new child satellite within Orbit Inspector.
Test Data:	Button press.
Expected Result:	Satellite Instantiated as child of selected Orbit Body.
Actual Result:	*See corresponding supporting evidence in subsequent section.* AE. Moon instantiated in orbit around the earth with period of 27 days. Video shows satellite instantiation, update and 2nd generation orbit inspector.
Action needed?	None.

Test Number	16
Objective / Version Number:	0.8.3.3 [PR: 0.0 – 0.8.2.0]
Description:	Testing FunctionButton that resets selected orbit body within the Orbit Inspector.
Test Data:	Button press.
Expected Result:	Orbit reset to initial position and velocity.
Actual Result:	*See corresponding supporting evidence in subsequent section.* AE. Clicking inspector "reset" button set the orbit parameters to original values.
Action needed?	None.

Test Number	17
Objective / Version Number:	0.8.3.4 [PR: 0.0 – 0.8.2.0]
Description:	Testing FunctionButton that deletes selected orbit body within the orbit inspector.
Test Data:	Button press.
Expected Result:	Orbit deleted: removed from queue and no longer rendered.
Actual Result:	*See corresponding supporting evidence in subsequent section.* AE. Orbit deleted upon button press.
Action needed?	None.

Test Number	18
Objective / Version Number:	0.8.5.0 [PR: 0.0 – 0.8.2.0]
Description:	Right-Click on orbit to snap camera to that body's orbit.
Test Data:	Right-Click on orbit body tag.

Expected Result:	Camera should follow the orbit body's x and y position, maintaining a fixed z distance.
Actual Result:	*See corresponding supporting evidence in subsequent section.* AE. Camera follows orbit of selected body.
Action needed?	None.

Test Number	19
Objective / Version Number:	0.9.2.0 [PR: 0.7, 0.3, 0.2, 0.1, 0.0]
Description:	Using path input field and "write" FunctionButton in simulation GUI, save a simulation to a .orbyte binary file.
Test Data:	Simulation configured such that the central body is earth and the orbiting body is the ISS. Path: "ISS.orbyte" Centre Body Mass : 5.97E24 Centre Body Scale : 6378E6 Position : (0, 6.778E6, 0) Scale : 109 Velocity : (7887, 0, 0) Button press.
Expected Result:	Simulation data written to a .orbyte file at specified path.
Actual Result:	*See corresponding supporting evidence in subsequent section.* AE. ISS orbit correctly simulated and successfully saved.
Action needed?	None. *Debug print to console to be added so that file size can be debugged.*

Test Number	20
Objective / Version Number:	0.9.2.1 [PR: 0.9.2.0, 0.9.1.1, 0.9.1.0, 0.7, 0.3, 0.2, 0.1, 0.0]
Description:	Using path input field and "read" FunctionButton in simulation GUI, read from a .orbyte file.
Test Data:	Path: "ISS.orbyte"
Expected Result:	Simulation opened displaying the orbit of the ISS around the earth.
Actual Result:	*See corresponding supporting evidence in subsequent section.* AE. Simulation successfully opened from supplied path with correct parameters.
Action needed?	None.

Test evidence

Supporting evidence for the tests is included below. Youtube links are to "unlisted" videos, such that only people with the given link are able to access the video, and not the public.

Evidence Supporting Test 1

Console Output:

Evidence Supporting Test 2

Figure 12 : Capture of simulation showing test orbit instantiated in correct place.

Figure 13 : Zoomed in capture of instantiated test orbit, showing that the Graphyte module successfully renders 3D geometry.


```
Earth
|| Mass: 5970000000000000281018368.000000
|| Position: 0.000000, 149000000000.000000, 0.000000
|| Velocity: 30000.000000, 0.000000, 0.000000
|| Period: 361.186192days
```

*NOTE: The period displayed in the console is not the result of a simulated orbit but from a "predicted path" of distance **pi * radius^2**. The orbit was not instantiated with the exact orbital parameters, therefore with a degree of error around 1%, this prediction is NOT a concern for the accuracy of the simulation. The actual accuracy of the simulation will be discussed in the EVALUATION section.*

Evidence Supporting Test 3

Figure 14 : Screenshot of Orbyte showing 2nd Generation orbit : Moon; orbiting the earth. Graphyte working correctly & Inspector open for moon.

Figure 15 : 2nd Generation Orbit Body Mesh with increased scale to show geometry.

Evidence Supporting Test 4

Console output upon button press:

```
Instantiated Orbiting Body with initial position: 0.000000, 58000000000.000000, 0.000000
and velocity: 47000.000000, 0.000000, 0.000000
```

Video evidence: https://youtu.be/QOVl0o4DKLY

Evidence Supporting Test 5

Figure 16 : Paused simulation

Video evidence: https://youtu.be/U8Neba0YVOE

Console output:

```
Function button clicked.

Paused Simulation
```

Evidence Supporting Test 6

Video evidence: https://youtu.be/TduRv3HxveI

Evidence Supporting Test 7

Console output showing tests: VIII, IX & X.

```
 Committing to text field value!
1=>1
 Valid field content
 Successfully wrote to double value from input field!
1User Pressed the Backspace
User is typing: 4
User is typing: E
User is typing: B
User is typing: 9

 Committing to text field value!

 Bad input recieved: 4EB9User Pressed the Backspace
User Pressed the Backspace
User Pressed the Backspace
User Pressed the Backspace
User is typing: 8
User is typing: &
User is typing: 9

 Committing to text field value!
8&9=>8
 Valid field content
 Successfully wrote to double value from input field!
8
```

Video Evidence: https://youtu.be/xTipiadRqD8

Evidence Supporting Test 8

Console output showing a rejection of the final test.

```
User is typing: 1
User is typing: $
User is typing: $
User is typing: .
User is typing: /

 Committing to text field value!

 Bad input recieved: 1$$./
```

Video evidence: https://youtu.be/M0XBVqfXU2E

Evidence Supporting Test 9

Console output showing rejection of incorrect test value:

```
User is typing: 0
User is typing: .
User is typing: 0
User is typing: 0
User is typing: H
User is typing: H
User is typing: G
User is typing: T
User is typing: T
User is typing: G

 Committing to text field value!

 Bad input recieved: 0.00HHGTTG
```

Video Evidence : https://youtu.be/M2xEZsfV_gM

Evidence Supporting Test 10

Video evidence : https://youtu.be/TTR4sdgwE4Q

Evidence Supporting Test 11

Console output showing rejection of incorrect data test:

```
Committing to text field value!

Bad input recieved:
```

Note: bad input in this case was "", hence nothing shown.

Video Evidence : https://youtu.be/4kn1SWUUGaw

Evidence Supporting Test 12

Video Evidence : https://youtu.be/9nHIPignPHY

Evidence Supporting Test 13

Console output showing rejection of incorrect data test:

```
User is typing: 1
User is typing: .
User is typing: E
User is typing: 1
User is typing: 9

 Committing to text field value!

 Bad input recieved: 1.E19
```

Video Evidence : https://youtu.be/YYHx9QyPKlQ

Evidence Supporting Test 14

Video Evidence : https://youtu.be/ZguXqlop1SM

Evidence Supporting Test 15

Console output showing result of button press & satellite instantiation:

```
Moon
|| Mass: 72999999999999997902848.000000
|| Position: 384400000.000000, 0.000000, 0.000000
|| Velocity: 0.000000, -1024.000000, 0.000000
|| Period: 27.299177days


_____

Earth has a new satellite: Moon | Total number of satellites: 1
```

Video Evidence : https://youtu.be/2nZl1J2RXl4

Evidence Supporting Test 16

Video Evidence : https://youtu.be/d4btG-WefgU

Evidence Supporting Test 17

Console output showing result of button press:

```
Function button clicked.

|||DELETED ORBIT BODY: Earth|||
```

Video Evidence : https://youtu.be/AqD_1dHHQ2I

Evidence Supporting Test 18

Video Evidence : https://youtu.be/CttMRIxoA7k

Evidence Supporting Test 19

Console output showing result of saving simulation:

```
 Adding body data with hash: 54
Testing fetch [If blank, problem!]: ISS
Writing data to file: simulations/ISS.orbyte
ç⌐d╟└‼E
```

.Orbyte file saved to:

ISS.orbyte

Type of file:	ORBYTE File (.orbyte)
Opens with:	Notepad	Change...
Location:	
Size:	123 bytes (123 bytes)

Video Evidence : https://youtu.be/WhX0AquuPHc
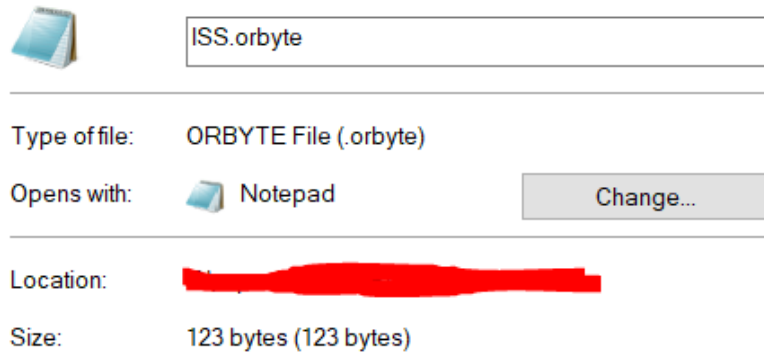
Evidence Supporting Test 20

Console output showing result of opening file:

```
Opening File
Reading mass: 5.97e+24
Reading scale: 6.378e+06
Reading c_pos: 0.000000, 0.000000, -51857419.550412
Reading No. Orbits: 1
Reading No. bytes: 3
Reading name: ISS
```

Video Evidence : https://youtu.be/8kg1Ek2RnJc

Qualitative testing

In order to gain more insight into the qualities of Orbyte, and to be able to test the applications usability, the end user (as introduced in **ANALYSIS**) was asked to supervise the set up of an example simulation. The end user watched and commented on the process of configuring the simulation such that the International space station was orbiting the earth. The following summarises Mr Rice's feedback:

- The user interface would be clearer if "fewer significant figures (were) displayed". They suggested rounding larger numbers and showing fields to fewer decimal places.
- They requested that each UI field "show the units", such as metres or seconds.
- Mr Rice suggested that it would be beneficial to be able to "reset the clock" within the simulation for timing purposes.

- They were "impressed by its (the simulation's) ability to simulate the orbit of the ISS".
- Mr Rice affirmed that the simulation could be used to "verify relationships learnt in A Level and GCSE" physics.
- The user was confident that the simulation could be "used in the classroom for demonstrations, once all bugs[15] are fixed".
- The user was impressed by the "sophistication of the simulation", particularly the simulation of 2nd generation orbits (satellites).

From observing the end user throughout the demonstration, it was evident that the simulation UI proved intuitive. The teacher knew what was happening, what every field requested and how it would affect the simulation. Additionally, the teacher acknowledged the benefits of being able to save simulation states, especially for use as a pre-configured simulation which could be used in a class data collection practical.

Stress Testing

To test the performance of Orbyte, this section covers how the FPS and memory usage of the simulation change with the instantiation of more orbits. See **ANALYSIS OF PERFORMANCE** under the **EVALUATION** section for discussion of results shown below.

Specifications:

Processor: Intel Core i7 @ 2.20 GHz
Memory: 32GB

Results:

Testing different numbers of orbits and comparing the FPS and memory usage.

---

[15] "Bugs" referring to some of the problems found in testing that were not fixed yet. All issues are now fixed, as shown in testing.

1 Orbit
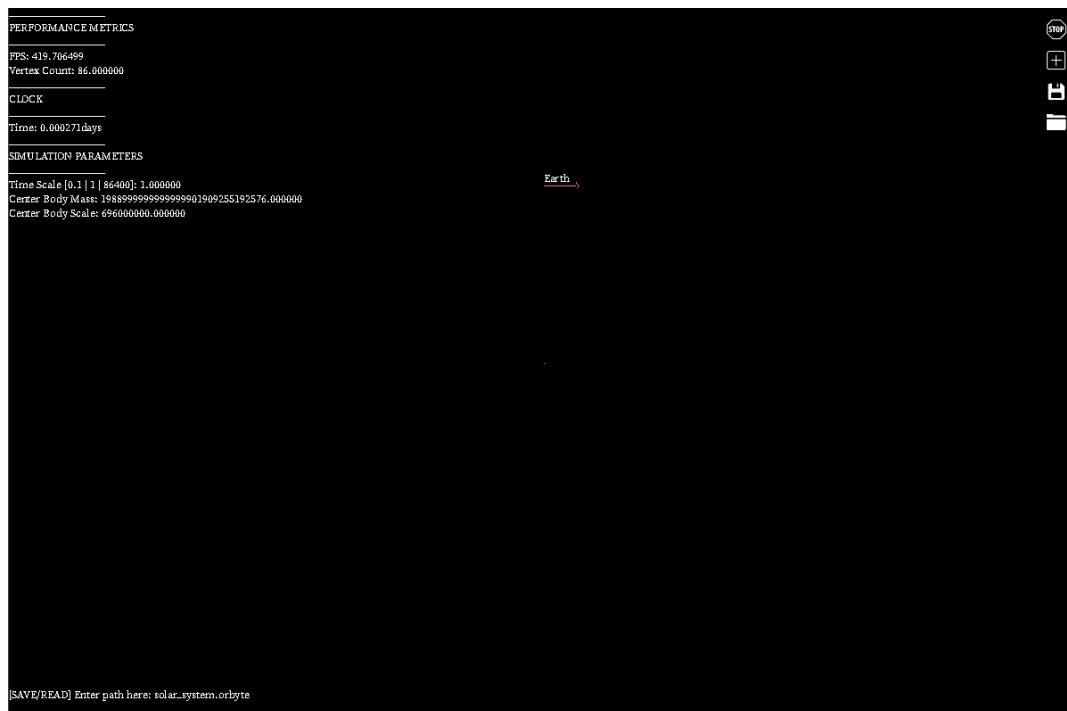
Frames Per Second: 420
Memory usage: 46MB

5 Orbits
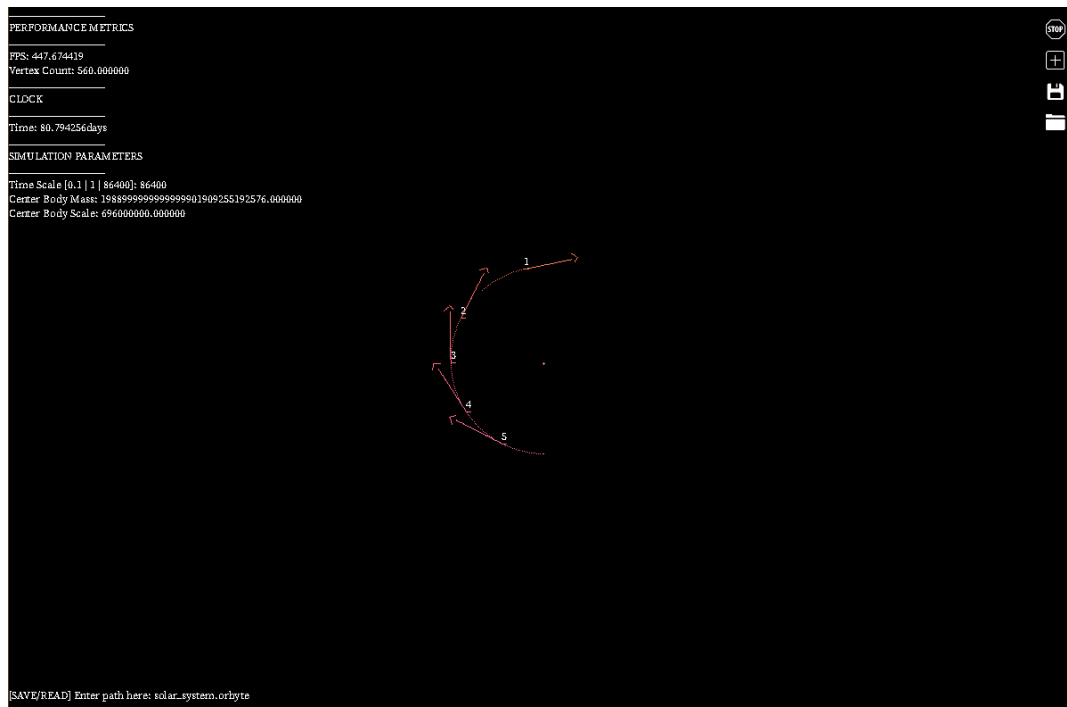
Frames Per Second: 448
Memory usage: 50MB

1 Orbit
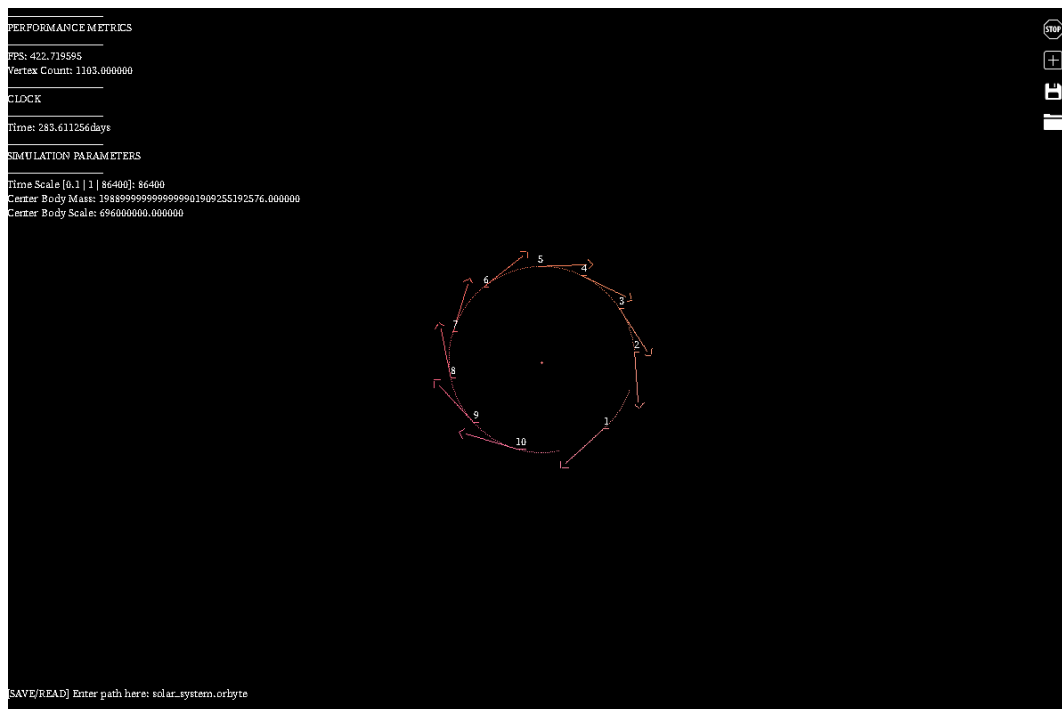
10 Orbits


Frames Per Second: 423
Memory Usage: 55MB

25 Orbits
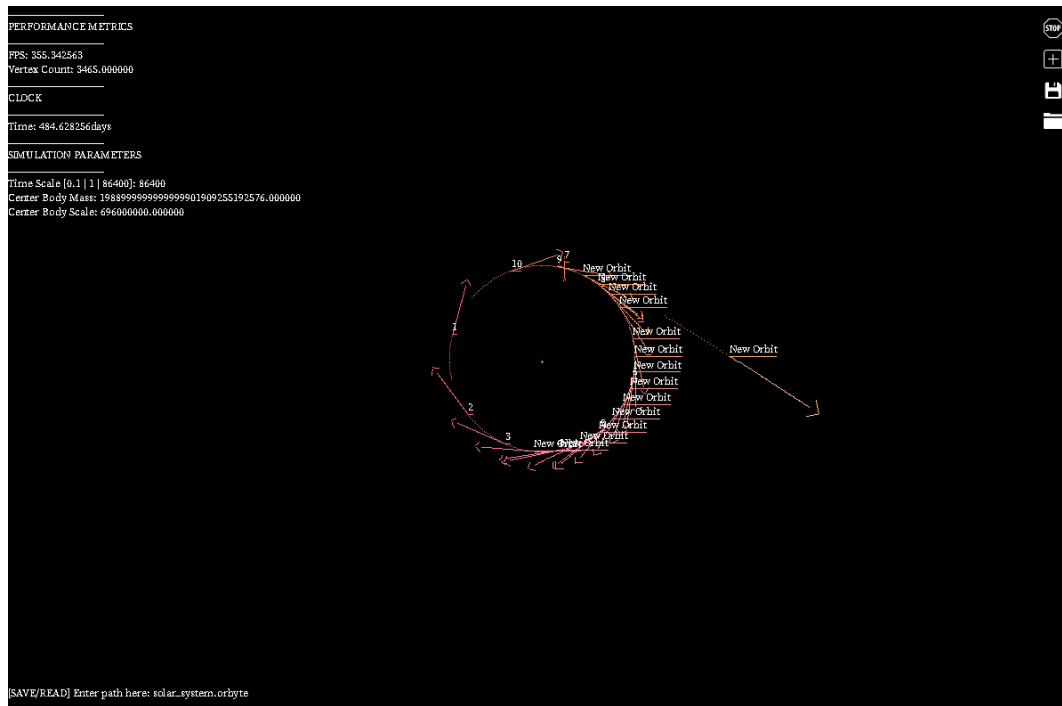
Frames Per Second: 355
Memory Usage: 69MB

Note: All orbiting bodies affect each other, hence the stray orbits due to instantiation too close to one another.

50 Orbits
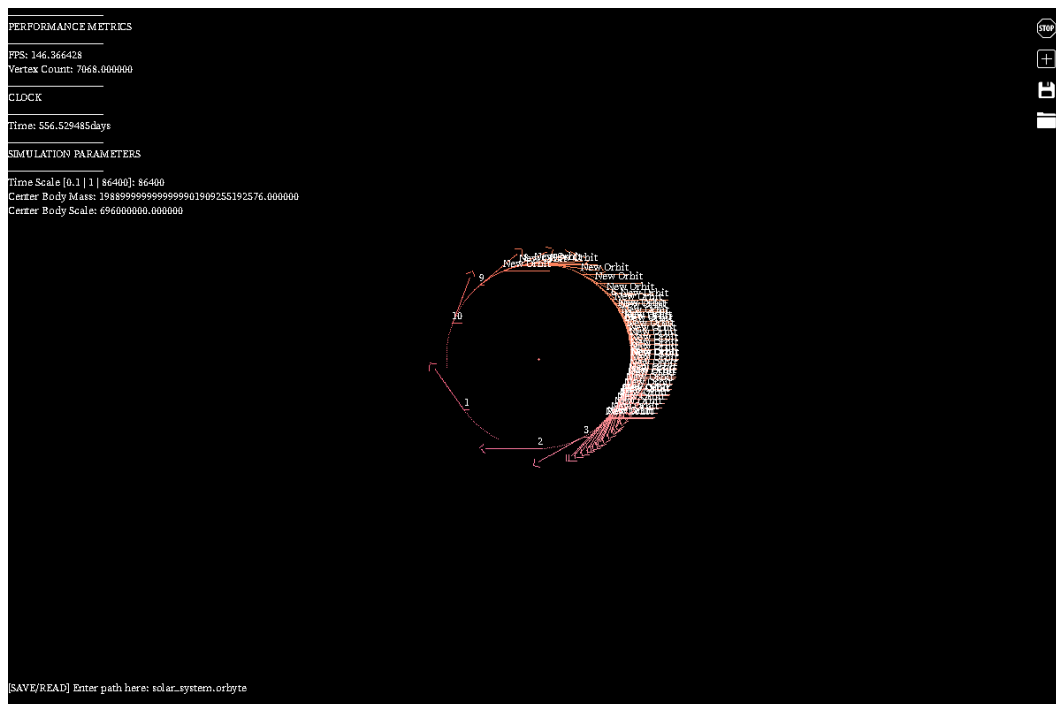
Frames Per Second: 146
Memory Usage: 93MB

100 Orbits
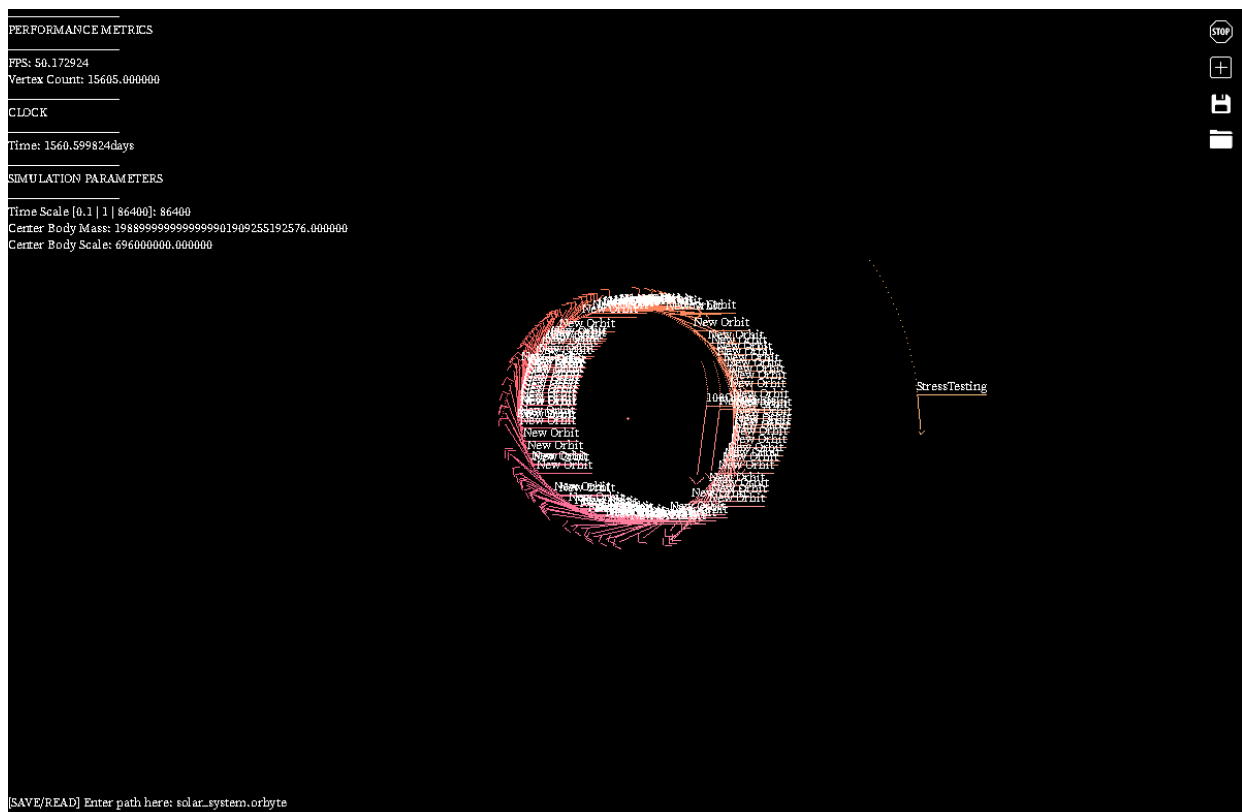
Figure 17 : Orbyte simulating 100 orbits.

Video Evidence: https://www.youtube.com/watch?v=a-to3tejO4Y

Frames Per Second: 50
Memory Usage: *Not measured*

The above image shows Orbyte simulating 100 1st Generation orbits at 50 frames per second. Every update, 100 bodies must all orbit the sun and orbit each other. Paths deviating from the circle, as shown in the image and in the video, are caused by the gravitational attraction of the bodies on each other (with some being instantiated close enough together to cause larger deviations).

Iterative Development Process

This section explores the process of developing Orbyte, and is supplementary to the Technical Solution & Testing sections. The aim of including this section in the write-up is to demonstrate the process of repeated refinement, optimisation and improvement in programming the project. Potential optimisations and modifications that would have been made, given more time, will be discussed within the **EVALUATION**.

Each of the following sections will discuss a particular problem or "bug" encountered when designing Orbyte, and establish how it was fixed. It is not possible to cover every improvement and patch made when developing this project, however some of the most interesting ones have been included.

Rotation issues

```
+    float x, y, z;
     rad = rot.x;
-    point.y = std::cos(rad) * point.y - std::sin(rad) * point.z;
-    point.z = std::sin(rad) * point.y + std::cos(rad) * point.z;
+
+    x = point.x;
+    y = point.y;
+    z = point.z;
+
+    point.y = (std::cos(rad) * y) - (std::sin(rad) * z);
+    point.z = (std::sin(rad) * y) + (std::cos(rad) * z);
+
+    x = point.x;
+    y = point.y;
+    z = point.z;

     rad = rot.y;
-    point.x = std::cos(rad) * point.x + std::sin(rad) * point.z;
-    point.z = -std::sin(rad) * point.x + std::cos(rad) * point.z;
+    point.x = (std::cos(rad) * x) + (std::sin(rad) * z);
+    point.z = (-std::sin(rad) * x) + (std::cos(rad) * z);
+
+    x = point.x;
+    y = point.y;
+    z = point.z;

     rad = rot.z;
-    point.x = std::cos(rad) * point.x - std::sin(rad) * point.y;
-    point.y = std::sin(rad) * point.x + std::cos(rad) * point.y;
+    point.x = (std::cos(rad) * x) - (std::sin(rad) * y);
+    point.y = (std::sin(rad) * x) + (std::cos(rad) * y);
```

Figure 18 : Changes made to the rotation method to fix the miscalculation of positions when rotating

The rotation method applies a rotation matrix to a 3D coordinate so that it is rotated around a centre point or "centroid".

The issue seen due to the incorrect application of the rotation matrix, was a transform alike to a "shear" occurring during rotation. This was ultimately due to the x, y, z values used as the 3D coordinate not representing the original point. The values were being changed each time the matrix was applied for each axis, leading to incorrect calculations.

The Git Diff shown above shows the change made. The issue was fixed and rotation now works as intended.

Instantiating Orbits

Due to how object instantiation is handled in C++, and how vectors store data, memory issues arise from unknowingly instantiating a class multiple times.

The orbiting_bodies vector attribute within the Simulation class used to contain instances of Body objects. This was not the intended design as it led to a Body being instantiated once, before being instantiated again via a copy constructor when added to the back of the "orbit

queue" or vector. This caused excessive use of memory and damaged the performance of the simulation, while also causing confusion when accessing orbits via the GUI due to the multiple instances of any one orbit.

This problem was fixed by changing the orbiting_bodies vector to use pointers to newly instantiated orbit bodies, such that there is only ever one instance of any one orbit at runtime.

```
- std::vector<Body> orbiting_bodies;
+ std::vector<Body*> orbiting_bodies;
```

Figure 19 : Showing change from storing a vector of objects to a vector of pointers

Performance Problem When Rendering Geometry

When adding a pixel to the draw queue, Graphyte checks that the attempted draw is within the screen dimensions. Previously, this only checked that the pixel was **less than** the screen width and height, despite the coordinate system implemented taking the **centre of the screen to be the origin.**

This led to an issue where looking at an object such that it filled the whole screen, clipping to the left and bottom of the screen, caused the performance of the simulation to plummet. This is because many pixels that were not on the screen were being drawn.

This bug was fixed by taking the absolute value of the x and y coordinates and comparing them to **half** the screen dimensions, so that only valid pixels were drawn.

```
void pixel(float x, float y)
void pixel(int x, int y)
{
        if (x < SCREEN_WIDTH && y < SCREEN_HEIGHT)
        if (std::abs(x) < (float)(SCREEN_WIDTH / 2) && std::abs(y) < (float)(SCREEN_HEIGHT / 2)) //abs value for negative values... that took so long to find.
```

Figure 20 : Git Diff showing new conditions that account for negative x and y values

Stuttering When Camera Following Orbit

When the camera was set to follow an orbit body there was an issue where the camera was stuttering and the movement was jarring instead of smoothly following the orbit. This was ultimately due to the camera being moved before the body was updated. This has been fixed (as shown below) and yields smooth camera movement when following an orbit.

```
for (Body* b : orbiting_bodies)
{
    b->Update_Body(deltaTime, time_scale, &orbiting_bodies); // Update body

    if (b->snap_camera)
    {
        vector3 cam_pos = b->Get_Position();
        gCamera.position.x = cam_pos.x;
        gCamera.position.y = cam_pos.y;
    }
    b->Draw(graphyte, gCamera); // Draw the body
}
```

Evaluation

Orbyte was intended to be an orbit simulation designed for GCSE and A Level students, enthusiasts and teachers. At its core was the intended use case of demonstrating: the "orbits of planets and satellites", how "satellites stay in orbit" and "how the speed of a satellite affects its radius" [ANALYSIS : BACKGROUND INFORMATION]. Ultimately, this has been accomplished. The simulation is sufficiently accurate, visually engaging, performant and educational. It demonstrates key orbital principles such as the inverse square law of forces acting between bodies in orbit, and provides a "sandbox" environment in which any orbits, from the earth around the sun, to the moon or ISS orbiting the earth, can be simulated. Judging the simulation holistically and qualitatively, it has met every objective and condition initially stipulated for "success", as is reflected in this section.

Review against objectives

The following table is a copy of the Objectives table, with an added "evaluation" column. Each objective is considered in comparison to the current functionality of Orbyte.

COMPLETED : Objective completely achieved, with respect to the objective description and performance criteria.

No.	Objective	Performance criteria	Evaluation
Version 0.0	**Fundamental Setup**		Version implemented successfully.
0.0.1	Create project and set up using SDL.	Successfully setup SDL dependencies. Compile with no errors.	Orbyte uses SDL and can be built with no errors. **COMPLETED**
0.0.2	Divide initialization steps into separate procedures and draw window.	Window is drawn and can be closed.	Initialization of SDL and Graphyte is handled by an individual method that is called by the main application method. Window is drawn and functions as expected. **COMPLETED**
0.0.3	Get User Keyboard Input and log it to console	User can press keys and corresponding debug message will be outputted to console.	Switch statement within application mainloop handles user input and registers expected key-presses successfully. **COMPLETED**
Version 0.1	**Orbit Object Implementation**		Version implemented successfully.
0.1.1.0	Set up Central Object Class (Accessors and Placeholder Methods)	Debugging procedures and Accessor methods within the class to access and display class attributes.	Simulation has a central body, functioning as the "largest" or most massive body in a system. This object is the fundamental source of gravitational attraction within the simulation. **COMPLETED**

0.1.1.1	Set up Orbiting Body Class (Accessors and Placeholder Methods)	Debugging procedures and Accessor methods within the class to access and display class attributes. Must follow good OOP practices.	Orbit bodies form the basis of every simulated orbit. Handling their own update methods and any child "2nd generation orbits." The class contains multiple key attributes and corresponding accessor / mutator methods, following good OOP convention. **COMPLETED**
0.1.1.2	Set up child class for satellites (Accessors and Placeholder Methods)	Debugging procedures and Accessor methods within the class to access and display class attributes.	**Satellite** inherits from **Body** and successfully demonstrates polymorphism through the overloading of inherited methods. **COMPLETED**
0.1.2.0	Implement Update Method for Orbit Body Class and Satellite Class	Mostly comments for what procedures need to occur each simulation step. Connect internal methods that should be called every frame, so that the entire update can be handled via one public method.	Orbit Body update method present and handles all internal processes that need to occur every frame. Calls RK4 step so that position can be calculated as a function of time, and updates body mesh for graphical representation. Satellite inherits the Update method from Body and overrides it, making amendments to the code for satellite functionality. **COMPLETED**
0.1.3.0	Create Simulation Class with attributes and accessors	Should contain necessary attributes, including a queue of orbiting bodies, a central body and various parameters. Implement all necessary methods / placeholders and include comments.	Simulation class successfully set up. Only one is instantiated during the application's execution. Attributes include an orbit body vector, a central body and necessary simulation parameters such as time scale & screen dimensions. **COMPLETED**
0.1.3.1	Outline Simulation Class Update Method	Method in place to handle entire application update.	Also referred to as the "application mainloop." Handles all frame-to-frame processes of Orbyte, from calling the update method on Orbit Bodies, to calling Graphyte's draw method. The simulation class encapsulates the entire functionality of Orbyte. **COMPLETED**
Version 0.2	**Orbital Simulation Mathematics**		Version implemented successfully.
0.2.1.0	Implement	2nd ODE solver set up	Runge-Kutta 4 underpins the

	Runge-Kutta 4 step in a procedure.	so that orbits can be simulated in realtime (1 second = 1 second). Using mathematics covered above.	simulation. Without the second order ordinary differential equation solver calculating position as a function of time, the simulation's mechanics and graphics would not be possible. It successfully simulates orbits in realtime, as well as larger, smaller or even negative time scales. **COMPLETED**
0.2.2.0	Integrate new RK4 method into the main program update procedure.	Connect RK4 method with Body update method and implement means to update all simulation bodies via simulation mainloop.	RK4 implementation used to calculate position as a function of time for bodies in orbit around the central body, and 2nd generation orbits around their parents and the central body. **COMPLETED**
0.2.3.0	Create Abstract Data Type: Vector3	Data type with accessor methods for x, y, z.	Vector3 struct implemented and used heavily throughout the simulation for calculations involving 3D vectors. **COMPLETED**
0.2.3.1	Create Magnitude Function in Vector Class	Function returns magnitude of the Vector.	Vector3 struct as parameter to the magnitude function. Used frequently throughout the simulation calculations and graphical representation of orbits. **COMPLETED**
0.2.3.2	Create Normalize Function in Vector Class	Function returns the normalized vector.	Vector3 struct as parameter to the normalize function. Returns a vector of magnitude 1 in the direction of the specified vector3. **COMPLETED**
0.2.3.3	Overload operators for Vector3 struct	Implement vector addition, scalar-vector multiplication, dot product.	Vector3 struct overloads many core c++ operators for usage in mathematical calculations. Extremely useful throughout the simulation. **COMPLETED**
Version 0.3	**Simulation Step**		Version implemented successfully.
0.3.1.0	Configure simulation class such that only one orbiting body is in the simulation queue. To sense-check results, make the parameters of the central	Earth instantiated at the correct distance from the sun, with correct velocity. Calculate orbit period (projection) and sense-check results: result should have error under 2%.	Earth can be instantiated such that it is the correct distance away from the sun (verified via debug console output and period calculations). Added to the list of orbits successfully. **COMPLETED** *Simulation queue: **PARTIALLY COMPLETED?** Throughout the design stage and within the technical solution, the "orbiting_bodies" vector*

	body that of the sun and the parameters of the orbiting body that of the earth.		*of orbits has been referred to as a queue. It has been justified earlier, however a textbook "queue" such as a circular queue has not been used on account of it not being performant to repeatedly dequeue and requeue bodies that are known to remain in the simulation between updates. Nonetheless, the vector is used as first-in, first-out and so is referred to as a queue.*
0.3.2.0	In Orbital Body Update Method, pass parameters to RK4 step and start to calculate position as a function of time. (Note this will be modified time, depending on the time-step parameter of the simulation.)	Debug methods each update to confirm values are changing, verification of correct calculation will be done at a later stage.	Discussed above. The **Body** update method calls the protected rk4 step that calculates the body's position as a function of time. It successfully does so and has been demonstrated in testing to do so accurately and performantly. **COMPLETED**
0.3.3.0	Write methods for the Orbital Body Class to output the current position.	Find the time period of the earth's orbit in the simulation and compare to known values. Note the error (if any), and then refer to previous steps to fix bugs (if any).	Correctly calculates the period of orbits. Verified through testing and instantiation of known orbits such as those of the planets in our solar system, and comparing both projected orbit periods and simulated orbit period (the time in-simulation that a complete orbit takes) to the known values. Minimal error seen. **COMPLETED**
0.3.4.0	Simulation should have acceptable performance for reasonable numbers of orbits.	25 orbits in the simulation should have at least 30 FPS.	The simulation is very performant. As shown in the **STRESS TESTING** section and later discussed in evaluation. **COMPLETED**
Version 0.4	**Graphics I**		Version implemented successfully.
0.4.1.0	Create method to draw a pixel to the screen given an x and y coordinate, using the SDL libraries.	Pixel drawn at expected location.	Graphyte is capable of drawing a pixel anywhere on the screen. **COMPLETED**

0.4.2.0	Create method to draw a line of pixels between two points.	Draw a 3D shape on screen.	Graphyte is capable of drawing a collection of pixels, generated from lines and points. As such it is able to draw 3D geometry such as the diamond used to represent orbit bodies, or a cube. More complex geometry is possible through a more detailed mesh. **COMPLETED**
0.4.3.0	Create method to rotate a collection of 3D points about a centroid.	Rotating 3D shape visible on screen. Constant 3D rotation over time (via update method) with no scaling / distortion.	Rotation matrix successfully applied to a point in 3D space so that shapes can be rotated every frame. **COMPLETED**
Version 0.5	**Satellite Child Class**		Version implemented successfully.
0.5.1.0	Instantiate one satellite in the simulation class main method. Set it's "parent" to be the earth and set its parameters to be similar to that of the moon. (*Note: We are presently forming a reductive version of our solar system, with only the sun, earth and moon, so that we can sense-check the results of our simulation.*)	Debug Methods to confirm attributes are set correctly.	The functionality of satellites exceeded expectations. Throughout their implementation the simulation struggled to accurately simulate their orbits and it was common for satellites to escape the orbit of their parents. At lower time-scales the simulation is more accurate, and therefore the orbit of satellites is more reliable. In testing, the moon was instantiated to orbit the earth and it was instantiated correctly, with an accurate orbit period. Ultimately the satellite class has been a success testament to the strength of the simulation. More can be done to improve their simulation. Perhaps calling the rk4 multiple times to interpolate between larger steps caused by higher time-scales will fix inaccuracies, however this will come with a very large performance overhead for larger time-scales. **COMPLETED**
Version 0.6	**Graphics II**		Version implemented successfully.
0.6.1.0	Draw the central body to the screen using custom graphics implementation.	Have a 3D shape drawn to the screen at the position corresponding to the origin.	Central body drawn to screen as a diamond shape, used also for 1st generation orbits. **COMPLETED**
0.6.2.0	Draw Orbiting bodies to the	Have a 3D shape drawn to the screen	Orbiting bodies drawn to the simulation window as 3D diamonds.

	screen using the custom graphics implementation. Have the draw function in the update method for the orbiting body class.	corresponding to the current position of the orbiting body. Drawn every frame.	Bodies able to be moved over the course of their orbit, frame-to-frame moved to the position calculated by the ODE solver. **COMPLETED**
0.6.3.0	Draw Satellites.	Have a 3D shape drawn to the screen corresponding to the current position of the satellite. Drawn every frame.	Satellites (2nd generation orbits) represented by a cube that is successfully drawn to the screen. **COMPLETED**
Version 0.7	**User Interface I**	**Draw User Interface using a library for GUIs in SDL.**	Version implemented successfully.
0.7.1.0	Draw sidebar with placeholder text corresponding to parameters for the simulation.	Text is drawn to the sidebar, occupying a portion of the side of the screen.	Simulation sidebar forms a crucial part of the GUI. Seen throughout all tests in the Testing section, the simulation parameters, clock and performance metrics are always visible on the left hand side of the screen. **COMPLETED**
0.7.2.0	Add Placeholder buttons	Clicking button outputs a debug message to console.	Buttons ultimately implemented successfully. The placeholders and debug messages proved useful in finding problems when not registering button presses etc. **COMPLETED**
0.7.2.1	Add Button to instantiate a new orbit body	Clicking button instantiates a new orbit with default parameters.	New orbit body button seen in the top right of the simulation. Proved to work as expected during testing. Vital part of the user interface and it gives users the freedom to add as many orbits as they want to, contributing to Orbyte's "sandbox" nature. **COMPLETED**
0.7.2.2	Add Button to pause simulation	Clicking button pauses simulation.	Used to stop the simulation from updating the position of orbit bodies and halt the clock's progression, so that changes can be made to the configuration of the simulation before it is resumed. Provides good structure to making changes to the simulation: 1. User pauses simulation. 2. Changes parameters.

			3. Resumes. This avoids any problems with changing the radius of the orbit and not having time to change the velocity before the orbiting body moves. By pausing the simulation, multiple fields can be edited before the next step is calculated. This button is possibly the most frequently used in the simulation. **COMPLETED**
0.7.3.0	Add a panel that functions as an "inspector." *This will describe attributes of the object currently in focus.*	Placeholder panel.	The inspector is what sets Orbyte apart from a demonstration. It is no longer a display of orbits, as the inspector turns it into a tool. By displaying orbital characteristics such as velocity, radius, acceleration and mass, the simulation provides more possibility for configuration and potential complexity than any other seen in the analysis section. **COMPLETED**
Version 0.8	**User Interface II**		Version implemented successfully.
0.8.1.0	Connect simulation to the User Interface. UI should access and display orbiting body class attributes.	See object attributes shown for 1 orbiting body.	Inspector shows parameters for selected orbit body. **COMPLETED**
0.8.2.0	Allow user to interact with any orbiting body such that its information is shown on the extended UI.	Inspector window shows selected object's properties	With multiple bodies in orbit, the user is able to click one of the bodies in orbit in order to open its inspector. This way different orbits can be inspected, whilst being simulated together. **COMPLETED**
0.8.3.0	Add input fields to general simulation parameters.	Able to change simulation parameters via input fields.	Crucially, in order to interact with the simulation, the user must be able to configure parameters via input fields. The user is able to change time scale and the mass or scale of the centre body. **COMPLETED**
0.8.3.1	Add input fields to inspector panel for orbit	Able to change orbit parameters via input fields.	The greatest level of customisability in an orbit simulation comes from being able to change the attributes of

	bodies.		a body in orbit: its position in metres, its initial velocity, its size and its name. Orbyte facilitates all of this through input fields in its inspector, that can be opened by selecting a body in orbit. **COMPLETED**
0.8.3.2	Add Button within inspector to instantiate satellite orbit around current body.	Clicking button instantiates a new satellite around current body.	2nd Generation orbits are instantiated via a 1st generation orbit body's inspector. The "add satellite" button is identical to the main "add orbit" button, except under the inspector. This facilitates adding orbits such as the moon, around the earth, with the earth and moon both orbiting the sun. **COMPLETED**
0.8.3.3	Add Button within inspector to reset selected orbit.	Clicking button resets orbit to initial parameters.	Under the orbit body inspector, it is possible to reset the orbit to its last configured initial position and velocity. This is helpful in case any unexpected behaviour develops that needs to be undone. **COMPLETED**
0.8.3.4	Add Button within inspector to delete selected orbit.	Clicking button removes body from orbit queue. Effectively deletes the orbit.	The orbit body inspector contains a button used to delete the selected orbit. Done by removing the body from the orbit queue and not rendering it. This is helpful as it removes clutter from the simulation and frees up processing resources as fewer bodies have to be updated each update. **COMPLETED**
0.8.4.0	Add method to display orbit object vector attributes, such as Velocity and Force. (Including option to hide these.)	Arrows are draw to the screen representing the direction and magnitude of the vectors.	Arrows are incredibly useful, both for debugging the simulation calculations by illustrating the vectors calculated as a result of the rk4 step, but primarily utilised as a part of the UI to visualize what is affecting the journey of a body in orbit. The acceleration of an object is drawn as a double headed arrow and the velocity is shown as a single headed arrow. **COMPLETED**
0.8.5.0	Add method to interact with orbit body such that the camera locks to it and follows its position around the simulation.	Camera's centre	Due to the "to scale" nature of the simulation's graphical representation of orbits, it is helpful to be able to follow an object in orbit with it at the centre of the screen, so it can be zoomed in on. By right clicking an orbit body, it is snapped to the centre of the screen. This makes it easier to

			view the geometry of the orbit body and its satellites' paths. The only issue with the current system is that the follow method only facilitates looking at the simulation along the z axis. Rotating the simulation with the arrow keys leads to the camera not following the orbit body. This could be fixed given more time by only using the object's screen space position and then converting that to a world space position for the camera. **PARTIALLY COMPLETED**
Version 0.9	**Data Storage**		Version implemented successfully
0.9.1.0	Write simulation data to storage.	Show representation of simulation data in storage system in console.	Simulations can be saved and written to a .orbyte file so that they can be opened another time. **COMPLETED**
0.9.1.1	Read simulation data from storage.	Show read data from storage solution in console.	Simulations can be read from a .orbyte file so that previous configurations and simulation states can be accessed and reopened. Especially useful for "templates" that can be made by one user and used by another. **COMPLETED**
0.9.2.0	Add ability to create new simulations & read / write to them.	Show new simulation in storage.	Path input field specifies which file to write to / create, and save button writes to that path. **COMPLETED**
0.9.2.1	Add input method to select simulations to read from / save to.	Use input method to save to and read from a simulation.	Path input field specifies which file to read from and open button reads from the file. **COMPLETED**

Analysis of independent feedback

Key User

When showing the key user, Mr Rice (see **ANALYSIS**), the current state of the simulation, he was impressed by the degree of sophistication, configurability and 3D nature of Orbyte. As discussed earlier in the **QUALITATIVE TESTING** section, the end user was confident that Orbyte could be used in a classroom to verify the relationships and principles covered at A

Level and GCSE. When shown that all orbits affect each other and it is possible to instantiate satellites that orbit 1st Generation Bodies, the end user praised the intricacy of the simulation. Despite the version that was shown to them containing a few bugs at the time, they expressed confidence in the simulation and verified that simulated orbit behaviour was as they would have expected to see.

Test Users

The following shows the results of a questionnaire given to multiple physics students after they tested the simulation. 10 Students tested Orbyte in total and their responses to the questions are shown below:

At first glance, do you understand what is happening in the simulation?

10 responses
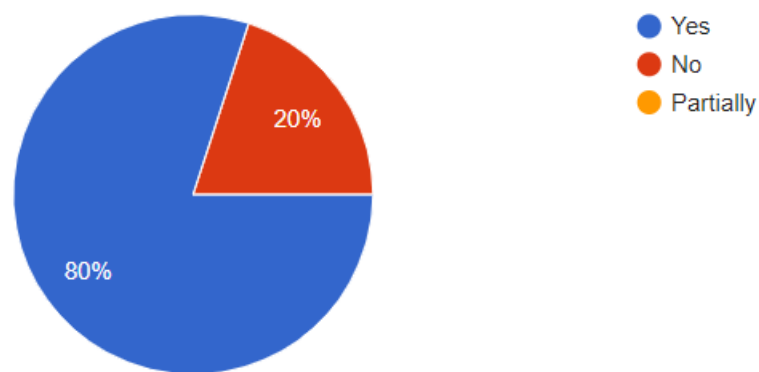
Figure 21 : Pie Chart. 80 of users immediately understood the simulation.

This is a reassuring statistic as it demonstrates the clarity of representation of orbits in Orbyte. 80% of users understood what was happening as soon as they started using the simulation.

Would you describe the window as "cluttered"?

10 responses
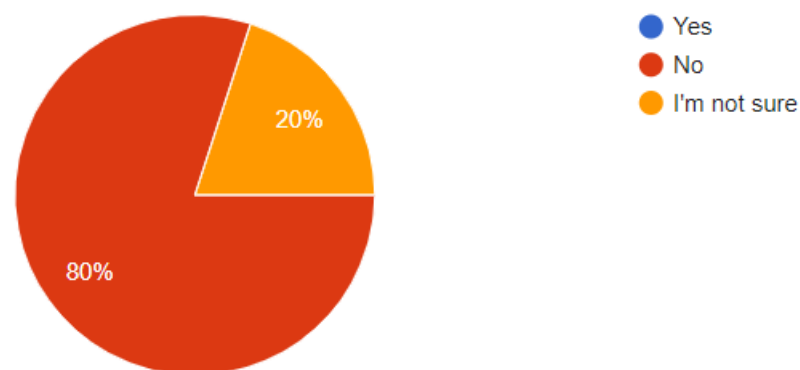
Figure 22 : Pie Chart. 80% of users would say the window was not cluttered. No one said it was.

One of the observations made of other simulations was that the UI was "cluttered" or invasive and left little screen-space left for the simulation itself. Orbyte was designed such

that the UI was as minimally intrusive as possible. The one caveat to this was the number of digits used to represent large numbers, and some users suggested that bigger number be represented in a more compact format.

On a scale of 1 to 10, how easy is it to add a new orbit to the simulation?
10 responses
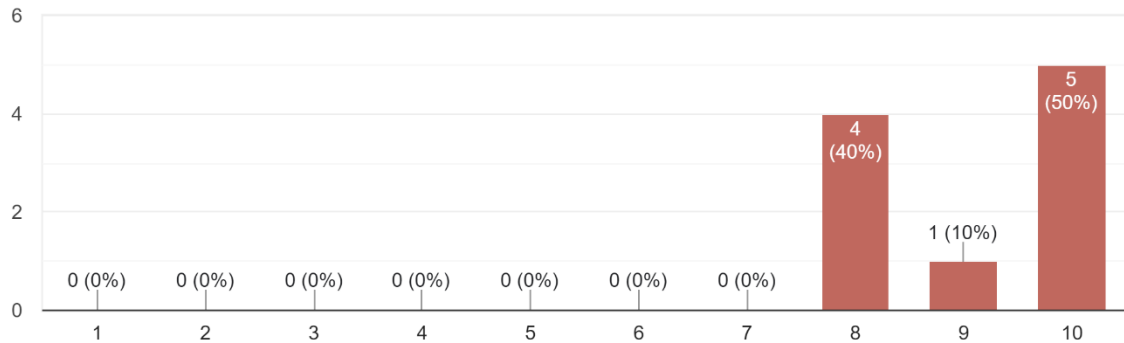
Figure 23 : Bar chart showing that users found it easy to add a new orbit to the simulation.

For use as a tool in the classroom and as an orbit simulation in general, a crucial feature is the ability to add new orbits. One button click instantiates a default "new orbit" into the simulation. All users found it reasonably easy to do so.

On a scale of 1 to 10, how easy is it to configure the parameters of the simulation?
10 responses
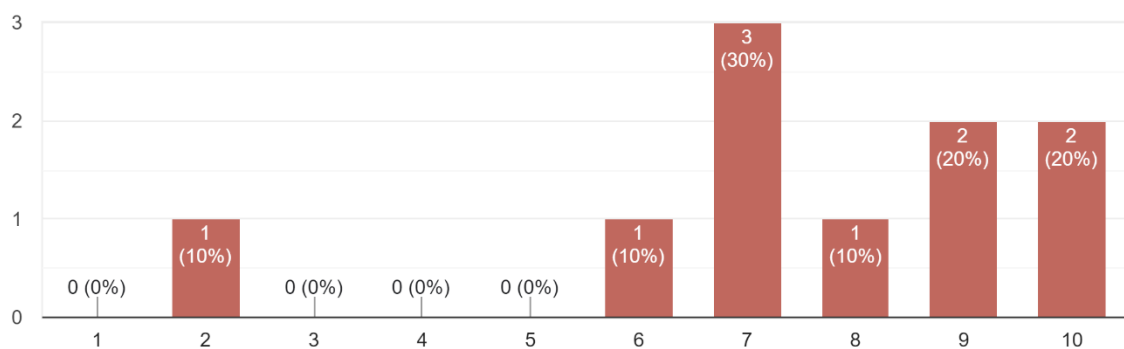
Figure 24 : Bar chart showing that most users found it easy to configure simulation parameters

To make Orbyte a viable tool for demonstration and practical work it is necessary to make it such that editing the parameters of the simulation is as easy as possible. Most users found it easy to change simulation parameters, such as time scale.

On a scale of 1 to 10, how easy is it to configure the parameters of an orbit?
10 responses
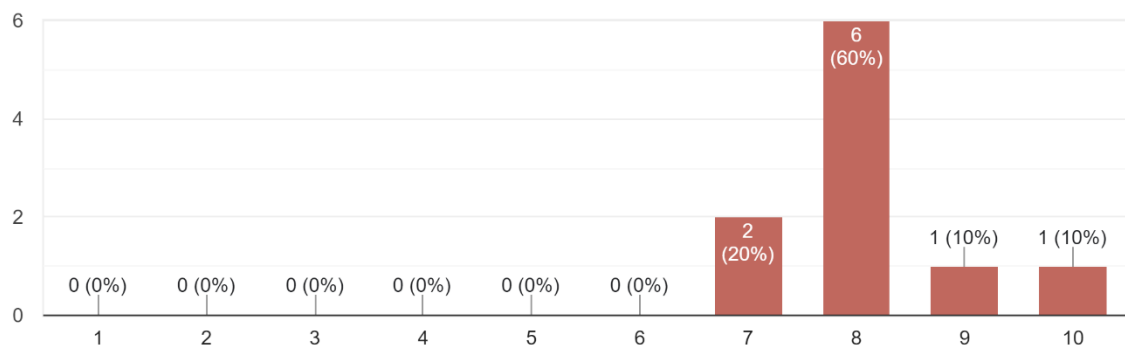
Figure 25 : Bar chart showing that users found it easy to configure the parameters of a specific orbit via the orbit inspector.

The orbit inspector allows a user to edit a specific orbit's properties, and users found it easy to use. This demonstrates that though the simulation is described as "sophisticated" and sufficiently "complex", users are confident in changing the variables of an orbit and seeing its affects unfold in the simulation before them.

On a scale of 1 to 10, how easy is it to read the text on the user interface?
10 responses
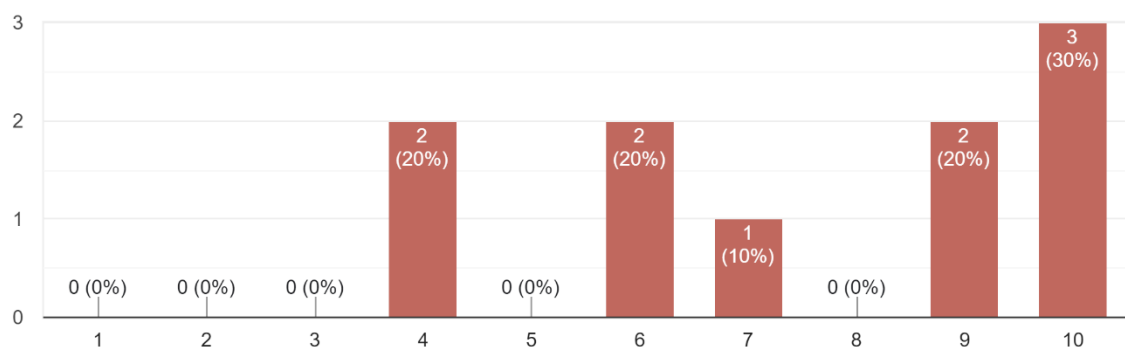
Figure 26 : Bar chart showing that users were overall unsure as to whether the text was easy to read.

The text in Orbyte was easy to read for some and difficult for others. In order to improve accessibility to the simulation for those who might find the text hard to read, a font-size option could be included in a settings panel.

Therefore would you describe Orbyte as intuitive and navigable?
10 responses
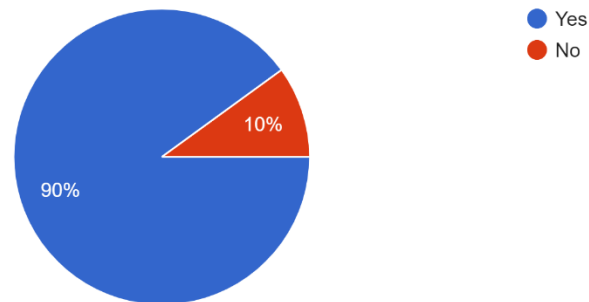
Figure 27 : 90% Of users would describe Orbyte as intuitive and navigable.

User-experience is one of the greatest priorities for an educational simulation, and as such it was paramount that users find the simulation intuitive and easy to navigate rather than awkward and confusing. Despite its complexity, Orbyte has evidently achieved this, as supported by 90% of users that tested the simulation.

What is the maximum number of orbits you would most likely use (approximately) in a simulation?
10 responses
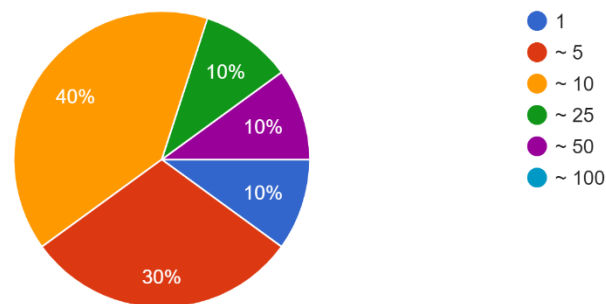
Figure 28 : The most common number of orbits a user would instantiate in the simulation would be ~10.

40% Of users would instantiate approximately 10 orbits in the simulation. Our system has 8 planets in orbit so this rough estimate for the average number of orbits is sensible. The simulation has already been proven performant up to 100 orbits and no user suggested that they would simulate that many. Orbyte is therefore as performant as it needs to be and the time invested in optimisation and creating Graphyte was well spent.

Do you prefer Orbyte to other simulations?
10 responses
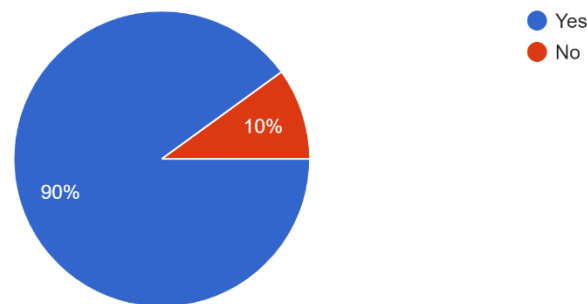
Figure 29 : 90% of users prefer Orbyte to other orbit simulations.

When asked if users preferred Orbyte over other orbit simulations, such as those discussed in the **ANALYSIS** section, 90% said they preferred Orbyte. Test users were then asked to justify their choice by stating their favourite feature in Orbyte, some of which are listed below.

- "How easy it is to get used to."
- "I like how the planet leaves an aesthetic trail."
- "Can simulate real life situations."
- "(I like that there are) moon simulations as well."
- "(The 3D graphics) look like the original Elite[16]."
- "3D graphics and rotation."
- "Adding planets."
- "(Being able to) change the camera angle with (arrow) keys."

The users were given minimal instructions before using the simulation and all picked up its controls quickly. The 3D graphics were very popular and users enjoyed the amount of freedom the simulation gave them.

Analysis of simulation performance

Orbyte was built from the ground up using SDL so that it could be as performant as possible. In its current state (Version 0.9), the simulation is fast and capable of simulating multiple orbits concurrently. The following evaluates the performance test covered in **STRESS TESTING**.

> *"Motion pictures, TV broadcasts, streaming video content, and even smartphones use the standard frame rate of 24fps." ~ [ADOBE]*

The "minimum" acceptable FPS intended for the simulation was 30 frames per second for 25 orbits, as established under the **OBJECTIVES**. The performance of the simulation greatly exceeded this benchmark and for 25 orbits the simulation's average frames per second was 355.

---

[16] The user was referring to the original Elite video game.

This level of performance is especially impressive when considering that every orbit in the simulation, affects every other orbit. This fps would be the same for any value for time scale, and seeing as the accuracy of the simulation is dependent on the time scale used, the simulation can achieve very accurate orbit projections with high performance, allowing for more orbits to be simulated with the same "smoothness" of a high frame per second count. This degree of performance is especially reassuring as it confirms that the simulation will run well on lower tier hardware as well.

Our solar system has 8 planets in orbit around the sun and each one of those planets may have several moons. Ultimately, Orbyte would be able to simulate the whole solar system given the degree of performance demonstrated in testing. Most users would not instantiate more than 100 orbits and even if they did, the test with 100 bodies in orbit was running at 50FPS, a more than acceptable number of frames per second.

Analysis of simulation accuracy

As an educational simulation, accuracy was a priority for Orbyte. As with all orbital calculations, the more accurate the input data used, the better the output of the equation represents the orbit you are trying to model. The accuracy of Orbyte is largely therefore in the hands of the user, and the degree of precision they enter data to. The user is also entirely in control of the timescale, which effects the accuracy of simulated orbits.

Numerical Representation Limitations

Orbyte uses **double** as its datatype for numerical calculations. A **double** is a floating point numerical representation using 64bits, which is capable of a vast range of values but its relative error is dependent on the number of bits available for the mantissa.

> *"Floating-point types use an IEEE-754 representation to provide an approximation of fractional values over a wide range of magnitudes."* ~
> *[FLOATING POINT TYPES]*

Due to the nature of numerical representation using a finite number of bits, there is always an element of "approximation" for fractional values. This is seen especially for very large numbers where, due to the size of the exponent, the relative error of the representation can become larger for greater magnitudes of numbers.

The accuracy of the 64 bit representation is sufficient for the purposes of Orbyte, and has been demonstrated to correctly simulate the orbit of the earth, though for much larger orbits, inaccuracy will develop over time. This is ultimately not a problem for Orbyte. The simulation's remit was to simulate the orbit of a planet around a sun and the orbit of satellites around the earth, which this degree of accuracy has been proven to be sufficient for.

Time Scale Limitations

The orbit of the earth is 365 days, and in the default simulation, one of the recommended timescales is 86400, this is equivalent to 1 day per second. This time-scale does not cause

much error in the simulation of the orbit as the timescale is small compared to the period of the orbit. For shorter orbits, such as that of the moon around the earth with period 27 days. A timescale of 1/27 of the orbital period is a large step, and as such the simulation's approximation becomes much more inaccurate due to there being no interpolation between steps. It is therefore important to make sure that the time scale of a simulation is much less than the orbit periods you are trying to simulate, otherwise error will accumulate very quickly and deviate from expected results.

Potential improvements

Features

This section lists some of the features that would have been implemented if more time was available for the development of Orbyte, and perhaps will be implemented in the future.

Custom Geometry

Allowing the user to open custom 3D object data so that orbit bodies could have unique geometries. This would be implemented by allowing the user to use the path input and select a .fbx or similar file type that stores 3D geometry information. This would then be used to generate a mesh for the body that would be drawn by Graphyte. The systems facilitating this feature are already in place, there was just not enough time available to be able to fully implement it as it is not a core objective.

Settings

One of the biggest changes I would make to Orbyte, given more time, would be to add a pop-out settings window that a user could use to make changes to how the application functions. Variables in settings would include background colour, orbit colour, font size & font, frames-per-second cap and simulation window dimensions.

Quality Of Life

Quality of life changes are improvements that could be made to Orbyte that would improve the user experience. In the time available, these changes were not prioritised and as such have not been implemented. Given more time, the following is what would have been included.

Displaying numbers to fewer significant figures

The key user and multiple test users all suggested that the simulation should display large numbers with lots of trailing 0s in a more compact format. So that instead of "1000000" taking up screen space, the display would read "1E6". If the number "0.00001" was to be shown, it would be displayed as "1E-5" If more time was available for UI development, this would be implemented.

Camera usability

Users have remarked that it is not entirely clear where the axis of the simulation are after rotation the camera. A "gizmo" or small 3D representation of the axis could be added to the simulation so that users are aware of the current orientation of the camera with relation to the origin. A reset button could be added to a camera GUI panel so that the camera can be reset to the original position.

Input Fields

Test users have expressed that it is not clear when an input field is selected and when they are able to type in them. They also remarked that it would be helpful to have some way to know when an input has been rejected for being invalid, perhaps by resetting the input field or writing an error message to the screen.

User Guide

Test users have expressed that it would be helpful to have an in-depth guide explaining the simulations functionality. This could be accomplished via a pop-up window with a short description of the simulation, how it works and important information such as the accuracy vs time scale trade-off.

Optimisations

To increase the performance of Orbyte, increasing Frames Per Second and reducing memory usage, the following changes would have been made given more time.

Drawing Pixels

Graphyte draws pixels iteratively. Whenever a pixel is to be drawn, the "pixel" method is called and the point is added to a vector of all pixels pending display. When the main Graphyte "draw" method is called, this vector of points is iterated through and all points are drawn to the screen, one after another. Whilst being simple to implement and easy to debug, this is not the most efficient way to draw pixels, requiring lots of iteration and many draw calls.

The solution intended to be implemented, however postponed due to not being essential and a lack of sufficient time to change the code, was to write all pixels to an SDL texture and draw the texture in one go, instead of drawing every pixel one by one. This would be more performant and would decrease the performance overhead of the Graphics.

Orbit Inspector

In the current implementation of the orbit inspector, every orbit body has its own set of UI elements despite the inspector format being common between all bodies. This is inefficient as it requires multiple instantiations of UI elements, for every orbit body in the simulation. Therefore every orbit occupies a larger space in memory.

Given more time, this could be fixed by re-writing the inspector system such that there was one inspector GUI block, as an attribute of Graphyte, that all orbits would access when they

need to show their inspector, as only one inspector should be open at a time. This would reduce the amount of memory required per orbit simulated and improve the space complexity of the simulation.

Deleting Orbits

Currently removing an orbit from the simulation is not as optimal as it could be. This is due to the residual memory usage of a deleted orbit, despite it no longer being simulated and the additional iteration required in the application mainloop in order to "clean" the orbit queue.

In the current implementation, orbits marked "to be deleted" have their corresponding deletion flag set to true and in the next application update, are removed from the orbit queue. This removes them from the list of orbits to be updated and so the processing load is reduced and the body is no longer drawn to the screen. However, the GUI elements and other attributes are not completely deconstructed until the application is closed, leading to unnecessary memory usage when orbits are deleted.

This could be fixed, given more time, by properly utilising the orbit body deconstructor to remove all graphics attributes and UI elements, such that they are no longer taking up space in memory and freeing up resources for more new orbits.

Conclusion

In conclusion, Orbyte has successfully fulfilled all the initial objectives for an educational orbit simulation. The simulation is 3D, accurate, performant and demonstrates all the key principles of orbits necessary for GCSE and A Level physics. The simulation offers a great degree of freedom and initial feedback from test users and the key user has been reassuring, affirming Orbyte's potential to be used for demonstrative and practical purposes in a classroom.

References

NASA. 2022. *Space Debris and Human Spacecraft*. [online] Available at: <https://www.nasa.gov/mission_pages/station/news/orbital_debris.html> [Accessed 28 September 2022].

AQA. 2015. **A-Level Physics Specification.** [online] Available at: https://filestore.aqa.org.uk/resources/physics/specifications/AQA-7407-7408-SP-2015.PDF [Accessed 04 October 2022].

PHET Orbit Simulation [online] Available at: https://phet.colorado.edu/en/simulations/gravity-and-orbits [Accessed 04 October 2022].

SATVIS. Satellite Orbit Visualization. [online] Available at: https://satvis.space/

SATELLITE EXPLORER. Satellite orbit explorer. [online] Available at: https://geoxc-apps.bd.esri.com/space/satellite-explorer/# [Accessed 04 October 2022].

SDL_WIKI. Simple DirectMedia Layer documentation. [online] Available at:
https://wiki.libsdl.org [Accessed 04 October 2022].

LAZY_FOO. C++ SDL2 Tutorial for Game Programming. [online] Available at:
https://lazyfoo.net/tutorials/SDL/ [Accessed c.October 2022]

ADOBE. A Beginners Guide to Frame Rates in movies. [online] Available at:
https://www.adobe.com/creativecloud/video/discover/frame-rate.html [Accessed 14 March
2022]

FLOATING POINT TYPES. Built-in types (C++). [online] available at:
https://learn.microsoft.com/en-us/cpp/cpp/fundamental-types-cpp [Accessed 15 March 2022]

Appendix

Source Code

Orbyte_Prototype.cpp

```cpp
// Orbyte_Prototype.cpp : This file contains the 'main' function. Program exe
cution begins and ends there.
//  SDL + GUI library built for SDL

// https://lazyfoo.net/tutorials/SDL/index.php See this resource for fundamen
tal SDL setup

#include <iostream>
#include <string>
#include <SDL.h>
#include <stdio.h> //This library makes debugging nicer, but shouldn't really
 be involved in user usage.
#include <vector>
#include <numeric>
#include "vec3.h"
#include "OrbitBody.h"
#include "Camera.h"
#include <sstream>
#include <SDL_ttf.h>
#include <SDL_image.h>
#include "Orbyte_Data.h"
#include "Orbyte_Graphics.h"
#include "utils.h"

class Simulation
{
private:
        const double SCREEN_WIDTH = 1200; //Screen dimensions
        const double SCREEN_HEIGHT = 800;
        const int SCREEN_FPS = 500; //fPS
        const int SCREEN_TICKS_PER_FRAME = 1000 / SCREEN_FPS;
        const int MAX_FPS = 500; //FPS Cap
```

```cpp
        //Performance testing
        double frames_since_debug = 0;
        double time_since_debug = 0;

        //time scale
        double time_scale = 1;
        TextField* time_scale_text;

        //Globally used font
        TTF_Font* gFont = NULL;

        //Window
        SDL_Window* gWindow = NULL;

        //Camera
        Camera gCamera;

        //The window renderer
        SDL_Renderer* gRenderer = NULL;

        //Graphyte
        Graphyte graphyte;

        //Data Controller
        DataController data_controller;

        //Orbit Bodies
        std::vector<Body*> orbiting_bodies;

        //CB
        CentralBody Sun;

        //Runtime variables
        bool quit = false;
        SDL_Event sdl_event;

        //Current time start time
        Uint32 startTime = 0;
        Uint32 deltaTime = 0; // delta time in milliseconds
        double timeSinceStart = 0;

        //Path source for Orbyte Files
        std::string path_source;

        bool loadMedia()
        {
                //success flag
                bool success = true;

                //Open the font
                gFont = TTF_OpenFont("SourceSerifPro-
Regular.ttf", 12); //Open_My_Font
                if (gFont == NULL)
                {

        printf("Failed to load lazy font! SDL_ttf Error: %s\n", TTF_GetError(
));
                        success = false;
                }
```

```c
                return success;
        }

        //Initialize SDL and window
        bool init()
        {
                if (SDL_Init(SDL_INIT_VIDEO) < 0)
                {

        printf("ERROR INITIALIZING SDL | SDL_ERROR : %s\n", SDL_GetError());
                        return false;
                }

                //Creating the window

        gWindow = SDL_CreateWindow("Orbyte Prototype", SDL_WINDOWPOS_UNDEFINE
D, SDL_WINDOWPOS_UNDEFINED, SCREEN_WIDTH, SCREEN_HEIGHT, SDL_WINDOW_SHOWN);
                if (gWindow == NULL)
                {

        printf("ERROR CREATING WINDOW | SDL_ERROR: %s\n", SDL_GetError());
                        return false;
                }

                //Create the SDL Renderer
                gRenderer = SDL_CreateRenderer(gWindow, -
1, SDL_RENDERER_ACCELERATED);
                if (gRenderer == NULL)
                {

        printf("Renderer could not be created! SDL Error: %s\n", SDL_GetError
());
                        return false;
                }
                else
                {
                        //Initialize renderer color

        SDL_SetRenderDrawColor(gRenderer, 0xFF, 0xFF, 0xFF, 0xFF);
                } //Not loading PNGs because that'd be a pain to implement.


                //Initialize PNG loading
                int imgFlags = IMG_INIT_PNG;
                if (!(IMG_Init(imgFlags) & imgFlags))
                {

        printf("SDL_image could not initialize! SDL_image Error: %s\n", IMG_G
etError());
                        return false;
                }

                //Initialize SDL_ttf
                if (TTF_Init() == -1)
                {

        printf("SDL_ttf could not initialize! SDL_ttf Error: %s\n", TTF_GetEr
ror());
                        return false;
                }
```

```cpp
            //Load media
            if (!loadMedia())
            {
                    printf("Failed to load media!\n");
                    return false;
            }

            //Starting Graphyte

    if (!graphyte.Init(*gRenderer, *gFont, { SCREEN_WIDTH, SCREEN_HEIGHT,
 0 }))
            {
                    printf("Graphyte could not initialize!");
                    return false;
            }
            return true;
    }

    //Frees media and shuts down SDL
    void close()
    {

            TTF_CloseFont(gFont);
            gFont = NULL;


            //Destroy window
            SDL_DestroyWindow(gWindow);
            SDL_DestroyRenderer(gRenderer);
            gWindow = NULL;
            gRenderer = NULL;

            //Quit SDL subsystems
            TTF_Quit();
            IMG_Quit();
            SDL_Quit();
    }

    void commit_to_text_field() //Commit changes to active textfield
    {
            if (graphyte.active_text_field != NULL)
            {
                    graphyte.active_text_field->Commit();
            }
    }

    void close_planet_inspectors() //This is a rough implementation, but
its all I have time for.
    {
            for (Body* b : orbiting_bodies)
            {
                    b->HideBodyInspector();
            }
    }

    void recenter_camera() //Set the camera back to the center of the sim
ulation
    {
            gCamera.position.x = 0;
```

```cpp
                gCamera.position.y = 0;
                for (Body* b : orbiting_bodies)
                {
                        b->snap_camera = false; //Not tracking any orbits
                }
        }

        /*
                Handle mouse click
        */
        void click(int mX, int mY, bool left_click)
        {

        std::cout << "\nChecking for clickable @: " << mX << ", " << mY << "\
n";
                if (graphyte.active_text_field != NULL) //Disable text field
                {
                        graphyte.active_text_field->Disable();
                }
                graphyte.active_text_field = NULL; //Set to null

                for (TextField* tf : graphyte.text_fields)
                {
                        if (tf->CheckForClick(mX, mY))
                        {
                                graphyte.active_text_field = tf;
                                printf("\n YOU CLICKED A THING \n");
                                return;
                        }
                }

                for (FunctionButton* fb : graphyte.function_buttons)
                {
                        if (fb-
>CheckForClick(mX, mY, !left_click)) //Check for click also calls attached fu
nctions
                        {
                                return;
                        }
                }

                if (!left_click)
                {
                        recenter_camera(); //Recenter camera if right click
                }

                close_planet_inspectors(); //CLose all planet inspectors.
        }

        Uint32 Update_Clock()
        {
                Uint32 current_time = SDL_GetTicks(); //milliseconds
                Uint32 delta = current_time - startTime;
                startTime = current_time;
                return delta;
        }

        /*

        Remove any orbits to be deleted from the orbit queue before update
```

```cpp
        */
        void clean_orbit_queue()
        {
                // This is not as performant as I'd like it to be!
                int length = orbiting_bodies.size();
                for (int i = 0; i < length; i += 0) //This is an odd loop
                {
                        if (orbiting_bodies[i]->to_delete)
                        {

orbiting_bodies.erase(orbiting_bodies.begin() + i);
                                length -= 1;
                        }
                        else {
                                i++;
                        }
                }
        }


        vector3 calculate_centre_of_mass(CentralBody cb) //NOT USED
        {
                //A Level Further Maths: Mechanics
                double total_mass = cb.mass;
                vector3 com = cb.position;

                for (auto& b : orbiting_bodies)
                {
                        double mass = b->Get_Mass();

                        com = com + (b->Get_Position() * mass);
                        total_mass += mass;
                }

                com = com * ((double)1 / total_mass);
                return com;
        }


        void toggle_pause()
        {
                if (time_scale == 0)
                {
                        time_scale = 1;
                        time_scale_text->Set_Text(std::to_string((double)1));
                }
                else {
                        std::cout << "\nPaused Simulation\n";
                        time_scale = 0;
                        time_scale_text->Set_Text(std::to_string((double)0));
                }
                return;
        }


        // Add orbit with given OrbitBodyData
        void add_specific_orbit(OrbitBodyData data)
        {

        orbiting_bodies.push_back(new Body(data.name, data.center, data.mass,
 data.scale, data.velocity, Sun.mu, graphyte, false));
        }
```

```cpp
        // Add general orbit with generic parameters
        void add_orbit_body()
        {

        orbiting_bodies.push_back(new Body("New Orbit", { 0, 5.8E10, 0 }, 3.2
85E23, 2.44E6, { 47000, 0, 0 }, Sun.mu, graphyte, false));
        }

        void save()
        {
                OrbitBodyCollection obc; // Collection of orbits
                std::vector<std::string> to_save; // Name of orbits to save

                // Check that body is not about to be deleted before saving.
                for (Body* b : orbiting_bodies)
                {
                        if (!b->to_delete)
                        {
                                to_save.push_back(b->name); // Add name
                                obc.AddBodyData(b-
>GetOrbitBodyData()); // Add orbit data
                        }
                }

                // Encapsulate all simulation data

        SimulationData sd = { Sun.mass, Sun.scale, obc, gCamera.position };

                // Write to .orbyte file
                data_controller.WriteDataToFile(sd, to_save, path_source);
        }

        void open()
        {
                std::cout << "\nOpening File";
                // New Simulation Data

        SimulationData new_sd = data_controller.ReadDataFromFile(path_source)
;
                // Pause simulation
                time_scale = 0;

                // Load all parameters from SimulationData
                Sun.mass = new_sd.cb_mass;
                recalculate_center_body_mu();
                std::cout << "\n Sun mass: " << Sun.mass;
                Sun.scale = new_sd.cb_scale;
                regenerate_center_body_vertices();
                gCamera.position = new_sd.c_pos;

                // Clear Old Orbits
                for (Body* old_orbit : orbiting_bodies)
                {
                        old_orbit->Delete();
                }

                OrbitBodyCollection obc = new_sd.obc;
                std::vector<OrbitBodyData> orbits = obc.GetAllOrbits();
                // Instantiate Orbits
                for (OrbitBodyData orbit : orbits)
```

```cpp
                        {
                                add_specific_orbit(orbit);
                        }
                }

        void regenerate_center_body_vertices() //Regen centre body geometry
        {
                Sun.RegenerateVertices();
        }

        void recalculate_center_body_mu() // Recalculate mu of centre body
        {
                Sun.RecalculateMu();
                for (Body* orbit : orbiting_bodies)
                {
                        orbit->Set_Mu(Sun.mu);
                }
        }
public:

        /// <summary>
        /// Start the simulation
        /// </summary>
        /// <param name="argc"></param>
        /// <param name="args"></param>
        /// <returns></returns>
        int run(int argc, char* args[])
        {
                //Start up SDL and create window
                if (!init())
                {
                        printf("Failed to initialize!\n");
                }
                else
                {
                        // Name, Pos, mass, Radius, Velocity

        //Body mercury = Body("Mercury", { 0, 5.8E10, 0 }, 3.285E23, 2.44E6,
{ 47000, 0, 0 }, Sun, graphyte, false);

        //Body venus = Body("Venus", { 0, 1E11, 0 }, 6E6, { 35000, 0, 0 }, Su
n, graphyte, false);


        Body earth = Body("Earth", {0, 1.49E11, 0}, 5.97E24, 6.37E6, { 30000,
 0, 0 }, Sun.mu, graphyte, false);
                        std::cout << earth.DebugBody();
                        //Name: "Earth"
                        //Radius of orbit: 1.49E11
                        //Mass: 5.97E24
                        //Scale: 6.37E6
                        //T_Velocity: 30000ms^-1
                        //Mu: Sun's mu (See Analysis)

                        orbiting_bodies.emplace_back(&earth);


                        /*
                                SIMULATION PARAMETERS GUI INITIALIZATION
                        */
```

```cpp
                    GUI_Block Simulation_Parameters(vector3{ -
SCREEN_WIDTH / 2, SCREEN_HEIGHT / 2, 0 });


        Text* text_pm = graphyte.CreateText("_____\nPERFORMANCE
METRICS\n_____", 24);
                    Simulation_Parameters.Add_Stacked_Element(text_pm);


        Text* text_FPS_Display = graphyte.CreateText("FPS", 10);

        Simulation_Parameters.Add_Stacked_Element(text_FPS_Display);



        Text* text_Vertex_Count_Display = graphyte.CreateText("Vertices", 10)
;

        Simulation_Parameters.Add_Stacked_Element(text_Vertex_Count_Display);


        Text* text_cl = graphyte.CreateText("_____\nCLOCK\n_____
_____", 24);
                    Simulation_Parameters.Add_Stacked_Element(text_cl);


        Text* text_time_Display = graphyte.CreateText("Time: ", 10);

        Simulation_Parameters.Add_Stacked_Element(text_time_Display);


        Text* text_sp = graphyte.CreateText("_____\nSIMULATION P
ARAMETERS\n_____", 24);
                    Simulation_Parameters.Add_Stacked_Element(text_sp);

                    //Testing input fields I guess

        Simulation_Parameters.Add_Stacked_Element(graphyte.CreateText("Time S
cale [0.1 | 1 | 86400]: ", 10));
                    DoubleFieldValue TimeScaleFV(&time_scale);

        TextField* tf = new TextField({ 10,10,0 }, TimeScaleFV, graphyte, std
::to_string(time_scale));
                    time_scale_text = tf;
                    graphyte.text_fields.push_back(tf);
                    Simulation_Parameters.Add_Inline_Element(tf);


        Simulation_Parameters.Add_Stacked_Element(graphyte.CreateText("Center
 Body Mass: ", 10));

        DoubleFieldValue CentreMassFV(&Sun.mass, [this]() { this-
>recalculate_center_body_mu(); });

        tf = new TextField({ 0, 0, 0 }, CentreMassFV, graphyte, std::to_strin
g(Sun.mass));
                    graphyte.text_fields.push_back(tf);
                    Simulation_Parameters.Add_Inline_Element(tf);
```

```cpp
        Simulation_Parameters.Add_Stacked_Element(graphyte.CreateText("Center
 Body Scale: ", 10));

        DoubleFieldValue CentreScaleFV(&Sun.scale, [this]() { this-
>regenerate_center_body_vertices(); });

        tf = new TextField({ 0, 0, 0 }, CentreScaleFV, graphyte, std::to_stri
ng(Sun.scale));
                    graphyte.text_fields.push_back(tf);
                    Simulation_Parameters.Add_Inline_Element(tf);


                    /*
                            PATH TO OPEN FROM FILE
                    */
                    GUI_Block path_gui;
                    path_gui.position = { (-SCREEN_WIDTH / 2), (-
SCREEN_HEIGHT / 2) + 36, 0 };

        Text* path_input_prompt = graphyte.CreateText("[SAVE/READ] Enter path
 here: ", 12);
                    path_gui.Add_Stacked_Element(path_input_prompt);

        StringFieldValue path_to_file(&path_source, NULL, "(([A-Z]|[a-
z]|(_))|[ ])+\\.orbyte"); //Custom regex

        TextField* path_input = new TextField({ ( - SCREEN_WIDTH / 2), (-
SCREEN_HEIGHT / 2), 0}, path_to_file, graphyte, "solar_system.orbyte");
                    path_source = "solar_system.orbyte"; //default
                    graphyte.text_fields.push_back(path_input);
                    path_gui.Add_Inline_Element(path_input);

                    /*
                            INSPECTOR PARAMETERS GUI INITIALIZATION
                    */

                    //Create Buttons
                    FunctionButton Pause([this]() { this-
>toggle_pause(); }, { (SCREEN_WIDTH / 2) - 25, (SCREEN_HEIGHT / 2) - 25, 0 },
 { 25, 25, 0 }, graphyte, "icons/stop.png");
                    graphyte.function_buttons.push_back(&Pause);

                    FunctionButton Add([this]() { this-
>add_orbit_body(); }, { (SCREEN_WIDTH / 2) - 25, (SCREEN_HEIGHT / 2) - 60, 0
 }, { 25, 25, 0 }, graphyte, "icons/add.png");
                    graphyte.function_buttons.push_back(&Add);

                    FunctionButton Save([this]() { this-
>save(); }, { (SCREEN_WIDTH / 2) - 25, (SCREEN_HEIGHT / 2) - 95, 0 }, { 25, 2
5, 0 }, graphyte, "icons/save.png");
                    graphyte.function_buttons.push_back(&Save);

                    FunctionButton Open([this]() { this-
>open(); }, { (SCREEN_WIDTH / 2) - 25, (SCREEN_HEIGHT / 2) - 130, 0 }, { 25,
25, 0 }, graphyte, "icons/open.png");
                    graphyte.function_buttons.push_back(&Open);

                    //Mainloop time
                    while (!quit)
```

```cpp
                    {
                            //render sun
                            Sun.Draw(graphyte, gCamera);


        clean_orbit_queue(); // Check if any orbits in the vector are schedul
ed for deletion.

                            for (Body* b : orbiting_bodies)
                            {
                                    b-
>Update_Body(deltaTime, time_scale, &orbiting_bodies); // Update body

                                    if (b->snap_camera)
                                    {
                                            vector3 cam_pos = b-
>Get_Position();

        gCamera.position.x = cam_pos.x;

        gCamera.position.y = cam_pos.y;
                                    }
                                    b-
>Draw(graphyte, gCamera); // Draw the body
                            }


        double debug_no_pixels = graphyte.Get_Number_Of_Points();

                            graphyte.draw(); //Draw everything!


                            int current_mouse_x = 0;
                            int current_mouse_y = 0;
                            //Handle input events
                            while (SDL_PollEvent(&sdl_event) != 0)
                            {

        switch (sdl_event.type) //Switch on type of input
                                    {
                                    default:
                                            break;

                                    case SDL_QUIT: //Exit
                                            quit = true;
                                            break;

                                    case SDL_TEXTINPUT: //Typing

        printf("User is typing: %s\n", sdl_event.text.text);

        if (graphyte.active_text_field != NULL)
                                            {

        graphyte.active_text_field->Add_Character(sdl_event.text.text);
                                            }
                                            break;

                                    case SDL_KEYDOWN: //Keypress
```

```cpp
switch (sdl_event.key.keysym.sym)
                                            {
                                            case SDLK_BACKSPACE:

printf("User Pressed the Backspace\n");

if (graphyte.active_text_field != NULL)
                                                {

graphyte.active_text_field->Backspace();
                                                }
                                                break;

                                            case SDLK_RETURN:

commit_to_text_field();
                                                    break;

                                            case SDLK_UP:
                                                //Rotate Up

gCamera.RotateCamera({ 0.01, 0, 0 });
                                                    break;

                                            case SDLK_DOWN:
                                                //Rotate Down

gCamera.RotateCamera({ -0.01, 0, 0 });
                                                    break;

                                            case SDLK_LEFT:
                                                //Rotate Left

printf("\n\nPressed The Left Arrow Key\n");

gCamera.RotateCamera({ 0, -0.01, 0 });
                                                    break;

                                            case SDLK_RIGHT:
                                                //Rotate Right

printf("\n\nPressed The Right Arrow Key\n");

gCamera.RotateCamera({ 0, 0.01, 0 });
                                                    break;
                                            }
                                            break;


case SDL_MOUSEBUTTONDOWN: //Mouseclick

if (sdl_event.button.button == SDL_BUTTON_LEFT) //Left click
                                            {
                                                    int mX = 0;
                                                    int mY = 0;

SDL_GetMouseState(&mX, &mY);
```

```cpp
        click(mX - SCREEN_WIDTH / 2, -
mY + SCREEN_HEIGHT / 2, true); //Adjust for screen dimensions
                                                }

        else if (sdl_event.button.button == SDL_BUTTON_RIGHT) //Right click
                                                {
                                                        int mX = 0;
                                                        int mY = 0;

        SDL_GetMouseState(&mX, &mY);

        click(mX - SCREEN_WIDTH / 2, -mY + SCREEN_HEIGHT / 2, false);
                                                }
                                                break;

                                        case SDL_MOUSEWHEEL:

        if (sdl_event.wheel.y > 0) //Scroll up
                                                {
                                                        //Zoom in

        gCamera.position.z *= 1.4;

        std::cout << "Moved Camera to new pos: " << gCamera.position.z << "\n
";
                                                }

        if (sdl_event.wheel.y < 0) //Scroll down
                                                {
                                                        //zoom out

        gCamera.position.z /= 1.4;

        std::cout << "Moved Camera to new pos: " << gCamera.position.z << "\n
";
                                                }
                                                break;
                                        }
                                }

                                //DELAY UNTIL END
                                deltaTime = Update_Clock(); // get new delta


        float interval = (float)1000 / MAX_FPS; // Intended interval (capped
FPS)

        if (deltaTime < (Uint32)interval) // If simulation is updating too qu
ickly
                                {

        Uint32 delay = (Uint32)interval - deltaTime;
                                        if (delay > 0)
                                        {

        SDL_Delay(delay); // Delay to pad out frame duration and limit FPS
                                                deltaTime += delay;
                                        }
```

```cpp
                                }

                                /*DEBUG*/
                                text_Vertex_Count_Display-
>Set_Text("Vertex Count: " + std::to_string(debug_no_pixels));


        timeSinceStart += ((double)deltaTime * time_scale);
                                text_time_Display-
>Set_Text("Time: " + std::to_string((timeSinceStart) / (1000 * 60 * 60 * 24))
 + "days");


                                frames_since_debug++;
                                time_since_debug += deltaTime;
                                if (frames_since_debug > 1000)
                                {

        double debug_fps = (frames_since_debug * 1000) / time_since_debug;

        std::cout << "\n" << (frames_since_debug * 1000) / time_since_debug;
                                text_FPS_Display-
>Set_Text("FPS: " + std::to_string(debug_fps));
                                        frames_since_debug = 0;
                                        time_since_debug = 0;
                                }
                        }

                }
                //Empty memory and close SDL
                close();

                return 0;
        }
};

int main(int argc, char* args[])
{
        std::cout << "\n_____\nSTARTING ORBYTE\
n_____\n";
        Simulation Sim;
        Sim.run(argc, args);
        return 0;
}
```

Orbyte_Graphics.h

```cpp
#pragma once
#ifndef ORBYTE_GRAPHICS_H
#define ORBYTE_GRAPHICS_H

#include <SDL.h>
#include <SDL_ttf.h>
#include <SDL_image.h>
#include <vector>
#include <string>
#include <numeric>
#include <iostream>
#include <functional> //Functions as variables and parameters
```

```cpp
#include <regex> //Regular Expressions

/*
        A "Texture" class is a way of encapsulating the rendering of more com
plex graphics. Images, fonts etc. would be loaded to a texture.
        Implementation heavily guided by this resource: https://lazyfoo.net/t
utorials/SDL/ A series of tutorials regarding creating an application
        using SDL.

        Largely used for fonts & icons in this application.
*/
class GTexture
{
private:
        //The actual hardware texture
        SDL_Texture* mTexture;

        //The renderer
        SDL_Renderer* renderer;

        //The font
        TTF_Font* font;

        //Image dimensions
        int mWidth;
        int mHeight;

public:
        //Constructor
        GTexture(SDL_Renderer* _renderer = NULL, TTF_Font* _font = NULL)
        {
                //Initialize
                renderer = _renderer;
                font = _font;

                mTexture = NULL; //SDL Texture
                mWidth = 0; //Dimensions
                mHeight = 0;

        }

        GTexture(const GTexture& source) : GTexture{&*source.renderer, &*sour
ce.font}
        {
                //Debugging
                std::cout << "Copy constructor";
        }

        //Deallocates memory
        ~GTexture()
        {
                //Deallocate
                free();
        }

        //Loads image at specified path
        bool loadFromFile(std::string path)
        {
                //Get rid of preexisting texture
                free();
```

```cpp
                //The final texture
                SDL_Texture* newTexture = NULL;

                //Load image at specified path
                SDL_Surface* loadedSurface = IMG_Load(path.c_str());
                if (loadedSurface == NULL)
                {

        printf("Unable to load image %s! SDL_image Error: %s\n", path.c_str()
, IMG_GetError());
                }
                else
                {
                        //Color key image

        //SDL_SetColorKey(loadedSurface, SDL_TRUE, SDL_MapRGB(loadedSurface-
>format, 0, 0xFF, 0xFF));
                        //Create texture from surface pixels

        newTexture = SDL_CreateTextureFromSurface(renderer, loadedSurface);
                        if (newTexture == NULL)
                        {

        printf("Unable to create texture from %s! SDL Error: %s\n", path.c_st
r(), SDL_GetError());
                        }
                        else
                        {
                                //Get image dimensions
                                mWidth = loadedSurface->w;
                                mHeight = loadedSurface->h;
                        }

                        //Get rid of old loaded surface
                        SDL_FreeSurface(loadedSurface);
                }

                //Return success
                mTexture = newTexture;
                return mTexture != NULL;
        }

        //Creates image from font string
        bool loadFromRenderedText(std::string textureText, SDL_Color textColo
r)
        {
                ////Get rid of preexisting texture
                reset_texture();

                if (textureText == "")
                {
                        textureText = " ";
                }
                //Render text surface

        SDL_Surface* textSurface = TTF_RenderUTF8_Solid_Wrapped(font, texture
Text.c_str(), textColor, 320);
                if (textSurface == NULL)
                {
```

```c
        printf("Unable to render text surface! SDL_ttf Error: %s\n", TTF_GetError());
                if (font == NULL)
                {
                        printf("Font pointer was null\n");
                }
        }
        else
        {
                //Create texture from surface pixels

        mTexture = SDL_CreateTextureFromSurface(renderer, textSurface);
                if (mTexture == NULL)
                {

        printf("Unable to create texture from rendered text! SDL Error: %s\n", SDL_GetError());
                }
                else
                {
                        //Get image dimensions
                        mWidth = textSurface->w;
                        mHeight = textSurface->h;
                }

                //Get rid of old surface
                SDL_FreeSurface(textSurface);
        }
        //Return success
        return mTexture != NULL;
}

void reset_texture()
{
        if (mTexture != NULL)
        {
                SDL_DestroyTexture(mTexture);
                mTexture = NULL;
                mWidth = 0;
                mHeight = 0;
        }
}

//Deallocates texture
void free()
{
        //Free texture if it exists
        if (mTexture != NULL)
        {
                //printf("FREE TEXTURE\n");
                SDL_DestroyTexture(mTexture);
                mTexture = NULL;
                renderer = NULL;
                font = NULL;
                mWidth = 0;
                mHeight = 0;
        }
}
```

```cpp
        //Renders texture at given point
        void render(int x, int y, int override_width = NULL, int override_hei
ght = NULL, SDL_Rect* clip = NULL, double angle = 0.0, SDL_Point* center = NU
LL, SDL_RendererFlip flip = SDL_FLIP_NONE)
        {
                //Set rendering space and render to screen
                SDL_Rect renderQuad = { x, y, mWidth, mHeight };
                if (mTexture != NULL)
                {
                        SDL_Rect dst;
                        dst.x = 0;
                        dst.y = 0;

        //Query the texture to get its width and height to use


        SDL_QueryTexture(mTexture, NULL, NULL, &dst.w, &dst.h);

        //printf("Problems accessing text: %s\n", SDL_GetError());

                        if (override_width != NULL)
                        {
                                renderQuad.w = override_width;

        //std::cout << "Overrided width of an icon: " << override_width;
                        }

                        if (override_height != NULL)
                        {
                                renderQuad.h = override_height;
                        }
                }

        if (SDL_RenderCopy(renderer, mTexture, NULL, &renderQuad) == -1) {

        //mTexture seems to be a problem => https://stackoverflow.com/questio
ns/25738096/c-sdl2-error-when-trying-to-render-sdl-texture-invalid-
texture FIXED

        printf("Problems rendering text: %s\n", SDL_GetError());
                }
        }

        //Accessor Methods for retrieving dimensions
        int getWidth()
        {
                return mWidth;
        }

        int getHeight()
        {
                return mHeight;
        }
};

/*
        Encapsulates all key text functionality, such as creating text, movin
g and setting text.
*/
class Text
```

```cpp
{
protected:
        GTexture texture = NULL;

public:
        int pos_x; //Position along x axis in screenspace
        int pos_y; //Position along y axis in screenspace
        std::string text; //Text displayed
        bool visible = true; //Default to visible

        Text(std::string str, int font_size, std::vector<int> position, SDL_R
enderer& _renderer, TTF_Font& _font, SDL_Color color = { 255, 255, 255 })
                : texture(GTexture(&_renderer, &_font))
        {
                //Constructor for the text class.
                text = str;


        if (!texture.loadFromRenderedText(str, color)) //If creating text is
unsuccessful
                {
                        printf("Failed to render text texture!\n");
                }
                pos_x = position[0];
                pos_y = position[1];
        }

        Text(Text& T) //Copy constructor
        {
                texture = T.texture; //Set the texture
                text = T.text; // Copy over text
                if (!texture.loadFromRenderedText(text, {255, 255, 255}))
                {
                        printf("Failed to render text texture!\n");
                }
                pos_x = T.pos_x;
                pos_y = T.pos_y;

        }

        int Set_Text(std::string str, SDL_Color color = {255,255,255})
        {

        if (str == text) //Prevents unnecessary assignments and reloading tex
ture
                {
                        //std::cout << "Redundant text assignment";
                        return 0;
                }

                if (str == "") //Just a tiny bit of redundancy to be safe.
                {
                        str = " ";
                }

                text = str;


        if (!texture.loadFromRenderedText(str, color)) //If not successful
                {
```

```cpp
                printf("Failed to render text texture!\n");
                return -1;
        }
        return 0;
    }

    virtual void Set_Position(vector3 pos)
    {
        //Sets the position of the text with centering
        pos_x = pos.x - texture.getWidth() / 2;
        pos_y = pos.y + texture.getHeight() / 2;
        visible = pos.z >= 0;
    }

    virtual void Set_Position_TL(vector3 pos)
    {

    //In some cases it is more helpful to set the position of the text wi
th the top left anchor point, what SDL uses as the "origin" for textures.
        pos_x = pos.x;
        pos_y = pos.y;
        visible = pos.z >= 0;
    }

    virtual void Set_Visibility(bool is_visible) //Show or hide the text
    {
        visible = is_visible;
    }

    GTexture& Get_Texture() //Accessor method for texture
    {
        return texture;
    }

    vector3 Get_Position() //Accessor method for position
    {
        return vector3{ (double)pos_x, (double)pos_y, 0 };
    }

    vector3 Get_Dimensions() //Accessor method for dimensions
    {

    return vector3{ (double)texture.getWidth(), (double)texture.getHeight
(), 0 };
    }

    int Render(const vector3 screen_dimensions) //Draw the texture to the
 screen
    {
        int s_x = screen_dimensions.x;
        int s_y = screen_dimensions.y;


    if (pos_x < s_x && pos_y < s_y && visible) //Checking if texture is o
n the screen
        {
            texture.render(pos_x + (s_x) / 2, -
pos_y + (s_y) / 2);
        }
        return 0;
```

```cpp
        }

        void Debug()
        {
                std::cout << text<<" | "<<pos_x << "\n";
        }

        ~Text()
        {
                free(); // Deconstructor
        }

        void free()
        {
                texture.free(); //Free the texture
        }
};

/*
        Icon class. This includes an image that can be used for buttons.
*/
class Icon
{
private:
        GTexture texture = NULL;
public:
        int pos_x; //Position along x axis in screenspace
        int pos_y; //Position along y axis in screenspace
        std::string path_to_image; //Path to icon
        bool visible = true; //Can be seen / should be drawn
        std::vector<int> dimensions; //Size (normally square)

        Icon(std::string path, std::vector<int> position, std::vector<int> _d
imensions, SDL_Renderer& _renderer)
                : texture(GTexture(&_renderer, NULL))
        {
                //Constructor for the text class.
                path_to_image = path;

        //No validation of path necessary as user never provides paths to ima
ges to use as icons


        if (!texture.loadFromFile(path)) //If fails to load the image at give
n path.
                {
                        printf("Failed to render icon texture!\n");
                        free();
                }
                pos_x = position[0];
                pos_y = position[1];
                dimensions = _dimensions;
        }

        int Render(const vector3 screen_dimensions) //Draw the icon
        {
                int s_x = screen_dimensions.x;
                int s_y = screen_dimensions.y;

                if (pos_x < s_x && pos_y < s_y && visible)
```

```cpp
                    {
                            //x + SCREEN_WIDTH / 2, -y + SCREEN_HEIGHT / 2
                            texture.render(pos_x + (s_x) / 2, -
pos_y + (s_y) / 2, dimensions[0], dimensions[1]);

            //std::cout << "\nTrying to draw icon @ " << pos_x << " " << pos_y;
                    }
                    return 0;
            }

            void SetPosition(vector3 new_position)
            {
                    vector3 my_dimensions = GetDimensions();
                    pos_x = new_position.x - (my_dimensions.x / 2);
                    pos_y = new_position.y + (my_dimensions.y / 2);
                    //Set the position by centre
            }

            void SetDimensions(std::vector<int> new_dimensions)
            {
                    dimensions = new_dimensions; //Change size of icon
            }

            vector3 GetDimensions() //Accessor method for size of icon
            {
                    return { (double)dimensions[0], (double)dimensions[1], 0 };
            }

            void free()
            {
                    texture.free();

            }
};

class TextField; //A Forward Declaration so nothing breaks

class FunctionButton; //A Forward Declaration so nothing breaks

/*
        Handles all graphics for the application. This includes all pixel wri
tes to the screen; loading and writing to textures; rendering
*/
class Graphyte
{
private: //Private attributes & Methods
        double SCREEN_WIDTH = 0; //What it says on the tin.
        double SCREEN_HEIGHT = 0; //These dimensions are not const values bec
ause they need to be set in the Init() method.

        SDL_Renderer* Renderer = NULL; //Renderer.
        TTF_Font* Font = NULL; //True Type Font. Needs to be loaded at init.

        std::vector<Text*> texts; //Vector of text elements to be drawn to th
e screen.
        std::vector<Icon*> icons; //Vectorr of icon elements to be drawn to t
he screen.
        std::vector<SDL_Point> points; //Vector of points to be drawn to the
screen. Iterate through & draw each point to screen as a pixel.
```

```cpp
public:  //Public attributes & Methods
        TextField* active_text_field = NULL; //This pointer will be used to edit text fields
        std::vector<TextField*> text_fields; //Public as it is accessed by Body to instantiate GUI, consider using an accessor method.
        std::vector<FunctionButton*> function_buttons; //It is possible to handle the input methods in a tidier way, but alas this is all I have time for.

        bool Init(SDL_Renderer& _renderer, TTF_Font& _font, vector3 _screen_dimensions)
        {
                Renderer = &_renderer; //SDL Renderer
                Font = &_font; //Font to be used for text

                //Screen dimensions
                SCREEN_WIDTH = _screen_dimensions.x;
                SCREEN_HEIGHT = _screen_dimensions.y;


        if (Renderer == NULL || Font == NULL) // Fails to initialize if without Renderer or if without font
                {
                        return false;
                }

                return true;
        }

        //This method instantiates a new Text object and returns it. The new text object will be added to the array of text objects: texts.
        Text* CreateText(std::string str, int font_size, SDL_Color color = {
255, 255, 255 })
        {

        Text* newText = new Text(str, font_size, { 0, 0 }, *Renderer, *Font, color);
                std::cout << "Created new text: " << newText->text << "\n";
                texts.push_back(newText);
                return newText;
        }

        //This method instantiates a new icon object and returns it. The new icon object will be added to the queue of icons.
        Icon* CreateIcon(std::string path, std::vector<int> dimensions)
        {

        Icon* newIcon = new Icon(path, {0, 0}, dimensions, *Renderer);
                std::cout << "\nCreated new Icon: " << path << "\n";
                AddIconToRenderQueue(newIcon);
                return newIcon;
        }

        //Generate parameters for a text field. To be used in a copy constructor.
        Text* GetTextParams(std::string str, int font_size, SDL_Color color =
 { 255, 255, 255 })
        {

        Text* newText = new Text(str, font_size, { 0, 0 }, *Renderer, *Font, color);
```

```cpp
            std::cout << "\nCreated new text: " << newText->text << "\n";
            return newText;
    }

    void AddTextToRenderQueue(Text* newText)
    {
            texts.push_back(newText); //Add text to back of render queue
    }

    void RemoveTextFromRenderQueue(Text* text)
    {
            for (int i = 0; i < texts.size(); i++)
            {
                    if (texts[i] == text)
                    {

    texts.erase(texts.begin() + i); //Erase text from queue
                            return;
                    }
            }
    }

    void AddIconToRenderQueue(Icon* icon)
    {
            icons.push_back(icon); //Add icon to queue
    }

    vector3 Get_Screen_Dimensions()
    {

    return { (float)SCREEN_WIDTH, (float)SCREEN_HEIGHT, 0 }; //Accessor m
ethod for screen dimensions
    }

    double Get_Number_Of_Points()
    {
            return points.size(); //Get number of points to be drawn
    }

    void pixel(int x, int y)
    {
            //Check if on screen

    if (std::abs(x) < (float)(SCREEN_WIDTH / 2) && std::abs(y) < (float)(
SCREEN_HEIGHT / 2)) //abs value for negative values... that took so long to f
ind.
            {
                    SDL_Point _point = { x + SCREEN_WIDTH / 2, -
y + SCREEN_HEIGHT / 2 };

    points.emplace_back(_point); //Add to points to be drawn
            }
    }

    //Create a line between two points p1, p2
    void line(float x1, float y1, float x2, float y2)
    {
            int dx = (x2 - x1);
            int dy = (y2 - y1);
```

```cpp
        int length = std::sqrt(dx * dx + dy * dy); //Thank you pythagoras
            float angle = std::atan2(dy, dx);

            for (int i = 0; i < length; i++)
            {
                    pixel(x1 + std::cos(angle) * i,
                            y1 + std::sin(angle) * i);
            }
        }

        //Draw everything to the screen. Called AFTER all points added to the
 render queue
        void draw()
        {

        SDL_SetRenderDrawColor(Renderer, 0, 0, 0, 255); //Render color set to
 black
            SDL_RenderClear(Renderer); //Clear screen
            int count = 0;

            //Draw every pixel in points
            for (auto& point : points)
            {
                    //Color is determined by position on screen

        SDL_SetRenderDrawColor(Renderer, 255, ((float)point.x / (float)SCREEN
_WIDTH) * 255, ((float)point.y / (float)SCREEN_HEIGHT) * 255, 255);
                    SDL_RenderDrawPoint(Renderer, point.x, point.y);
                    count++;
            }

            //Render GUI!
            for (Text* t : texts)
            {
                    t-
>Render({ SCREEN_WIDTH, SCREEN_HEIGHT, 0 }); //Draw text
            }

            for (Icon* i : icons)
            {
                    i-
>Render({ SCREEN_WIDTH, SCREEN_HEIGHT, 0 }); //Draw icon

            }

            SDL_RenderPresent(Renderer); //Draw to screen
            points.clear(); //Clear all pixels
        }

        void free() //Deconstructor
        {
            for (auto& t : texts)
            {
                    t->free();
            }
            texts.clear();
            points.clear();
            SDL_StopTextInput();
        }
```

```cpp
};

/*
        Used to graphically represent vectors.
*/
class Arrow
{
public:
        void Draw(vector3 position, vector3 direction, double magnitude, int
heads, Graphyte& graphyte)
        {
                //These are all 2D vectors.
                vector3 start = position;
                vector3 end = position + (direction * magnitude);
                vector3 perp_dir = vector3{ direction.y, -direction.x, 0 };
                double arrow_head_size = magnitude / 10;


        //Draw arrow heads. Convention dictates 1 for velocity and 2 for acce
leration
                for (int i = 0; i < heads; i++)
                {
                        vector3 ah1 = end + (direction * arrow_head_size);
                        vector3 ah2 = end + (perp_dir * arrow_head_size);
                        vector3 ah3 = end + (perp_dir * -arrow_head_size);

                        graphyte.line(start.x, start.y, end.x, end.y);
                        graphyte.line(ah1.x, ah1.y, ah2.x, ah2.y);
                        /*graphyte.line(ah2.x, ah2.y, ah3.x, ah3.y);*/
                        graphyte.line(ah3.x, ah3.y, ah1.x, ah1.y);

                        end = ah1;
                }
        }
};

class Button
{
private:
        vector3 position; //position of button on screen
        int width; //dimensions
        int height;
        int left_wall_offset; //How far left wall is from centre
        std::function<void()> function = NULL; //Main function to call when c
licked
        std::function<void()> alt_function = NULL; //Alt function to call whe
n right clicked

protected:
        bool enabled = true;
        void CallFunction()
        {
                if (function != NULL)
                {
                        function(); // Call attached function
                }
        }

        void CallAltFunction()
        {
```

```cpp
                if (alt_function != NULL)
                {
                        alt_function(); // Call attached alternative function
                }
        }

        void AttachFunction(std::function<void()> f)
        {
                function = f; // Set function
        }

        void AttachAltFunction(std::function<void()> f)
        {
                alt_function = f;// Set alt function
        }

public:
        Button(vector3 pos, vector3 dimensions)
        {
                position = pos;
                width = dimensions.x;
                height = dimensions.y;
                left_wall_offset = width / 2;
        }

        void SetDimensions(vector3 dimensions)
        {
                width = dimensions.x;
                height = dimensions.y;
        }

        void SetPosition(vector3 pos)
        {
                position = pos;
        }

        virtual void SetEnabled(bool _enabled)
        {
                /*std::cout << "BUTTON CHANGED: " << _enabled;*/
                enabled = _enabled; //Enable or disable button
        }

        bool Clicked(int x, int y)
        {
                if (!enabled)
                {
                        //Disabled button should not be clicked!
                        return false;
                }


        // (0<AM.AB<AB.AB) ^ (0<AM.AD<AD.AD) Where M is a point we're checking

        // Find each vertex using left_wall_offset, position and height of the button.

        vector3 A = { position.x - left_wall_offset, position.y + height / 2, 0 };
```

```cpp
        vector3 B = { position.x - left_wall_offset + width, position.y + hei
ght / 2, 0 };

        vector3 D = { position.x - left_wall_offset, position.y - height / 2,
 0 };

        vector3 C = { position.x - left_wall_offset + width, position.y - hei
ght / 2, 0 };

                // M is the location of the click
                vector3 M = { x, y, 0 };

                vector3 AM = M - A;
                vector3 AB = B - A;
                vector3 BC = C - B;
                vector3 BM = M - A;


        bool in_area = 0 <= AB*AM && AB*AM <= AB*AB && 0 <= BC*BM && BC*BM <=
 BC*BC;
                // Return if click in button bounds.
                return in_area;
        }
};

class FunctionButton : public Button
{
private:
        Icon* icon = NULL; //Icon to show where the button is
public:
        /// <summary>
        /// Constructor
        /// </summary>
        FunctionButton(std::function<void()> f, vector3 pos, vector3 dimensio
ns, Graphyte& g, std::string path_to_icon, std::function<void()> alt_f = NULL
) : Button(pos, dimensions)
        {

        std::cout << "\n Instantiated a function button. Method present?: " <
< (bool)f << "\n";

        if (path_to_icon != "") //Never accessed by user therefor no need for
 complex regular expressions or validation.
            {

        icon = g.CreateIcon(path_to_icon, {(int)dimensions.x, (int)dimensions
.y, 0});
                    icon->SetPosition(pos);
            }
            AttachFunction(f); //Must have a main function
            if (alt_f)
            {

        AttachAltFunction(alt_f); //Does not have to have an alt function
            }
        }

        ~FunctionButton()
        {
```

```cpp
                        free();
                }

        bool CheckForClick(int x, int y, bool alt = false)
        {
                if (Clicked(x, y))
                {
                        //Button has been clicked =>
                        std::cout << "\nFunction button clicked.\n";
                        if (!alt) //If main click
                        {
                                CallFunction();
                        }
                        else {
                                CallAltFunction();
                        }
                        return true;
                }
                return false;
        }

        void SetEnabled(bool _enabled) override // Set both button and icon t
o be enabled or disabled
        {
                Button::SetEnabled(_enabled);
                if (icon)
                {
                        icon->visible = _enabled;
                }
        }

        void free()
        {
                icon->free();
                AttachFunction(NULL);
        }
};

class FieldValue
{
public:
        virtual void ReadField(std::string content) = 0;
};

class DoubleFieldValue : public FieldValue
{
private:
        double* value = NULL; // This is the pointer to the variable the inpu
t field is associated with. E.g. time step or object mass
        std::function<void()> read_f = NULL; // Function to call if successfu
lly read

        bool ValidateValue(std::string content)
        {
                try
                {
                        std::regex dbl_regex("(-+)?([0-9]+(\\.[0-9]+)?)(E[0-
9]+)?"); // Regex

                        if (std::regex_match(content, dbl_regex))
                        {
```

```cpp
                                        // Success
                                        double test_validity = atof(content.c_str());

            std::cout << content << "=>" << test_validity;
                                        return true;
                        }
                        else {
                                throw(content); // Issue
                        }
                }
                catch (std::string bad)
                {

            std::cout << "\n Bad input recieved: " << bad; // Its bad
                        return false; // Failure
                }
        }
public:
        DoubleFieldValue(double* write_to, std::function<void()> f = NULL)
        {
                value = write_to;
                if (f)
                {
                        read_f = f;
                }
        }

        void ReadField(std::string content) override
        {
                if (ValidateValue(content)) // Check if valid
                {

            std::cout << "\n Valid field content"; // Passed validation

            if (value != NULL) // If pointer to value to write to is not null
                        {

            double new_value = atof(content.c_str()); // Conversion

            if (new_value != *value) // Check for redundant set
                                {
                                        *value = new_value;

        std::cout << "\n Successfully wrote to double value from input field!
  \n" << new_value; // Debug

                                        if (read_f != NULL)
                                        {

            read_f(); // Call attached method if it exists.
                                        }
                                }
                        }
                }
        }
};

class StringFieldValue : public FieldValue
{
private:
```

```cpp
        std::string* value = NULL; // This is the pointer to the variable the
 input field is associated with. E.g. object name
        std::function<void()> read_f = NULL;
        std::string regex;
        bool ValidateValue(std::string content)
        {
                try
                {

        std::regex dbl_regex(regex); // Create regex object from string defin
ed in constructor.
                        if (std::regex_match(content, dbl_regex))
                        {
                                return true; // Success
                        }
                        else {

        throw(content); // Content has failed the validation => Catch
                        }
                }
                catch (std::string bad)
                {

        std::cout << "\n Bad input recieved: " << bad; // Its bad
                        return false; // Failure
                }
        }
public:
        StringFieldValue(std::string* write_to, std::function<void()> f = NUL
L, std::string _regex = "(([A-Z]|[a-z])|[0-9])+")
        {
                value = write_to;
                regex = _regex;
                if (f)
                {
                        read_f = f;
                }
        }

        void ReadField(std::string content) override
        {
                if (ValidateValue(content))
                {
                        std::cout << "\n Valid field content";
                        if (value != NULL)
                        {
                                *value = content;

        std::cout << "\n Successfully wrote to value from input field!  \n" <
< content;

                                if (read_f != NULL)
                                {
                                        read_f();
                                }
                        }
                }
        }
};
```

```cpp
class TextField : public Text
{
private:
        SDL_Color text_color = { 255, 255, 255, 0xFF };
        std::string input_text = " ";
        bool enabled = false;
        Button* button = NULL;
        FieldValue &fvalue;

        void Update_Text()
        {
                if (input_text != "")
                {
                        Set_Text(input_text, text_color);
                }
                else {
                        Set_Text(input_text, text_color);
                }
        }

        void update_button_dimensions()
        {

        vector3 dimensions = { Get_Texture().getWidth(), Get_Texture().getHei
ght(), 0 };
                button->SetDimensions(dimensions);
        }

        void write_value()
        {
                fvalue.ReadField(text); //ACCESS VIOLATION!
        }
public:
        TextField(vector3 position, FieldValue& writeto, Graphyte& g, std::st
ring default_text = "This Is An Input Field") :

         Text(*g.GetTextParams(default_text, 16, text_color)), fvalue(writeto
)
        {

        vector3 dimensions = { Get_Texture().getWidth(), Get_Texture().getHei
ght(), 0 };
                input_text = default_text;
                button = new Button(position, dimensions);
                Set_Position({ position.x, position.y, 10 });
                g.AddTextToRenderQueue(this); //Beautiful
        }

        void Set_Position(vector3 position) override //THIS IS AN OVERRIDE SO
 THAT BUTTON POS CAN BE SET AS WELL
        {
                //Sets the position of the text with centering
                pos_x = position.x - texture.getWidth() / 2;
                pos_y = position.y + texture.getHeight() / 2;
                visible = position.z >= 0;

                button->SetPosition(position);
        }

        void Set_Position_TL(vector3 pos) override
```

```cpp
        {

        //In some cases it is more helpful to set the position of the text wi
th the top left anchor point, what SDL uses as the pivot for textures.
                pos_x = pos.x;
                pos_y = pos.y;
                visible = pos.z >= 0;

                button-
>SetPosition({ pos.x + texture.getWidth() / 2 , pos.y - texture.getHeight() /
 2, 0 });


        std::cout << vector3{ pos.x + texture.getWidth() / 2, pos.y - texture
.getHeight() / 2, 0 }.Debug();
        }

    void Set_Visibility(bool is_visible) override
    {
            Text::Set_Visibility(is_visible);
            button->SetEnabled(is_visible);
    }

    void Backspace()
    {
            if (input_text.length() > 0 && enabled)
            {
                    input_text.pop_back();
                    Update_Text();
            }
    }

    void Add_Character(char* chr)
    {
            if (enabled)
            {
                    input_text += chr;
                    Update_Text();
            }
    }

    bool CheckForClick(int x, int y)
    {
            if (button->Clicked(x, y))
            {
                    //Button has been clicked =>
                    Enable();
                    return true;
            }
            return false;
    }

    void Commit()
    {
            std::cout << "\n Committing to text field value! \n";
            write_value();
            update_button_dimensions();

    // We do this so that the button resizes after this new text commit.
    }
```

```cpp
        void Enable()
        {
                SDL_StartTextInput();
                enabled = true;
        }

        void Disable()
        {
                SDL_StopTextInput();
                enabled = false;
                Commit();
                std::cout << "Disabled text";
        }

        ~TextField()
        {
                free();
                Disable();
                //Should be safely destroyed now.
        }
};

//GUI Helpers
struct GUI_Block //"Blocks" are collections of text elements to help with pos
itioning them on screen. It is objectively awesome that its possible for me t
o do this off the framework I've created.
{
        //This struct is going to facilitate pretty much all application GUI.
 Lol.
        std::vector<Text*> elements;
        vector3 position;
        bool is_visible = true;

        GUI_Block(vector3 _position = {0, 0, 0})
        {
                position = _position;
        }

        ~GUI_Block() //Deconstructor... I love c++
        {
                elements.clear();
        }

        void Add_Floating_Element(Text* text, vector3 relative_position)
        {
                elements.push_back(text);
                text->Set_Position_TL(position + relative_position);
        }

        void Hide()
        {
                for (Text* t : elements)
                {
                        t->Set_Visibility(false);
                }
                is_visible = false;
        }

        void Show()
```

```cpp
        {
                for (Text* t : elements)
                {
                        t->Set_Visibility(true);
                }
                is_visible = true;
        }

        void Add_Stacked_Element(Text* text) //This method adds the text to t
he bottom of the block.
        {
                vector3 new_pos = position;

                if (elements.size() > 0)
                {
                        Text* above_this = elements.back();
                        new_pos.y  = above_this-
>Get_Position().y - (above_this->Get_Texture().getHeight());
                }


        //std::cout<< position.Debug() << "=>" << new_pos.Debug() << " WIDTH
IS: " << text->Get_Texture().getWidth() / 2 << "\n";

                text->Set_Position_TL(new_pos);
                elements.push_back(text);
        }

        void Add_Inline_Element(Text* text)
        {
                vector3 new_pos = position;
                if (elements.size() > 0)
                {
                        Text* left = elements.back();
                        new_pos.y = left->Get_Position().y;
                        new_pos.x = left->Get_Position().x + left-
>Get_Texture().getWidth();
                }
                text->Set_Position_TL(new_pos);
                elements.push_back(text);
        }

        void clear()
        {
                elements.clear();
        }

};

//Geometry
struct edge
{
        int a, b;
};

struct Mesh
{
        std::vector<vector3> vertices;
        std::vector<edge> edges;
};
```

```
#endif /*ORBYTE_GRAPHICS_H*/
```

Camera.h

```cpp
#pragma once
#ifndef CAMERA_H
#define CAMERA_H
#include "vec3.h"
#include <iostream>
#include <string>
#include <SDL.h>
#include <stdio.h> //This library makes debugging nicer, but shouldn't really
 be involved in user usage.
#include <vector>
#include <numeric>
#include <sstream>

class Camera
{
private:
        vector3 camera_rotation;

public:
        vector3 position = {0, 0, 0};
        float clipping_z = 1;

        Camera(float _near_clipping_plane = 1, vector3 _position = { 0, 0, -
1.5E9 })
        {
                position = _position;

        std::cout << "Instantiated Camera With Position: " << position.Debug(
) << "\n";
        }

        void RotateCamera(vector3 add_rotation)
        {
                camera_rotation = camera_rotation + add_rotation;
        }

        vector3 rotate(vector3 rot, vector3 point, vector3 c)
        {

                //centroid adjustments
                point.x -= c.x;
                point.y -= c.y;
                point.z -= c.z;

                //float start_magnitude = Magnitude(point);

                //Rotate point
                float rad = 0;
                double x, y, z;
                rad = rot.x;

                x = point.x;
                y = point.y;
```

```cpp
                z = point.z;

                point.y = (std::cos(rad) * y) - (std::sin(rad) * z);
                point.z = (std::sin(rad) * y) + (std::cos(rad) * z);

                x = point.x;
                y = point.y;
                z = point.z;

                rad = rot.y;
                point.x = (std::cos(rad) * x) + (std::sin(rad) * z);
                point.z = (-std::sin(rad) * x) + (std::cos(rad) * z);

                x = point.x;
                y = point.y;
                z = point.z;

                rad = rot.z;
                point.x = (std::cos(rad) * x) - (std::sin(rad) * y);
                point.y = (std::sin(rad) * x) + (std::cos(rad) * y);

                //centroid adjustments
                point.x += c.x;
                point.y += c.y;
                point.z += c.z;

                return point;

        }

        vector3 WorldSpaceToScreenSpace(vector3 world_pos, float screen_heigh
t, float screen_width)
        {

        //manipulate world_pos here such that it is rotated around centre of
universe

        vector3 rotated_world_pos = rotate(camera_rotation, world_pos, { 0, 0
, 0 });

                vector3 pos = rotated_world_pos - position;

        //std::cout << "WORLD POS: " << world_pos.x << " | CAMERA POS: " << p
osition.x << " | => " << pos.x;
                if (pos.z < clipping_z)
                {
                        //DONT DRAW IT
                        //printf("Culled a vertex hopefully \n");
                        return { 0, 0, -1 };
                }
                else {

        //Why ignore the screen_width? Because the screen is wide and we don'
t want to stretch the projection
                        vector3 Screen_Space_Pos = {
                        (pos.x / pos.z) * screen_height,
                        (pos.y / pos.z) * screen_height,
                        pos.z
                        };
                        return Screen_Space_Pos;
```

```
                }
        }
};

#endif /*CAMERA_H*/
```

OrbitBody.h

```cpp
#pragma once

#ifndef ORBITBODY_H
#define ORBITBODY_H

#include <iostream>
#include <string>
#include <SDL.h>
#include <stdio.h>
#include <vector>
#include <numeric>
#include <sstream>
#include "vec3.h"
#include "Orbyte_Data.h"
#include "Orbyte_Graphics.h"
#include "Camera.h"

class CentralBody
{
        /*

        Why does this class not inherit from Body? Because this is designed t
o be static. Any similarity between class attributes and methods

        is due to how graphics have been implemented and consistency with nam
ing conventions, not a design oversight.
        */
private:
        Mesh mesh;

        void Generate_Vertices(double scale)
        {
                std::vector<vector3> _vertices{
                        {1, 0, 0},
                        {-1, 0, 0},

                        {0, 1, 0}, //2
                        {0, -1, 0},

                        {0, 0, 1}, //4
                        {0, 0, -1}
                };

                for (auto& v : _vertices)
                {
                        v.x *= scale;
                        v.x += position.x;
                        v.y *= scale;
                        v.y += position.y;
                        v.z *= scale;
```

```cpp
                v.z += position.z;
            }

            mesh.vertices = _vertices;

            //Edges Now
            std::vector<edge> _edges{
                {0, 3},
                {0, 2},
                {0, 4},
                {0,5},

                {1, 2},
                {1,3},
                {1,4},
                {1,5},

                {2,4},
                {2,5},
                {3,4},
                {3,5}
            };

        mesh.edges = _edges; //I want to put this in its own method, however
that's unnecessary as this is static soooo...

        }

public:
        double mass = 1.989E30;
        double mu = 0;
        double scale;
        const double Gravitational_Constant = 6.6743E-11;
        vector3 position;

        CentralBody(double _mass = 1.989E30, double _scale=6.96E8)
        {
            mu = Gravitational_Constant * mass;
            position = { 0, 0, 0 };
            scale = _scale;
            Generate_Vertices(_scale);
        }

        int Draw(Graphyte& g, Camera& c)
        {

        vector3 screen_dimensions = g.Get_Screen_Dimensions(); //Vector3 cont
aining Screen Dimensions, we ignore z
            std::vector<vector3> verts = mesh.vertices;

            for (vector3& p : verts)
            {

        p = c.WorldSpaceToScreenSpace(p, screen_dimensions.x, screen_dimensio
ns.y);

                if (p.z > 0)
                {
                    g.pixel(p.x, p.y);
                }
```

```cpp
				}

				for (edge edg : mesh.edges)
				{
					if (verts[edg.a].z > 0 && verts[edg.b].z > 0)
					{

		//std::cout << "Debugging C_Body rendering: " << verts[edg.a].Debug()
 << "\n";
						g.line(verts[edg.a].x,
							verts[edg.a].y,
							verts[edg.b].x,
							verts[edg.b].y
						);
					}
				}

				verts.clear();

				return 0;
		}

		Mesh Get_Mesh()
		{
				return mesh;
		}

		void RegenerateVertices()
		{

		std::cout << "\n Recalculating vertex positions w/ scale: " << scale;
				std::cout << "\n Me: (regen) " << this;
				this->mesh.vertices.clear();
				this->Generate_Vertices(scale);
		}

		void RecalculateMu()
		{
				mu = Gravitational_Constant * mass;
		}
};

class Satellite; //A Forward Declaration so nothing collapses

class Body
{
private:
		std::vector<Satellite*> satellites;
		Graphyte& graphyte;

		int Add_Satellite(Satellite* sat);

		void Delete_Satellites();

		void Create_Satellite();

		int Update_Satellites(float delta, float time_scale, std::vector<Body
*>* bodies_in_system);

		int Draw_Satellites(Graphyte& g, Camera& c);
```

```cpp
        void Close_Satellite_Inspectors();

        int Clean_Up_Satellites();

protected:
        Mesh mesh;
        vector3 last_trail_point;
        std::vector<vector3> trail_points;

        vector3 start_pos;
        vector3 start_vel;
        double time_since_start = 0;

        //Orbit information
        vector3 position{ 0, 0, 0 };
        double radius;
        vector3 velocity{ 0,0,0 };
        double angular_velocity = 0;
        vector3 acceleration{ 0, 0, 0 };
        double mu = 0;
        double mass = 0;
        double scale;
        const double Gravitational_Constant = 6.6743E-11;

        //Labels
        Text* name_label = NULL;
        Text* inspector_name = NULL;
        Text* inspector_mass = NULL;
        Text* inspector_radius = NULL;
        Text* inspector_velocity = NULL;
        Text* inspector_angular_velocity = NULL;
        Text* inspector_acceleration = NULL;
        Text* inspector_period = NULL;

        //Inspector Function Buttons
        FunctionButton* inspector_reset = NULL;
        FunctionButton* inspector_delete = NULL;
        FunctionButton* inspector_satellite = NULL;

        //Field Values
        DoubleFieldValue ScaleFV;
        DoubleFieldValue MassFV;
        DoubleFieldValue PosXFV, PosYFV, PosZFV;
        DoubleFieldValue VelXFV, VelYFV, VelZFV;

        StringFieldValue NameFV;

        //GUI
        GUI_Block* gui = NULL;

        //BUTTON
        FunctionButton* f_button = NULL;

        virtual void update_inspector()
        {
                if (gui->is_visible)
                {
```

```cpp
                    if (inspector_name != NULL){ inspector_name-
>Set_Text(name);}//The set text method checks if we are making a redundant se
t => more performant
                    if (inspector_mass != NULL) { inspector_mass-
>Set_Text("| Mass: " + std::to_string(mass) + "kg"); }
                    if (inspector_radius != NULL) { inspector_radius-
>Set_Text("| Radius: " + std::to_string(Magnitude(position) / 1000) + "km");
}
                    if (inspector_velocity != NULL) { inspector_velocity-
>Set_Text("| Velocity: " + velocity.Debug()); }

        if (inspector_angular_velocity != NULL) { inspector_angular_velocity-
>Set_Text("| Angular Velocity: " + std::to_string(angular_velocity * 60 * 60
* 24) + "rad/day"); }

        if (inspector_acceleration != NULL) { inspector_acceleration-
>Set_Text("| Acceleration: " + acceleration.Debug()); }
                    if (inspector_period != NULL) { inspector_period-
>Set_Text("| Orbit Period: " + std::to_string(Calculate_Period() / (60 * 60 *
 24)) + " days"); }
                }
        }

        std::vector<vector3> two_body_ode(float t, vector3 _r, vector3 _v, st
d::vector<Body*>* masses)
        {
                vector3 a;
                vector3 pos = _r; //displacement
                vector3 v = _v; //velocity


                //SUN
                vector3 r = pos; //displacement

                vector3 nr = Normalize(r);
                //std::cout << "NR.z" << nr.z << "\n";
                if (nr.z == 0) { nr.z = 1; r.z = 0; }
                if (nr.y == 0) { nr.y = 1; r.y = 0; }
                if (nr.x == 0) { nr.x = 1; r.x = 0; }
                double mag = Magnitude(r);
                a = vector3({
                        (-mu * r.x) / (pow(mag, 3)),
                        (-mu * r.y) / (pow(mag, 3)),
                        (-mu * r.z) / (pow(mag, 3))
                        });

                //Others

                for (Body* b : *masses)
                {
                        if (b != this)
                        {
                                vector3 r = pos - b-
>Get_Position(); //displacement

                                double _mu = Gravitational_Constant * b-
>Get_Mass();

                                //std::cout << "R.z" << r.z << "\n";
                                vector3 nr = Normalize(r);
```

```cpp
                                    //std::cout << "NR.z" << nr.z << "\n";
                                    if (nr.z == 0) { nr.z = 1; r.z = 0; }
                                    if (nr.y == 0) { nr.y = 1; r.y = 0; }
                                    if (nr.x == 0) { nr.x = 1; r.x = 0; }
                                    double mag = Magnitude(r);
                                    a = a + vector3({
                                            (-_mu * r.x) / (pow(mag, 3)),
                                            (-_mu * r.y) / (pow(mag, 3)),
                                            (-_mu * r.z) / (pow(mag, 3))
                                    });
                            }
                    }
                    //std::cout << "rk_result: " << (pow(nr.z, 3)) << "\n";
                    return { v, a };
            }

            std::vector<vector3> rk4_step(float _time, vector3 _position, vector3
     _velocity, std::vector<Body*>* masses, float _dt = 1)
            {

            //std::cout << "\n DEBUGGING RK4 STEP FOR: " + name + "\n" + "positio
    n: " + _position.Debug() + "\nvelocity: " + _velocity.Debug();
                    //structure of the vectors: [pos, velocity]

            std::vector<vector3> rk1 = two_body_ode(_time, _position, _velocity,
    masses);

            std::vector<vector3> rk2 = two_body_ode(_time + (0.5 * _dt), _positio
    n + (rk1[0] * 0.5f * _dt), _velocity + (rk1[1] * 0.5f * _dt), masses);

            std::vector<vector3> rk3 = two_body_ode(_time + (0.5 * _dt), _positio
    n + (rk2[0] * 0.5f * _dt), _velocity + (rk2[1] * 0.5f * _dt), masses);

            std::vector<vector3> rk4 = two_body_ode(_time + _dt, _position + (rk3
    [0] * _dt), _velocity + (rk3[1] * _dt), masses);


            vector3 result_pos = _position + (rk1[0] + (rk2[0] * 2.0f) + (rk3[0]
    * 2.0f) + rk4[0]) * (_dt / 6.0f);

            vector3 result_vel = _velocity + (rk1[1] + rk2[1] * 2 + rk3[1] * 2 +
    rk4[1]) * (_dt / 6);

            //std::cout << "\n Result FOR: " + name + "\n" + "position: " + resul
    t_pos.Debug() + "\nvelocity: " + result_vel.Debug();
                    return { result_pos, result_vel, rk1[1]};
            }

            //We need to override initial velocities in case user wants a perfect
    ly circular orbit.
            virtual void Project_Circular_Orbit(vector3& _velocity)
            {

            //We manipulate the velocity so that a perfectly circular orbit is ac
    hieved
                    if (position.x != 0)
                    {
                            _velocity.y = sqrt(mu / position.x);
                    }
                    if (position.y != 0)
```

```cpp
                {
                        _velocity.x = sqrt(mu / position.y);
                }
                if (position.z != 0)
                {
                        _velocity.x = sqrt(mu / position.z);
                }


        //Don't have to return a value because parameter is passed by referen
ce.
        }

        virtual std::vector<vector3> Generate_Vertices(double scale)
        {
                std::vector<vector3> _vertices{
                        {1, 0, 0},
                        {-1, 0, 0},

                        {0, 1, 0}, //2
                        {0, -1, 0},

                        {0, 0, 1}, //4
                        {0, 0, -1}
                };

                for (auto& v : _vertices)
                {
                        v.x *= scale;
                        v.x += position.x;
                        v.y *= scale;
                        v.y += position.y;
                        v.z *= scale;
                        v.z += position.z;
                }

                //Edges Now
                std::vector<edge> _edges{
                        {0, 3},
                        {0, 2},
                        {0, 4},
                        {0,5},

                        {1, 2},
                        {1,3},
                        {1,4},
                        {1,5},

                        {2,4},
                        {2,5},
                        {3,4},
                        {3,5}
                };
                mesh.edges = _edges;

                printf("\n Generated vertices");

                return _vertices;
        }
```

```cpp
void MoveToPos(vector3 new_pos)
{
        vector3 old_pos = position;
        position = new_pos;
        radius = Magnitude(position);

        vector3 delta = position - old_pos;

        for (auto& p : mesh.vertices)
        {
                p = p + delta;
        }

        return;
}

vector3 rotate(vector3 rot, vector3 point, vector3 c)
{

        //centroid adjustments
        point.x -= c.x;
        point.y -= c.y;
        point.z -= c.z;

        //float start_magnitude = Magnitude(point);

        //Rotate point
        float rad = 0;
        float x, y, z;
        rad = rot.x;

        x = point.x;
        y = point.y;
        z = point.z;

        point.y = (std::cos(rad) * y) - (std::sin(rad) * z);
        point.z = (std::sin(rad) * y) + (std::cos(rad) * z);

        x = point.x;
        y = point.y;
        z = point.z;

        rad = rot.y;
        point.x = (std::cos(rad) * x) + (std::sin(rad) * z);
        point.z = (-std::sin(rad) * x) + (std::cos(rad) * z);

        x = point.x;
        y = point.y;
        z = point.z;

        rad = rot.z;
        point.x = (std::cos(rad) * x) - (std::sin(rad) * y);
        point.y = (std::sin(rad) * x) + (std::cos(rad) * y);

        //centroid adjustments
        point.x += c.x;
        point.y += c.y;
        point.z += c.z;

        return point;
```

```cpp
        }

        void rotate_about_centre(vector3 rot)
        {
                for (auto& p : mesh.vertices)
                {
                        p = rotate(rot, p, position);
                }
        }

        void CreateInspector(Graphyte& g)
        {
                gui = new GUI_Block();
                vector3 screen_dimensions = g.Get_Screen_Dimensions();
                gui->position = { (screen_dimensions.x / 2) - 300, -
(screen_dimensions.y / 2) + 330, 0 };
                inspector_name = g.CreateText(name + ": ", 12);
                gui->Add_Stacked_Element(inspector_name);
                std::cout << inspector_name;
                inspector_mass = g.CreateText(std::to_string(mass), 12);
                gui->Add_Stacked_Element(inspector_mass);

        inspector_radius = g.CreateText(std::to_string(Magnitude(position)),
12);
                gui->Add_Stacked_Element(inspector_radius);

        inspector_velocity = g.CreateText("velocity should be here", 12);
                gui->Add_Stacked_Element(inspector_velocity);

        inspector_angular_velocity = g.CreateText("angular velocity should be
here", 12);
                gui->Add_Stacked_Element(inspector_angular_velocity);

        inspector_acceleration = g.CreateText("acceleration should be here",
12);
                gui->Add_Stacked_Element(inspector_acceleration);
                inspector_period = g.CreateText("period should be here", 12);
                gui->Add_Stacked_Element(inspector_period);

                //Input fields
                gui-
>Add_Stacked_Element(g.CreateText("EDIT PARAMETERS_____", 12));

                gui->Add_Stacked_Element(g.CreateText("| Name: ", 10));
                TextField* tf = new TextField({ 10,10,0 }, NameFV, g, name);
                g.text_fields.push_back(tf);
                gui->Add_Inline_Element(tf);

                gui->Add_Stacked_Element(g.CreateText("| Scale: ", 10));

        tf = new TextField({ 10,10,0 }, ScaleFV, g, std::to_string(scale));
                g.text_fields.push_back(tf);
                gui->Add_Inline_Element(tf);

                gui->Add_Stacked_Element(g.CreateText("| Mass: ", 10));

        tf = new TextField({ 10,10,0 }, MassFV, g, std::to_string(mass));
                g.text_fields.push_back(tf);
                gui->Add_Inline_Element(tf);
```

```cpp
            //POSITION:

            gui->Add_Stacked_Element(g.CreateText("| Position x: ", 10));

      tf = new TextField({ 10,10,0 }, PosXFV, g, std::to_string(position.x)
);
            g.text_fields.push_back(tf);
            gui->Add_Inline_Element(tf);

            gui->Add_Stacked_Element(g.CreateText("| Position y: ", 10));

      tf = new TextField({ 10,10,0 }, PosYFV, g, std::to_string(position.y)
);
            g.text_fields.push_back(tf);
            gui->Add_Inline_Element(tf);

            gui->Add_Stacked_Element(g.CreateText("| Position z: ", 10));

      tf = new TextField({ 10,10,0 }, PosZFV, g, std::to_string(position.z)
);
            g.text_fields.push_back(tf);
            gui->Add_Inline_Element(tf);

            //VELOCITY:

            gui->Add_Stacked_Element(g.CreateText("| Velocity x: ", 10));

      tf = new TextField({ 10,10,0 }, VelXFV, g, std::to_string(velocity.x)
);
            g.text_fields.push_back(tf);
            gui->Add_Inline_Element(tf);

            gui->Add_Stacked_Element(g.CreateText("| Velocity y: ", 10));

      tf = new TextField({ 10,10,0 }, VelYFV, g, std::to_string(velocity.y)
);
            g.text_fields.push_back(tf);
            gui->Add_Inline_Element(tf);

            gui->Add_Stacked_Element(g.CreateText("| Velocity z: ", 10));

      tf = new TextField({ 10,10,0 }, VelZFV, g, std::to_string(velocity.z)
);
            g.text_fields.push_back(tf);
            gui->Add_Inline_Element(tf);

            //FUNCTION BUTTONS:

            //Create the button
            f_button = new FunctionButton([this]() { this-
>ShowBodyInspector(); }, name_label->Get_Position(), name_label-
>Get_Dimensions(), g, "", [this]() { this-
>snap_camera_to_body(); }); //TODO: Fix this please

      g.function_buttons.push_back(f_button); //No idea how this has access
 to function_buttons but so it does...
            std::cout << "\n\n\n\nFUNCTION BUTTON " << f_button;
```

```cpp
                inspector_delete = new FunctionButton([this]() { this-
>Delete(); }, { (screen_dimensions.x / 2) - 275, -
(screen_dimensions.y / 2) + 30, 0 }, {25, 25, 0}, g, "icons/delete.png");
                g.function_buttons.emplace_back(inspector_delete);

                inspector_reset = new FunctionButton([this]() { this-
>Reset(); }, { (screen_dimensions.x / 2) - 245, -
(screen_dimensions.y / 2) + 30, 0 }, { 25, 25, 0 }, g, "icons/reset.png");
                g.function_buttons.emplace_back(inspector_reset);

                inspector_satellite = new FunctionButton([this]() { this-
>Create_Satellite(); }, {(screen_dimensions.x / 2) - 215, -
(screen_dimensions.y / 2) + 30, 0}, {25, 25, 0}, g, "icons/add.png");
                g.function_buttons.emplace_back(inspector_satellite);

                HideBodyInspector();
        }

        void snap_camera_to_body()
        {
                snap_camera = true;
        }

public:
        std::string name;
        bool to_delete = false; //Used in mainloop to schedule objects for de
letion next update. => deconstructor (see free())
        bool snap_camera = false;

        Body(std::string _name, vector3 _center, double _mass, double _scale,
 vector3 _velocity, double _mu, Graphyte& g, bool override_velocity = false):
                graphyte(g),
                ScaleFV(&scale, [this]() { this-
>RegenerateVertices(); }), MassFV(&mass), NameFV(&name, [this]() { this-
>Rename(); }), // Scale Dield Value. When written to, recalculate geometry
                PosXFV(&this->position.x, [this]() { this-
>RecenterBody(); }), PosYFV(&this->position.y, [this]() { this-
>RecenterBody(); }), PosZFV(&this->position.z, [this]() { this-
>RecenterBody(); }),
                VelXFV(&this->velocity.x, [this]() { this-
>SetStartVelocity(); }), VelYFV(&this->velocity.y, [this]() { this-
>SetStartVelocity(); }), VelZFV(&this->velocity.z, [this]() { this-
>SetStartVelocity(); })
        {
                position = _center;
                start_pos = position;
                radius = Magnitude(position);


        name_label = g.CreateText(_name, 16); // Create floating text label t
hat follows orbit body
                name_label->pos_x = 100; //Test values (overwritten later)
                name_label->pos_y = 100;

                mu = _mu; //Setting attributes
                name = _name;
                if (override_velocity)
                {
```

```cpp
        Project_Circular_Orbit(_velocity); // Force a circular orbit. Not rec
commended as will override given parameters. [NO LONGER SUPPORTED]
            }
            velocity = _velocity;
            start_vel = velocity;

            scale = _scale;
            mass = _mass;


    std::cout << "Instantiated Orbiting Body with initial position: " <<
start_pos.Debug() << " and velocity: " << velocity.Debug() << "\n";


    mesh.vertices = Generate_Vertices(scale); // Generate body geometry

            CreateInspector(g); // Create GUI for body
    }

    ~Body()
    {
            free();
    }

    void free() //I *WANT* to improve this, but the NEA deadline is loomi
ng so this is how it stays for now.
    {

            satellites.clear();
            mesh.vertices.clear();
            trail_points.clear();
            mesh.edges.clear();

            gui = nullptr;
            f_button->SetEnabled(false);
            f_button = nullptr;
    }

    void RegenerateVertices()
    {

    std::cout << "\n Recalculating vertex positions w/ scale: " << scale;
            std::cout << "\n Me: (regen) " <<this;
            this->mesh.vertices.clear();
            this->mesh.vertices = this->Generate_Vertices(scale);
            std::cout << "\n" << Magnitude(mesh.vertices[0] - position);
    }

    void RecenterBody()
    {
            MoveToPos(position);
            //Project_Circular_Orbit(velocity);
            start_pos = position;
            time_since_start = 0;
    }

    void SetStartVelocity()
    {
            start_vel = velocity;
```

```cpp
            time_since_start = 0;
        }

        void Rename()
        {
            std::cout << "\nRenamed an orbiting body.";
            name_label->Set_Text(name);
            return;
        }

        void ShowBodyInspector()
        {
            inspector_delete->SetEnabled(true);
            inspector_reset->SetEnabled(true);
            inspector_satellite->SetEnabled(true);
            gui->Show();
        }

        void HideBodyInspector()
        {
            inspector_delete->SetEnabled(false);
            inspector_reset->SetEnabled(false);
            inspector_satellite->SetEnabled(false);
            gui->Hide();
            Close_Satellite_Inspectors();
        }

    OrbitBodyData GetOrbitBodyData() //To be used when saving to a .orbyt
e file
        {
            return OrbitBodyData(name, position, mass, scale, velocity);
        }

        virtual std::string DebugBody() //For debugging purposes...
        {

        std::string text ="\n\n_____\n" +  name + "\n|| Mass: " +
std::to_string(mass) + "\n|| Position: " + position.Debug() + "\n|| Velocity:
 " + velocity.Debug() + "\n|| Period: " + std::to_string(Calculate_Period() /
 (60 * 60 * 24)) + "days" + "\n_____\n\n";
            return text;
        }

        void Reset()
        {
            time_since_start = 0;
            position = start_pos;
            radius = Magnitude(position);
            velocity = start_vel;
            RegenerateVertices();
        }

        void Delete()
        {
            std::cout << "\n|||DELETED ORBIT BODY: " << name << "|||";
            HideBodyInspector();
            name_label->Set_Visibility(false);
            f_button->SetEnabled(false);
            to_delete = true;
            Delete_Satellites();
```

```cpp
        }

        virtual int Update_Body(float delta, float time_scale, std::vector<Body*>* bodies_in_system)
        {
                if (time_scale == 0) // If paused, don't update.
                {
                        return 0;
                }


        Update_Satellites(delta, time_scale, bodies_in_system); // Call Update Method of all child satellites


        rotate_about_centre({0.0001 * time_scale * delta / 1000, 0.0001 * time_scale * delta / 1000, 0.0001 * time_scale * delta / 1000 }); // Gradual rotation about body origin to mimic a planet's rotation about its axis

                vector3 this_pos = position;
                float t = (delta / 1000); //time in seconds

        std::vector<vector3> sim_step = rk4_step(time_since_start, this_pos, velocity, bodies_in_system, t * time_scale); // Get RK4 result into a sim_step buffer.
                this_pos = sim_step[0];

        //if (position.z > 0) { std::cout << position.Debug() << "\n"; std::cout << velocity.Debug() << "\n"; }
                MoveToPos(this_pos); // Shift vertices to new position

        angular_velocity = Magnitude(velocity) / Magnitude(position); // angular velocity = tangential velocity / radius
                time_since_start += t * time_scale;


        // Add a "breadcrumb" or trail point if a certain distance away from last

        if (Magnitude(this_pos - last_trail_point) > (0.5 * radius) / 24)
                {
                        trail_points.emplace_back(this_pos);
                        last_trail_point = this_pos;
                }

                // Remove trail point to only show most recent
                if (trail_points.size() > 24)
                {
                        trail_points.erase(trail_points.begin());
                }

                velocity = sim_step[1]; // Get result from RK4 buffer
                acceleration = sim_step[2];

                update_inspector(); // Update GUI

                return 0; // Successful update.
        }
```

```cpp
        int Draw_Arrows(Graphyte& g, Camera& c, vector3 start, vector3 screen
_dimensions)
        {
                //Draw arrow for velocity
                Arrow arrow_velocity;
                double arrow_modifier = c.position.z < 0 ? c.position.z * -
(1 / 1E6) : c.position.z * (1 / 1E6);

        vector3 arrow_end = c.WorldSpaceToScreenSpace(position + (velocity *
arrow_modifier), screen_dimensions.x, screen_dimensions.y);
                vector3 dir = arrow_end - start;

        arrow_velocity.Draw(start, Normalize(dir), Magnitude(dir), 1, g); //D
raw arrow, with 1 head.

                //Draw arrow for acceleration
                Arrow arrow_acceleration;
                arrow_modifier = c.position.z < 0 ? c.position.z * -
(1/2) : c.position.z * (1/2);

        arrow_end = c.WorldSpaceToScreenSpace(position + (acceleration * arro
w_modifier), screen_dimensions.x, screen_dimensions.y);
                dir = arrow_end - start;

        arrow_acceleration.Draw(start, Normalize(dir), Magnitude(dir), 2, g);
 //Draw arrow, with 2 heads.

                return 0;
        }

        int Draw(Graphyte& g, Camera& c)
        {

        vector3 screen_dimensions = g.Get_Screen_Dimensions(); //Vector3 cont
aining Screen Dimensions, we ignore z
                std::vector<vector3> verts = this->mesh.vertices;
                for (auto& p : verts)
                {

        p = c.WorldSpaceToScreenSpace(p, screen_dimensions.x, screen_dimensio
ns.y);

                        if (p.z > 0)
                        {
                                g.pixel(p.x, p.y);
                        }
                }

                for (vector3 t_p : trail_points)
                {

        t_p = c.WorldSpaceToScreenSpace(t_p, screen_dimensions.x, screen_dime
nsions.y);
                        if (t_p.z > 0)
                        {
                                g.pixel(t_p.x, t_p.y);

        //std::cout << t_p.y << "\n"; //testing a hunch
                        }
                }
```

```cpp
                for (edge edg : mesh.edges)
                {
                        if (verts[edg.a].z > 0 && verts[edg.b].z > 0)
                        {
                                //std::cout << verts[edg.a].Debug() << "\n";
                                g.line(verts[edg.a].x,
                                        verts[edg.a].y,
                                        verts[edg.b].x,
                                        verts[edg.b].y
                                );
                        }
                }

                verts.clear();

                //Make two lines for the orbit body label:
                vector3 start = position;
                vector3 end1 = position + vector3{scale, -scale, 0};

        start = c.WorldSpaceToScreenSpace(start, screen_dimensions.x, screen_
dimensions.y);

        end1 = c.WorldSpaceToScreenSpace(end1, screen_dimensions.x, screen_di
mensions.y);
                vector3 end2 = end1 + vector3{ (double)name_label-
>Get_Texture().getWidth(), 0, 0 };
                vector3 label_pos = end1 + ((end2 - end1) * 0.5);
                label_pos.y += (double)name_label-
>Get_Texture().getHeight() / 2;

        //Move & Draw Label: Done in the draw method because we need access t
o the camera and creating a new method makes no sense.
                g.line(start.x, start.y, end1.x, end1.y);
                g.line(end1.x, end1.y, end2.x, end2.y);
                name_label->Set_Position(label_pos);
                if (f_button != NULL)
                {
                        f_button->SetPosition(label_pos);
                }

                Draw_Arrows(g, c, start, screen_dimensions);

                Draw_Satellites(g, c);

                return 0;
        }

        Mesh Get_Mesh()
        {
                //Return vertices
                return mesh;
        }

        std::vector<vector3> Get_Trail_Points()
        {
                return trail_points;
        }

        vector3 Get_Tangential_Velocity()
```

```cpp
        {
                return velocity;
        }

        vector3 Get_Position()
        {
                return position;
        }

        double Get_Mass()
        {
                return mass;
        }

        vector3 Get_Acceleration()
        {
                return acceleration;
        }

        void Set_Mu(double _mu);

        double Get_Mu()
        {
                return mu;
        }

        /// <summary>
        /// The amount of time in seconds for the body to complete 1 orbit
        /// </summary>
        /// <returns>Time period</returns>
        virtual double Calculate_Period()
        {

        double length_of_orbit = 2 * 3.14159265359 * radius; //// 2 * pi * R
                double t = length_of_orbit / Magnitude(velocity);
                return t;
        }
};

class Satellite : public Body
{
private:
        Body* parentBody; // Pointer to parent, e.g. Moon -> Earth
        //Satellites have different geometry! Cube.
        std::vector<vector3> Generate_Vertices(double scale) override {
                //Polymorphism!
                std::vector<vector3> _vertices{
                        {0, 1, 1},
                        {1, 0, 1},
                        {0, -1, 1},
                        {-1, 0, 1},

                        {0, 1, -1},
                        {1, 0, -1},
                        {0, -1, -1},
                        {-1, 0, -1},
                };

                for (auto& v : _vertices)
                {
```

```cpp
                        v.x *= scale;
                        v.x += position.x;
                        v.y *= scale;
                        v.y += position.y;
                        v.z *= scale;
                        v.z += position.z;
                }

                //Edges Now
                std::vector<edge> _edges{
                        {0,1},
                        {0, 3},
                        {2, 1},
                        {2, 3},

                        {4, 5},
                        {4, 7},
                        {6, 5},
                        {6, 7},

                        {0, 4},
                        {1, 5},
                        {2, 6},
                        {3, 7}
                };
                mesh.edges = _edges;

                return _vertices;
        };
        //Override Circular Orbit Projection [NO LONGER SUPPORTED]
        void Project_Circular_Orbit(vector3& _velocity) override {
                vector3 p_velocity = parentBody->Get_Tangential_Velocity();

        //We manipulate the velocity so that a perfectly circular orbit is ac
hieved
                if (position.x != 0)
                {
                        _velocity.y = sqrt(mu / position.x);
                }
                if (position.y != 0)
                {
                        _velocity.x = sqrt(mu / position.y);
                }
                if (position.z != 0)
                {
                        _velocity.x = sqrt(mu / position.z);
                }
                std::cout << "Someone has projected a circular orbit!";

        _velocity = _velocity + p_velocity; // Adding the parent velocity bec
ause we need this to orbit something moving through space, not orbiting where
 it thought it was.


        //Don't have to return a value because parameter is passed by referen
ce.
        }

protected:
        //Override Inspector Values
```

```cpp
        void update_inspector() override
        {
                if (gui->is_visible)
                {
                        if (inspector_name != NULL) { inspector_name-
>Set_Text(parentBody-
>name + "'s: " + name); }//The set text method checks if we are making a redu
ndant set => more performant
                        if (inspector_mass != NULL) { inspector_mass-
>Set_Text("| Mass: " + std::to_string(mass) + "kg"); }
                        // Override Radius
                        if (inspector_radius != NULL) { inspector_radius-
>Set_Text("| Radius: " + std::to_string(Magnitude(position - parentBody-
>Get_Position()) / 1000) + "km"); }
                        // Relative Velocity
                        if (inspector_velocity != NULL) { inspector_velocity-
>Set_Text("| Velocity: " + (velocity - parentBody-
>Get_Tangential_Velocity()).Debug()); }

        if (inspector_angular_velocity != NULL) { inspector_angular_velocity-
>Set_Text("| Angular Velocity: " + std::to_string(angular_velocity * 60 * 60
* 24) + "rad/day"); }

        if (inspector_acceleration != NULL) { inspector_acceleration-
>Set_Text("| Acceleration: " + (acceleration - parentBody-
>Get_Acceleration()).Debug()); }
                        if (inspector_period != NULL) { inspector_period-
>Set_Text("| Orbit Period: " + std::to_string(Calculate_Period() / (60 * 60 *
 24)) + " days"); }
                }
        }
        //Override Period Calculation
        double Calculate_Period() override
        {
                //double T = 2 * 3.14159265359 * sqrt((pow(radius, 3) / mu));

        double length_of_orbit = 2 * 3.14159265359 * (Magnitude(parentBody-
>Get_Position() - position)); //Relative Position
                double t = length_of_orbit / Magnitude(velocity - parentBody-
>Get_Tangential_Velocity());
                return t;
        }

public:
        //Constructor
        Satellite(std::string _name, Body* _parentBody, vector3 center, doubl
e _mass, double _scale, vector3 _velocity, Graphyte& g, bool override_velocit
y = false):
                Body(_name, center + _parentBody-
>Get_Position(), _mass, _scale, _velocity + _parentBody-
>Get_Tangential_Velocity(), _parentBody-
>Get_Mu(), g, false), parentBody(_parentBody)
                {

        std::cout << "\n_____\nSATELLITE INSTANTIATION\n_____\n
" << "parent body name: " << parentBody-
>name << "\nparent body location: " << parentBody-
>Get_Position().Debug() << "\nmy location: " << Get_Position().Debug() + "\n"
;
```

```cpp
        std::cout << "\nSAT POS (RELATIVE) CONSTRUCTOR:" + (position).Debug()
 + "\n";

        std::cout << "SAT VEL (RELATIVE) CONSTRUCTOR:" + (velocity).Debug() +
 " MEANT TO BE: " + _velocity.Debug() + "\n";
        }
        //Override Update
        int Update_Body(float delta, float time_scale, std::vector<Body*>* bo
dies_in_system) override
        {
                if (time_scale == 0) // If paused, don't update.
                {
                        return 0;
                }

                //rotate(0.0005f, 0.0005f, 0.0005f);
                vector3 this_pos = position;
                float t = (delta / 1000); //time in seconds

        std::vector<vector3> sim_step = rk4_step(time_since_start, this_pos,
velocity, bodies_in_system, t * time_scale);
                this_pos = sim_step[0];
                radius = Magnitude(this_pos - parentBody->Get_Position());

                MoveToPos(this_pos);
                angular_velocity = Magnitude(velocity - parentBody-
>Get_Tangential_Velocity()) / Magnitude(position - parentBody-
>Get_Position());
                time_since_start += t * time_scale;


        if (Magnitude(this_pos - last_trail_point) > (0.5 * radius) / 24)
                {
                        trail_points.emplace_back(this_pos);
                        last_trail_point = this_pos;
                        //printf("Added Point");
                }
                if (trail_points.size() > 24)
                {
                        //printf("Erased Point");
                        trail_points.erase(trail_points.begin());
                }

                velocity = sim_step[1];

        //std::cout << "SAT VEL (RELATIVE):" + (velocity - parentBody-
>Get_Tangential_Velocity()).Debug() + "\n";
                acceleration = sim_step[2];

        //std::cout << "\nSatellite Accel: " << Normalize(acceleration).Debug
();

                update_inspector();

                return 0;
        }

        std::string DebugBody() override //For debugging purposes...
        {
```

```cpp
        std::string text = "\n\n_____\n" + name + "\n|| Mass: " +
std::to_string(mass) + "\n|| Position: " + (position - parentBody-
>Get_Position()).Debug() + "\n|| Velocity: " + (velocity - parentBody-
>Get_Tangential_Velocity()).Debug() + "\n|| Period: " + std::to_string(Calcul
ate_Period() / (60 * 60 * 24)) + "days" + "\n_____\n\n";
                return text;
        }
};

int Body::Update_Satellites(float delta, float time_scale, std::vector<Body*>
* bodies_in_system)
{
        //Now update Satellites
        Clean_Up_Satellites();
        for (Satellite* sat : satellites)
        {
                sat->Update_Body(delta, time_scale, bodies_in_system);
        }
        return 0;
}

int Body::Add_Satellite(Satellite* sat)
{
        satellites.emplace_back(sat);
        std::cout << name << " has a new satellite: " << sat-
>name << " | Total number of satellites: " << satellites.size() << "\n";
        return satellites.size() - 1;
}

void Body::Create_Satellite()
{
        // TODO: Figure this out I guess!
        //Add_Satellite(Satellite("Moon", this, { 3.8E8, 0, 0 }, 7.3E22, 1.7E
5, { 0, -1200, 0 }, graphyte, false)); //Continue with this.
        Satellite* new_sat = new Satellite("Moon", this, { 3.844E8, 0, 0 }, 7
.3E22, 1.7E5, { 0, -1024, 0 }, graphyte, false);
        std::cout << new_sat->DebugBody();
        Add_Satellite(new_sat); //Continue with this.
}

void Body::Delete_Satellites()
{
        for (Satellite* sat : satellites)
        {
                sat->Delete();
        }
        Clean_Up_Satellites();
}

int Body::Clean_Up_Satellites()
{
        int length = satellites.size();
        for (int i = 0; i < length; i += 0) //This is an odd loop
        {
                if (satellites[i]->to_delete)
                {
                        satellites.erase(satellites.begin() + i);
                        length -= 1;
                }
```

```cpp
            else {
                    i++;
            }
        }
        return 0;
}

void Body::Close_Satellite_Inspectors()
{
        for (Satellite* sat : satellites)
        {
                sat->HideBodyInspector();
        }
}

void Body::Set_Mu(double _mu)
{
        mu = _mu;
        for (Satellite* s : satellites)
        {
                s->Set_Mu(mu);
        }
}

int Body::Draw_Satellites(Graphyte& g, Camera& c)
{
        for (Satellite* sat : satellites)
        {
                sat->Draw(g, c);

        }
        return 0;
}
#endif /*ORBITBODY_H*/
```

Orbyte_Data.h

```cpp
#pragma once
#ifndef ORBYTE_DATA_H
#define ORBYTE_DATA_H

#include<iostream>
#include<fstream>
#include "vec3.h"
#include <bitset>
#include "utils.h"
#include "OrbitBody.h"
#include <ShObjIdl_core.h>
#include <Windows.h>

struct OrbitBodyData
{
        //Information for storage
        std::string name; // Name of Body
        vector3 center; // Position of Body
        double mass; // Mass of Body
        double scale; // Scale of Body
        vector3 velocity; // Velocity of Body
```

```cpp
        uint8_t bytes_for_name; //So the first 8 bits of the file will tell u
s how many bytes the name contains. Name being the only var with "unlimited l
ength"
        OrbitBodyData(std::string _name = "", vector3 _center = { 0, 0, 0 },
double _mass = 1, double _scale = 1, vector3 _velocity = {0, 0, 0})
        {
                name = _name;
                center = _center;
                scale = _scale;
                velocity = _velocity;
                mass = _mass;
                bytes_for_name = (uint8_t)name.length();
        }
};

//A Hash table of orbit objects
struct OrbitBodyCollection
{
private:
        //101 has been chosen as it is prime and not too close to a power of
two
        // => Maximum of 101 bodies in storage!
        std::vector<OrbitBodyData> data = std::vector<OrbitBodyData>(101);
        int count = 0;

        int Hash(std::string name)
        {
                // Reverse the string. A palindrome could XOR itself
                std::string reverse = name;
                reverse_string(reverse, reverse.length() - 1, 0);

                // XOR to produce Hash
                std::string hash = bitwise_string_xor(name, reverse);
                int total = 0;

                for (int i = 0; i < hash.length(); i++)
                {
                        total += int(hash.at(i));
                }

                // Return Index To Write
                return (total % data.size());
        }

        void TryWrite(OrbitBodyData d, int index)
        {

        // Saving one empty space so that Reading does not recurse infinitely
                // Max bodies can store: 100
                if (count < data.size() - 1)
                {
                        if (data[index].name == "")
                        {

        // If address is empty. Bodies Cannot have no name!
                                data[index] = d;
                                count++;
                                return;
                        }
```

```cpp
                    else
                    {

        //Some recursion for collision avoidance with open addressing
                        TryWrite(d, (index + 1) % data.size());
                    }
                }
            }
            else {

        std::cout << "\nErr. Cannot write to OrbitBodyCollection => Hash Tabl
e is full!\n";
            }
        }

        OrbitBodyData TryRead(std::string name, int index)
        {
            if (data[index].name == "")
            {
                //Unsuccessful Read
                return data[index]; //Equivalent to NULL
            }
            if (data[index].name == name)
            {
                //Successful Read
                return data[index];
            }
            else {
                //Increment & read again

        return TryRead(name, (index + 1) % data.size()); //Careful with recur
sion depth!
            }
        }

public:
        void AddBodyData(OrbitBodyData new_data)
        {
            // Try write data to address generated by Hash() method
            data[Hash(new_data.name)] = new_data;

            // debug

        std::cout << "\n Adding body data with hash: " << Hash(new_data.name)
                << "\n" << "Testing fetch [If blank, problem!]: "
                << GetBodyData(new_data.name).name << "\n";
        }

        OrbitBodyData GetBodyData(std::string name)
        {
            // Return OrbitBodyData stored in the Hash table
            // at address given by Hash(name)
            return TryRead(name, Hash(name));
        }

        std::vector<OrbitBodyData> GetAllOrbits()
        {
            std::vector<OrbitBodyData> result;
            for (OrbitBodyData d : data)
            {
                if (d.name != "")
```

```cpp
                {
                        result.push_back(d);
                }
            }
            return result;
        }
};

struct SimulationData
{
        double cb_mass = 0; // Mass of center body
        double cb_scale = 0; // Scale of center body
        OrbitBodyCollection obc; // All orbits
        vector3 c_pos; // Camera position
};

class DataController
{
public:
        // Write simulation data to a .orbyte file at specified path.
        int WriteDataToFile(SimulationData sd, std::vector<std::string> bodie
s_to_save, std::string path)
        {
                double no_orbits = bodies_to_save.size();
                path = "simulations/" + path;

                std::cout << "Writing data to file: " << path << "\n";

                std::ofstream out(path, std::ios::binary | std::ios::out);;
                if (!out)
                {
                        std::cout << "\nERR. Writing to file failed.";
                        return 1; // Problem
                }
                std::cout << (char*)&sd;
                uint8_t my_size = sizeof(sd);

        out.write((char*)&sd.cb_mass, sizeof(double)); // Center Body Mass

        out.write((char*)&sd.cb_scale, sizeof(double)); // Center Body Scale

        out.write((char*)&sd.c_pos, sizeof(vector3)); // Camera Position

        out.write((char*)&no_orbits, sizeof(double)); // Number Of Orbits

                for (int i = 0; i < bodies_to_save.size(); i++)
                {
                        //Access Data via hashing algorithm & hash table

        OrbitBodyData data = sd.obc.GetBodyData(bodies_to_save[i]);


        double bfn = data.bytes_for_name; // Number of characters in name
                        out.write((char*)&bfn, sizeof(double));

                        std::string n = data.name;
                        for (char n_char : n) //Write Name
                        {
                                out.write((char*)&n_char, sizeof(char));
```

```cpp
                }

                vector3 c = data.center; // Body position
                out.write((char*)&c, sizeof(vector3));

                double m = data.mass; // Body Mass
                out.write((char*)&m, sizeof(double));

                double s = data.scale; // Body Scale
                out.write((char*)&s, sizeof(double));

                vector3 v = data.velocity; // Body Velocity
                out.write((char*)&v, sizeof(vector3));
            }
            out.close();

            std::cout << "\nBytes: " << my_size + 1;
            return 0;
        }

    // Read simulation data from a .orbyte file at given path.
    SimulationData ReadDataFromFile(std::string path)
    {
            SimulationData sd; // New Simulation Data
            double no_orbits; // Number of orbits to read
            path = "simulations/" + path;

            std::ifstream in(path, std::ios::binary | std::ios::in);
            if (!in)
            {
                    std::cout << "\nERR. Reading from file failed.";

    return sd; //But center mass will be 0, so known error.
            }
            // Mass

    in.read((char*)&sd.cb_mass, sizeof(double)); std::cout << "\nReading
mass: " << sd.cb_mass;
            // Scale

    in.read((char*)&sd.cb_scale, sizeof(double)); std::cout << "\nReading
 scale: " << sd.cb_scale;
            // Camera Position

    in.read((char*)&sd.c_pos, sizeof(vector3)); std::cout << "\nReading c
_pos: " << sd.c_pos.Debug();
            // Number of Orbits

    in.read((char*)&no_orbits, sizeof(double)); std::cout << "\nReading N
o. Orbits: " << no_orbits;

            // Iterate through orbits
            for (int i = 0; i < no_orbits; i++)
            {
                    OrbitBodyData data; // New Data

                    double bfn; // Bytes for Name
                    in.read((char*)&bfn, sizeof(double));
                    data.bytes_for_name = bfn;
```

```cpp
        std::cout << "\nReading No. bytes: " << (double)data.bytes_for_name;

                    std::string name = ""; // Reading name
                    for (int j = 0; j < data.bytes_for_name; j++)
                    {
                            char c; // Character in name
                            in.read((char*)&c, sizeof(char));
                            name += c;
                    }

            data.name = name; std::cout << "\nReading name: " << name;
                    // Position
                    in.read((char*)&data.center, sizeof(vector3));
                    // Mass
                    in.read((char*)&data.mass, sizeof(double));
                    // Scale
                    in.read((char*)&data.scale, sizeof(double));
                    // Velocity
                    in.read((char*)&data.velocity, sizeof(vector3));
                    // Add to OrbitBodyCollection
                    sd.obc.AddBodyData(data);
            }

            in.close();


            if (!in.good())
            {
        std::cout << "\n\nErr. Reading from .orbyte file failed!\n\n";
            }

            return sd;
        }
};

#endif /*ORBYTE_DATA_H*/
```

Utils.h

```cpp
#pragma once

#ifndef UTILS_H
#define UTILS_H

#include "OrbitBody.h"
#include <vector>

//Take two strings and XOR them. String a defines length of output string.
std::string bitwise_string_xor(std::string a, std::string b)
{
        std::stringstream stream;
        for (int i = 0; i < a.length(); i++)
        {
                stream << (a.at(i) ^ b.at(i));
        }
```

```cpp
        return stream.str();
}

//Reverses a string using a recursive "pincer" index method, i decrements and
 j increments.
void reverse_string(std::string& a, int i, int j)
{
        if (i <= j) { return; }
        std::swap(a[j], a[i]);
        reverse_string(a, i - 1, j + 1);
}

#endif /*UTILS_H*/
```

Vec3.h

```cpp
#ifndef VEC3_H
#define VEC3_H
struct vector3
{
        double x, y, z;

        vector3 operator*(double right)
        {
                vector3 result = { x * right, y * right, z * right };
                return result;
        }

        vector3 operator+(double right)
        {
                vector3 result = { x + right, y + right, z + right };
                return result;
        }

        vector3 operator-(double right)
        {
                vector3 result = { x - right, y - right, z - right };
                return result;
        }

        vector3 operator+(vector3 v)
        {
                vector3 result = { x + v.x, y + v.y, z + v.z };
                return result;
        }

        vector3 operator-(vector3 v)
        {
                vector3 result = { x - v.x, y - v.y, z - v.z };
                return result;
        }

        double operator*(vector3 v)
        {
                double result = (x*v.x) + (y*v.y) + (z*v.z);
                return result;
        }
```

```cpp
        vector3* operator=(const vector3& v)
        {
                x = v.x;
                y = v.y;
                z = v.z;
                return this;
        }

        std::string Debug()
        {

        return std::to_string(x) + ", " + std::to_string(y) + ", " + std::to_
string(z);
        }

};

// Length of a vector3. Thank you Pythagoras.
double Magnitude(vector3 vec)
{
        return sqrt((vec.x * vec.x) + (vec.y * vec.y) + (vec.z * vec.z));
}

// Distance between two points in 3D space
double Distance(vector3 vecFrom, vector3 vecTo)
{
        vector3 a = vecTo - vecFrom;
        double distance = Magnitude(a);

        return distance;
}

// "Dot Product" of two vectors.
double Scalar_Product(vector3 a, vector3 b)
{
        return (a.x * b.x) + (a.y * b.y) + (a.z * b.z);
}

// Vector3 in same direction but with magnitude 1
vector3 Normalize(vector3 vec)
{
        double mag = Magnitude(vec);
        vector3 norm = {
                vec.x / mag,
                vec.y / mag,
                vec.z / mag
        };
        return norm;
}

#endif /*VEC3_H*/
```