

Orbyte

Author: Joshua James-Lee

Candidate number: <Your AQA number>

Centre name: Magdalen College School Oxford

Centre number: #####

Qualification code: 7517D

Date: May 2023

Analysis	4
Background information	4
Investigation	4
Interviews/ Observations	4
Questionnaires	5
Overview of problem	7
Limitations and constraints	7
Input, Process, Storage, Output (IPSO)	8
Data dictionary	8
Data volumes	9
Context diagrams	9
Data Flow diagrams (DFDs)	Error! Bookmark not defined.
System flowcharts	9
Entity Relationship diagram (ERD)	10
Entity Attribute Model (EAM)	10
Objectives	10
Potential solutions	14
Documented design	14
Database design	14
Entity Relationship diagram (ERD)	15
Entity Attribute Model (EAM)	15
Data Flow diagrams (DFDs) - to 3NF	15
Data dictionaries	15
SQL statements	16
Overall System Design	16
Top-down diagram	17
Input, Process, Storage, Output (IPSO)	17
Data dictionaries	18
Record structures	19
Data structures	19
OOP class design	20
Interface design	21
Algorithms	22
Hardware selection/design (if appropriate)	26
Security and integrity of data	Error! Bookmark not defined.
Technical solution	26
Database solution	33
Evidence of setting up tables and relationships	Error! Bookmark not defined.
System overview	33
Evidence of classes	38
Evidence of complete code listings and user interface	57
Evidence of procedures and variables	Error! Bookmark not defined.

Firstname Surname

Candidate number: #####

Centre number: #####

Testing	58
Testing strategy	58
Test plan	58
Test evidence	58
Video evidence	58
Failed tests	58
Testing qualitative objectives	58
Evaluation	59
Review against objectives	59
Analysis of independent feedback	59
Potential improvements	59
References	60

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.” ||

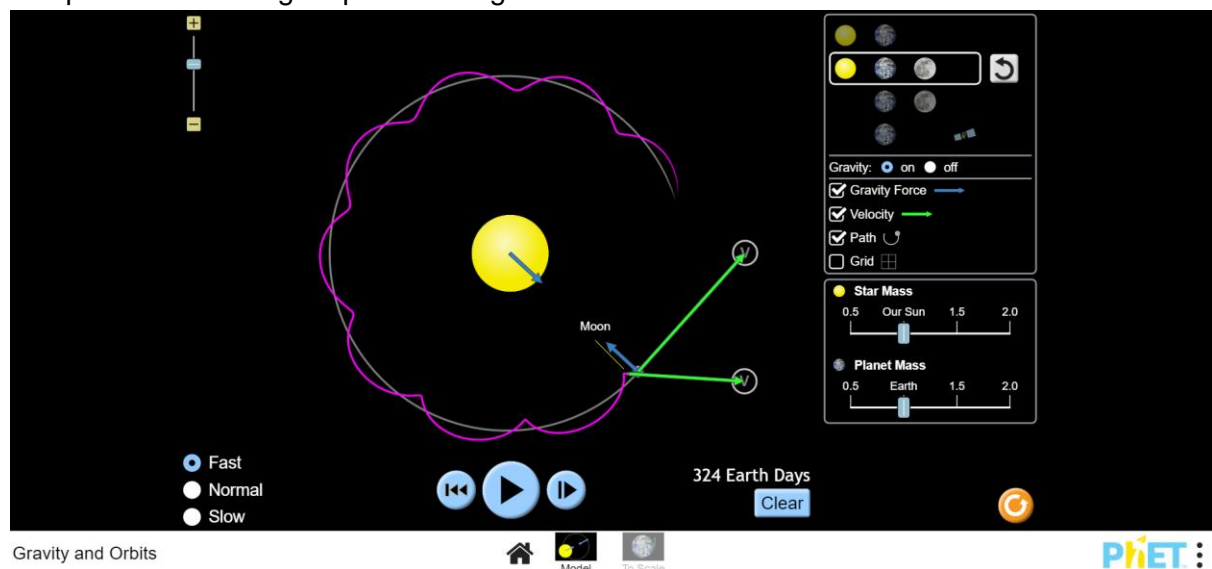
Analysis of the current industry standard

The following investigation¹ 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.



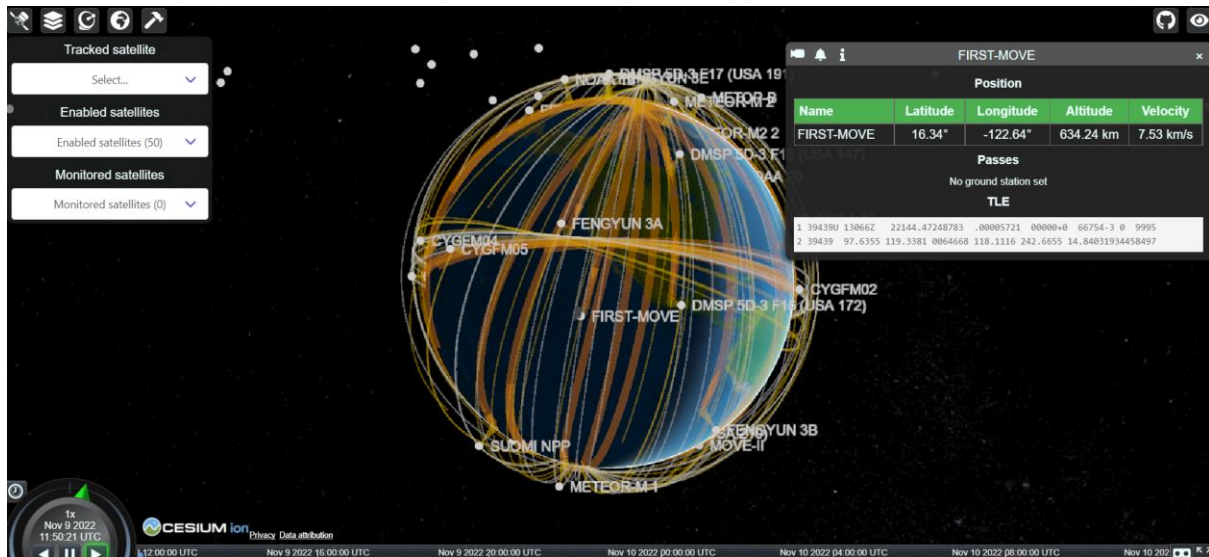
Satellite visualization

“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.

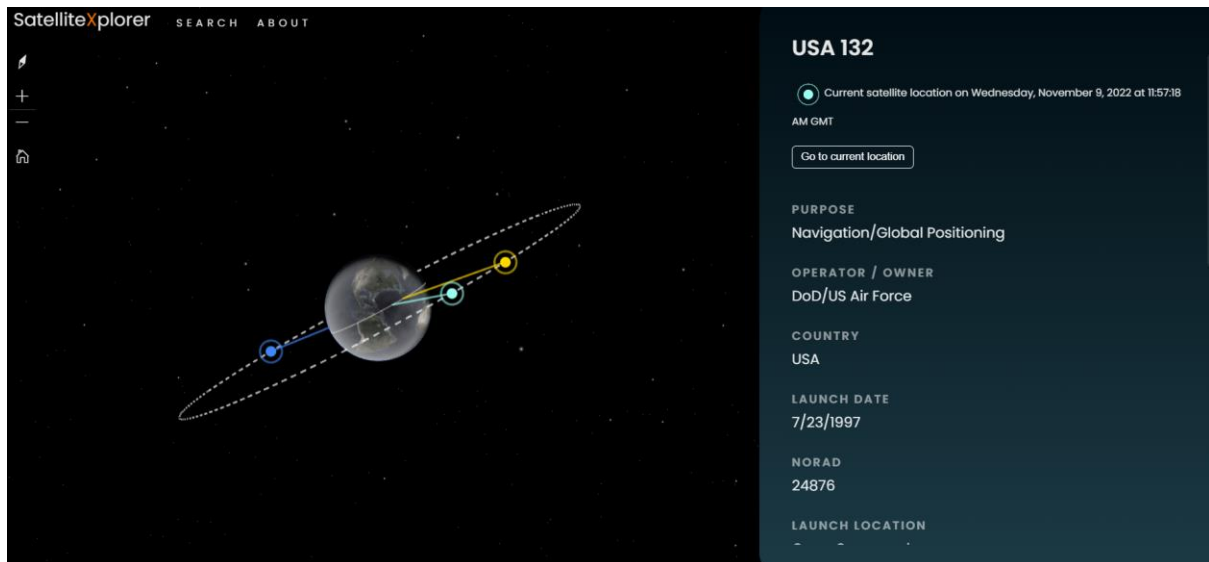
compact user interface, especially the ability to select an individual orbiting body to display more information about it.



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: <ul style="list-style-type: none"> - 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: <ul style="list-style-type: none"> - Force Vectors - Velocity Vectors - Path Orbiting Body Information: <ul style="list-style-type: none"> - Acceleration - Velocity - Period - Radius 	Interview and current solution analysis.
Storage	Update Queue: <ul style="list-style-type: none"> - Orbiting body objects GUI elements Last simulation state and setup.	
Processing	<ul style="list-style-type: none"> - 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)

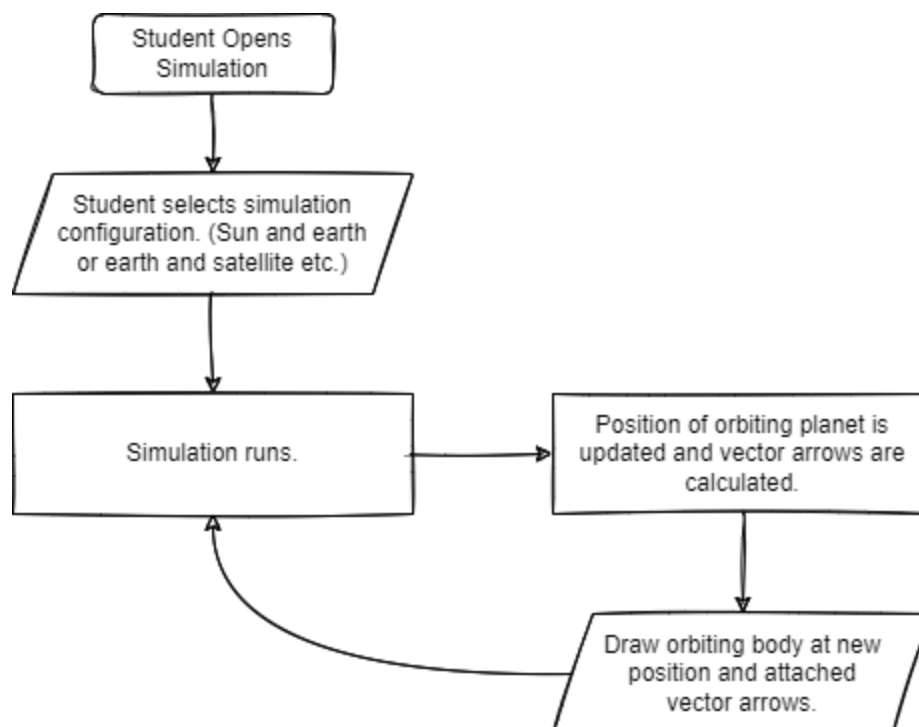
Mass	Float		5.972×10^{24}
Position	Vector		(1, 1, 1)
Gravitational Constant	Float		6.6743×10^{-11}
Acceleration	Vector		(1,1,1)
Time Step	Float		100

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 flowcharts



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.

$$\overrightarrow{a_{gravity}} = \dot{\vec{v}} = \ddot{\vec{s}}$$

$$\vec{r}(t) = \int \vec{v} dt = \iint \vec{a} dt$$

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.
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.
0.1.3.0	Create Simulation Class with attributes and accessors	Debugging procedures and accessor methods to test the class and attributes.
0.1.3.1	Outline Simulation Class Update Method	Comments and references to empty

		procedures.
Version 0.2	Orbital Simulation Mathematics	
0.2.1.0	Implement Runge-Kutta 4 step in a procedure.	Test with known values and compare output to what the answer should be.
0.2.2.0	Integrate new RK4 method into the main program update procedure.	Test with debug methods.
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.
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.	Debug methods to confirm attributes set correctly.
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.
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).
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.	See 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.

Version 0.5	Satellite Child Class	
0.5.1.0	<p>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.</p> <p><i>(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.)</i></p>	Test with 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 and a list of bodies in 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.
0.7.3.0	<p>Add a pop-out window that functions as an "inspector."</p> <p><i>This will describe attributes of the object currently in focus.</i></p>	Placeholder window.
Version 0.8	User Interface II	
0.8.1.0	Connect simulation to the User Interface. UI	See object attributes

	should access and display orbiting body class 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 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.
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.

Documented design

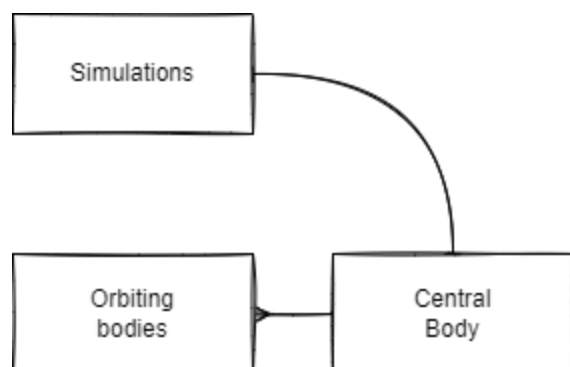
Database design

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.

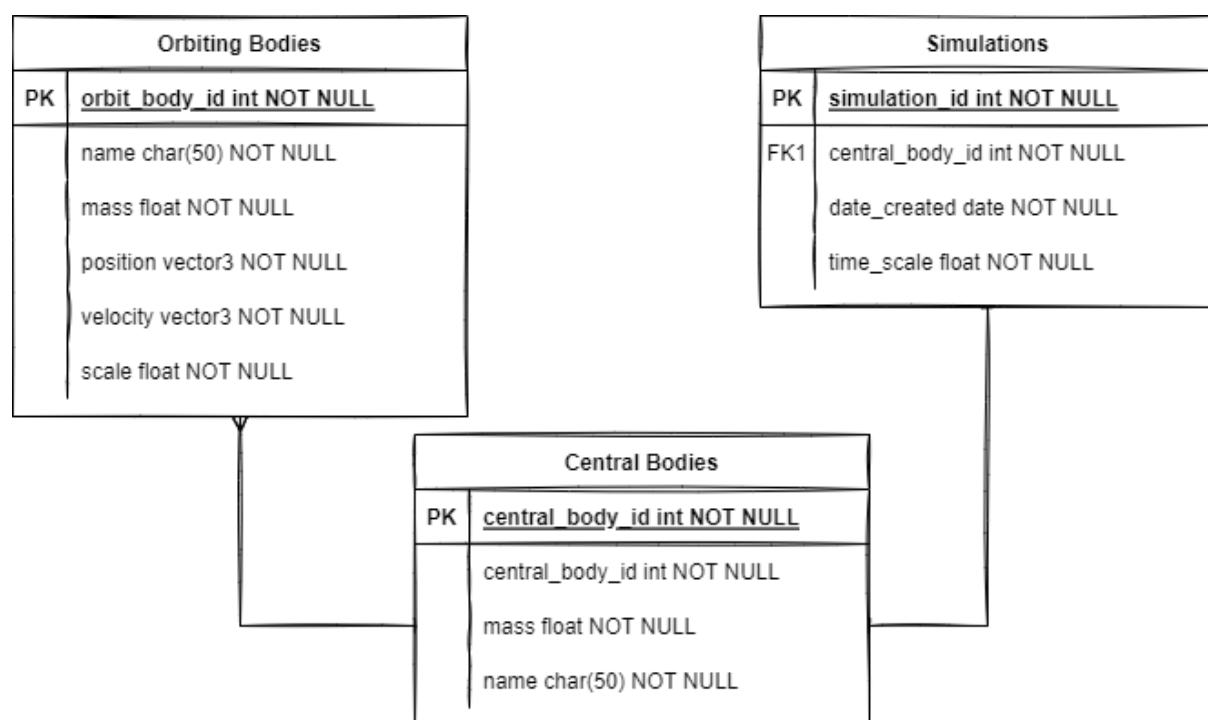
Saved simulations can be used in a classroom situation where a template is opened by students, previously made by the teacher before the class started.

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)

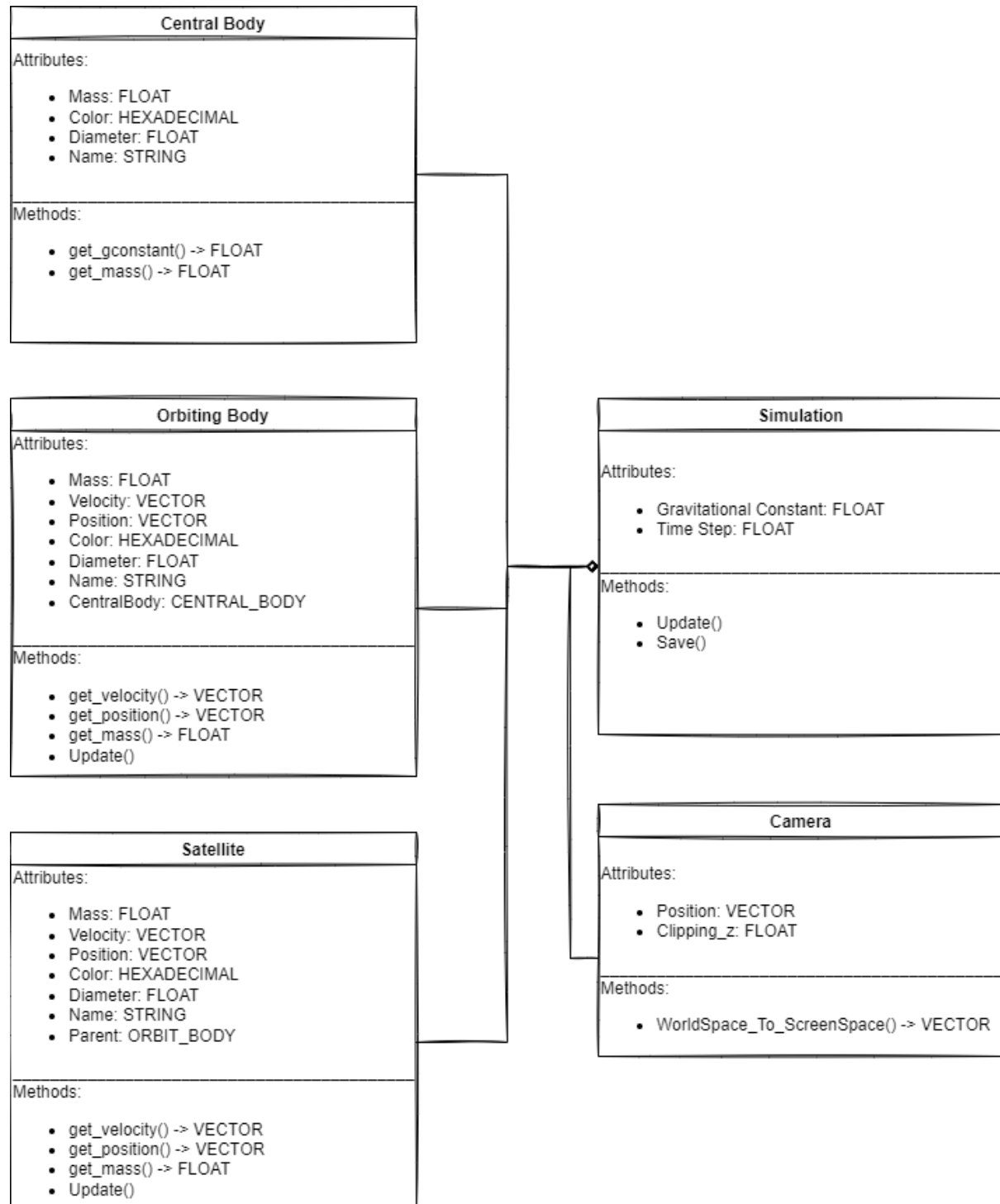


Entity Attribute Model (EAM)



Overall System Design

UML Diagram



Application Process Diagram

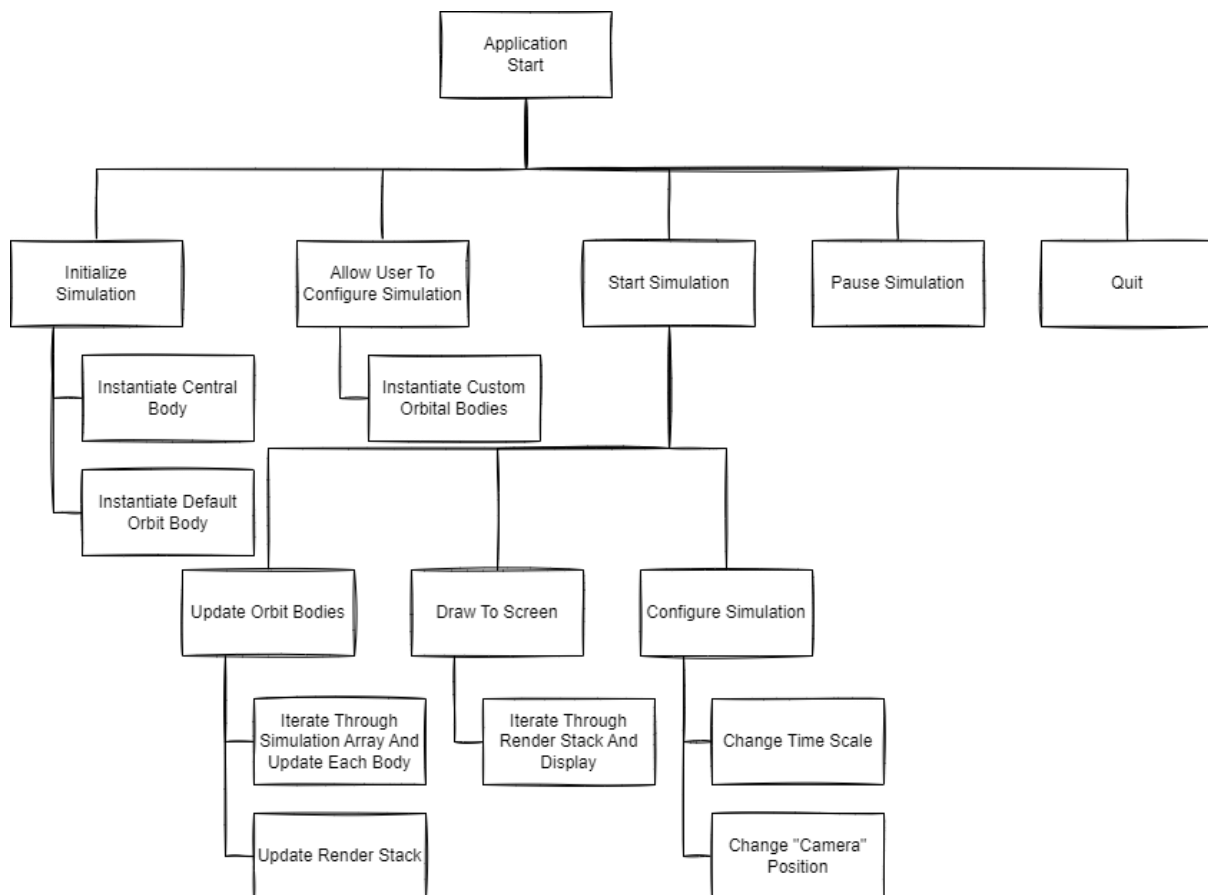


Figure 1: Process diagram detailing application flow

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	<ul style="list-style-type: none"> - ID (path) - Display Resolution - Central Object Mass - Central Object Scale
Input	Instantiating Orbiting Objects	<ul style="list-style-type: none"> - Mass

		<ul style="list-style-type: none"> - Scale - Initial Position - Initial Velocity (<i>Can be calculated such that a circular orbit is produced</i>) - Name
Input	Runtime Configuration	<ul style="list-style-type: none"> - Change time-scale - Camera Position / “zoom”
Process	Update Orbiting Bodies	<ul style="list-style-type: none"> - 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	<ul style="list-style-type: none"> - 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	<ul style="list-style-type: none"> - Draw the shapes representing the orbiting bodies to their respective positions.
Output	Display body information	<ul style="list-style-type: none"> - Display the appropriate information pertaining to a selected orbiting body (if any).
Output	Draw attribute vectors	<ul style="list-style-type: none"> - 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

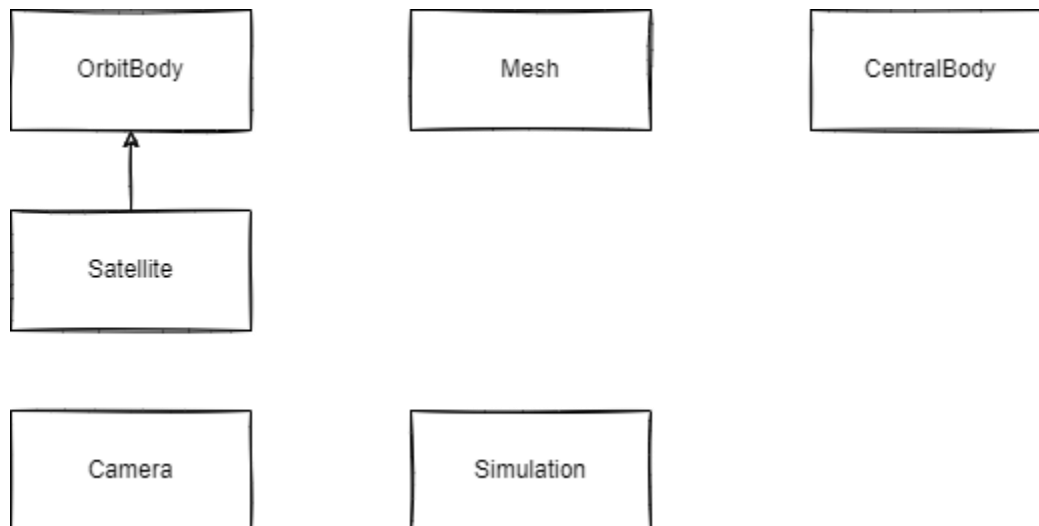
OrbitBodyData

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 class² 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.

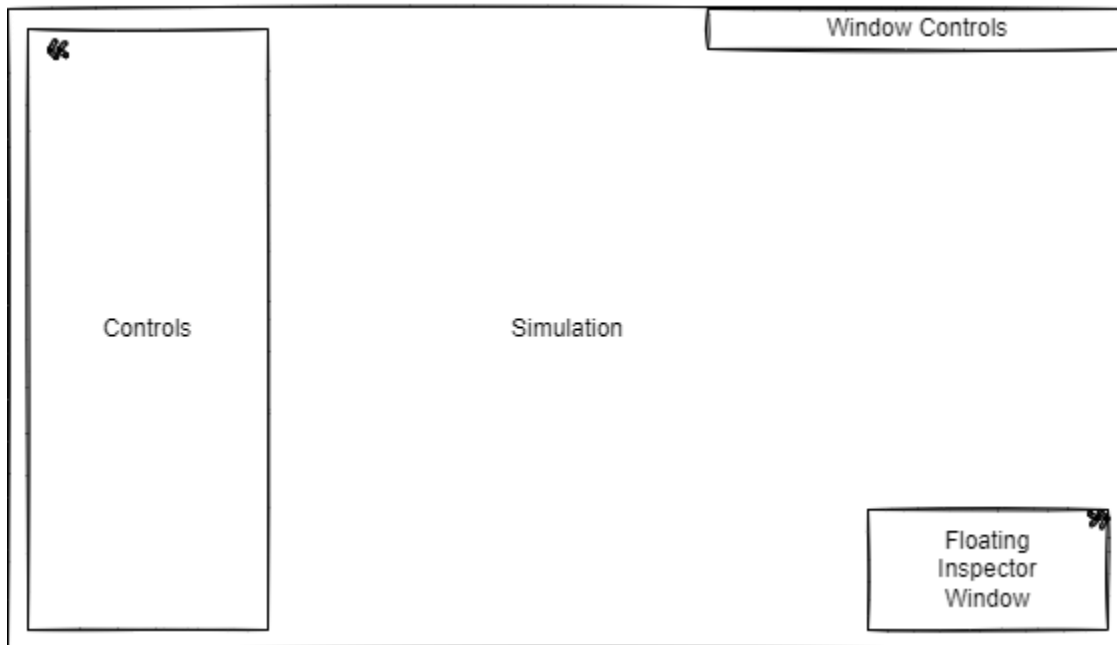


Figure 2: GUI Design

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

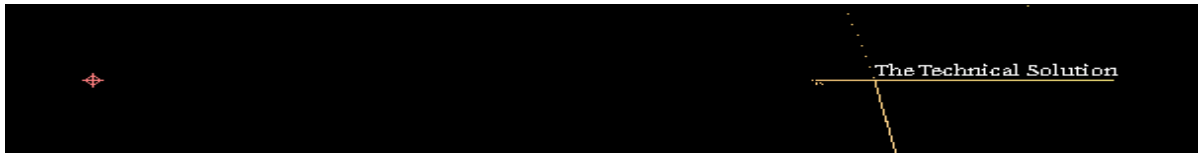
    Return (total % max_orbits)
EndFunction
```

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 classes³. 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.

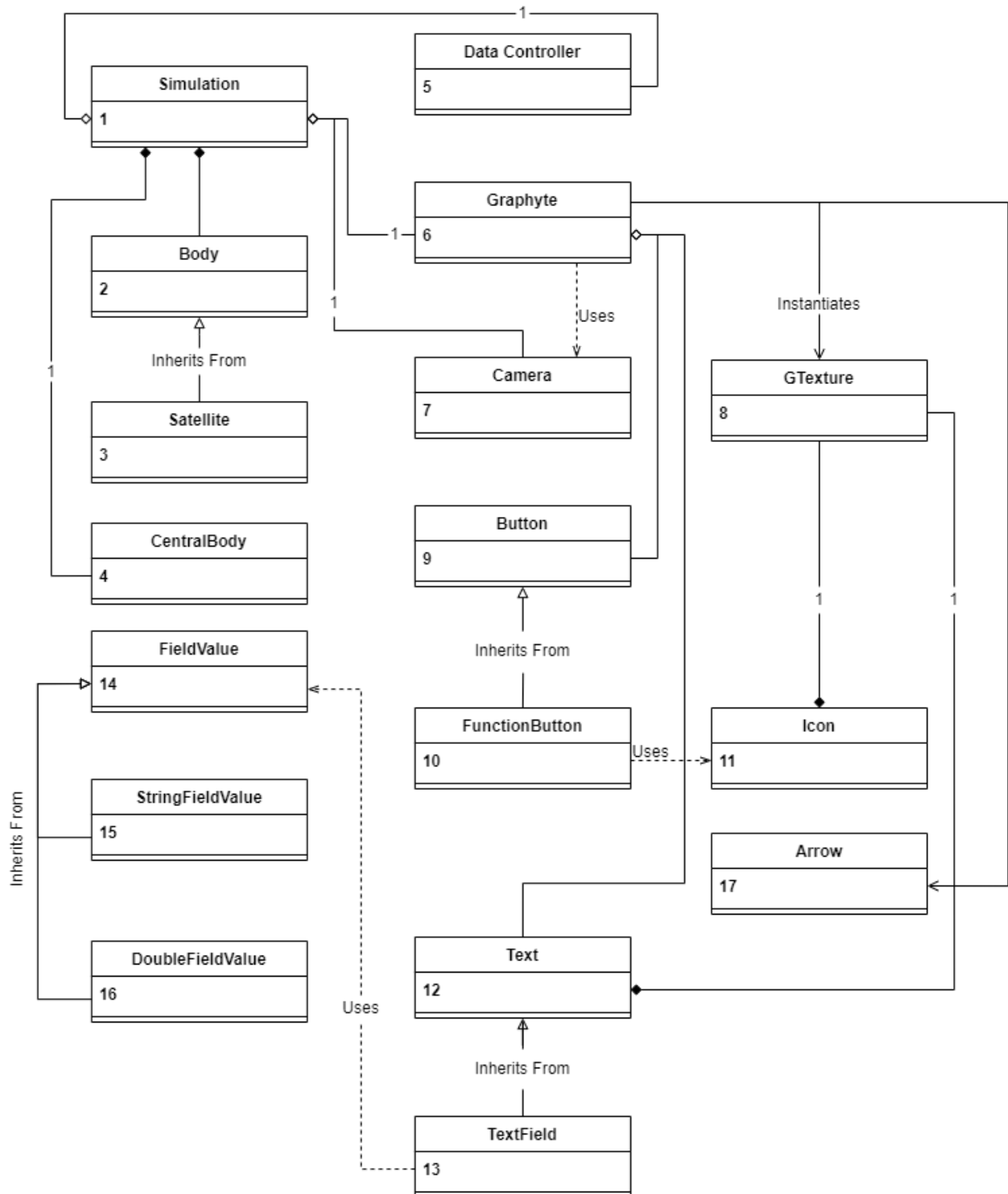


Figure 3: Technical Solution UML Diagram

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: $G \times M_{\text{large}}$

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.

	<ul style="list-style-type: none"> - 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> 	<ul style="list-style-type: none"> + CreateIcon() + GetTextParams() + AddTextToRenderQueue() + AddIconToRenderQueue() + Get_Screen_Dimensions() + Get_Number_Of_Points() + pixel() + line() + draw() + free()
7: Camera	<ul style="list-style-type: none"> - camera_rotation : VECTOR3 + position : VECTOR3 + clipping_z : FLOAT 	<ul style="list-style-type: none"> + Camera() + RotateCamera() + rotate() + WorldSpaceToScreenSpace()
8: GTexture ⁶	<ul style="list-style-type: none"> - MTexture : SDL_TEXTURE - renderer : SDL_RENDERER - font : TTF_FONT - mWidth : INTEGER - mHeight : INTEGER 	<ul style="list-style-type: none"> + GTexture() + GTexture(GTexture source)⁷ + loadFromFile() + loadFromRenderedText() + reset_texture() + free() + render() + getWidth() + getHeight()
9: Button	<ul style="list-style-type: none"> - position : VECTOR3 - width : INTEGER - height : INTEGER - left_wall_offset : INTEGER - function : FUNCTION<VOID()> # enabled : BOOLEAN 	<ul style="list-style-type: none"> # CallFunction() # AttachFunction() + Button() + SetDimensions() + SetPosition() + SetEnabled() + Clicked()
10: FunctionButton (INHERITS FROM BUTTON)	<ul style="list-style-type: none"> - icon : ICON 	<ul style="list-style-type: none"> + FunctionButton() + CheckForClick() + SetEnabled() + free()
11: Icon	<ul style="list-style-type: none"> - texture : GTEXTURE + pos_x : INTEGER + pos_y : INTEGER + path_to_image : STRING + visible : BOOLEAN + dimensions : VECTOR<INTEGER> 	<ul style="list-style-type: none"> + Icon() + Render() + SetPosition() + SetDimensions() + GetDimensions() + free()
12: Text	<ul style="list-style-type: none"> # texture : GTEXTURE + pos_x : INTEGER + pos_y : INTEGER + text : STRING 	<ul style="list-style-type: none"> + 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.

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 performance⁸ across all hardware is ensured.

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

⁸ Performance and simulation optimisation is discussed further on.

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*” vector⁹.

```
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.

```
// 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 se, however by removing the need to re-enqueue objects every frame, significant performance has been preserved.

Orbit Body Class

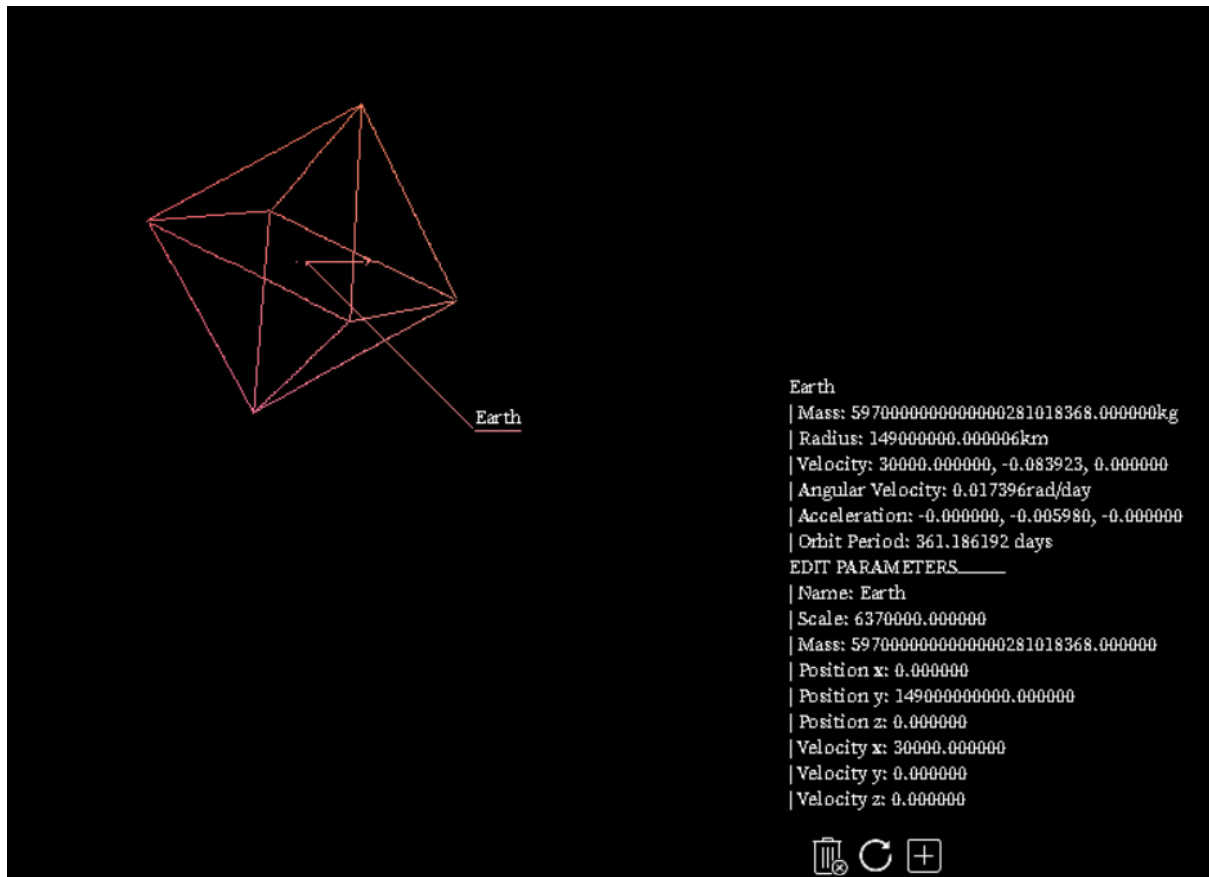


Figure 5 : Example orbit body with inspector enabled.

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, Graphite& g, bool override_velocity = false):
    graphite(g),
    ScaleFV(_scale, [this](){ this->RegenerateVertices(); }), MassFV(_mass), NameFV(_name, [this](){ this->Rename(); }), // Scale Field 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 recommended 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

```
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

```
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

```
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.

```
/// <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

```
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.

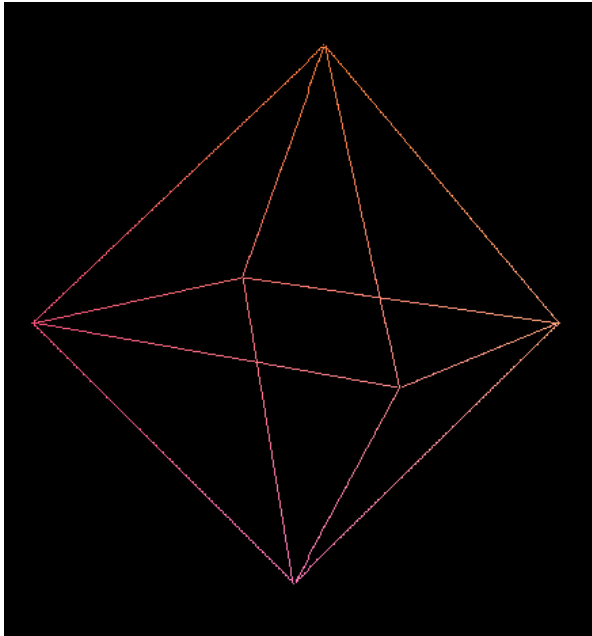


Figure 6: Orbit Body Geometry

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

```
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.

```
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.

```
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.

```
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 generation¹⁰ orbits and 2nd generation orbits are the geometries generated in the virtual **Generate_Vertices** method.

```
//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},

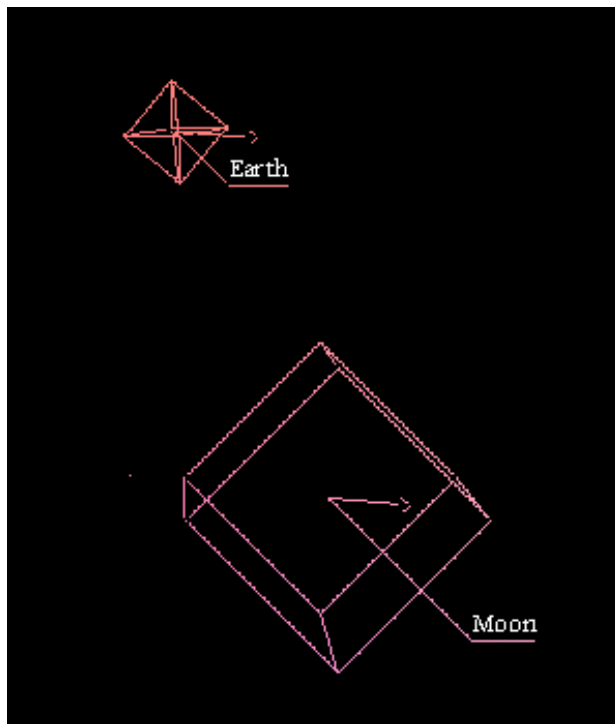
        {0, 4},
        {1, 5},
        {2, 6},
        {3, 7}
    };
    mesh.edges = _edges;

    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.


```

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:

```

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.

```

// 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]**

```
//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 crucial¹¹ 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.

```
bool Clicked(int x, int y)
{
    // (0<AM.AB<AB.AB) ^ (0<AM.AD<AD.AD) Where M is a point we're checking
    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 }; //All the vertices
    std::cout << "\n Dimensions" << width << "|" << height;
    vector3 M = { x, y, 0 };

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

    bool in_area = 0 <= AB*AM && AB*AM <= AB*AB && 0 <= BC*BM && BC*BM <= BC*BC;
    std::cout << "\n RESULT: " << in_area;
    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.

Function Buttons

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

```
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.

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).

```
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.

```
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.

```
private:
double* value = NULL; // This is the pointer to the variable the input field is associated with. E.g. time step or object mass
bool ValidateValue(std::string content)
{
    try
    {
        std::regex dbl_regex("(~+)?([0-9]+\\.?[0-9]+|([0-9]+)*)");
        if (std::regex_match(content, dbl_regex))
        {
            double test_validity = atof(content.c_str());
            std::cout << content << "=>" << test_validity;
            if (test_validity == NULL) // Will not allow zero!
            {
                throw(content); // Knew I was tired when I found the "throw" and "catch" system the funniest thing ever invented
            }
            else {
                return true;
            }
        }
        else {
            throw(content);
        }
    }
    catch (std::string bad)
    {
        std::cout << "\n Bad input recieved: " << bad; // Its bad
        return false;
    }
}
```

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.

```
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.

```
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.

```
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.

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

Below shows the validation method.

```
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.

```
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.

```
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 path¹².

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

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.

```
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

```
//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.

```
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:

```
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**.

```

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.

```

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.

```

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.

```

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.

```

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

```
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.

```
// 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.

```

// 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

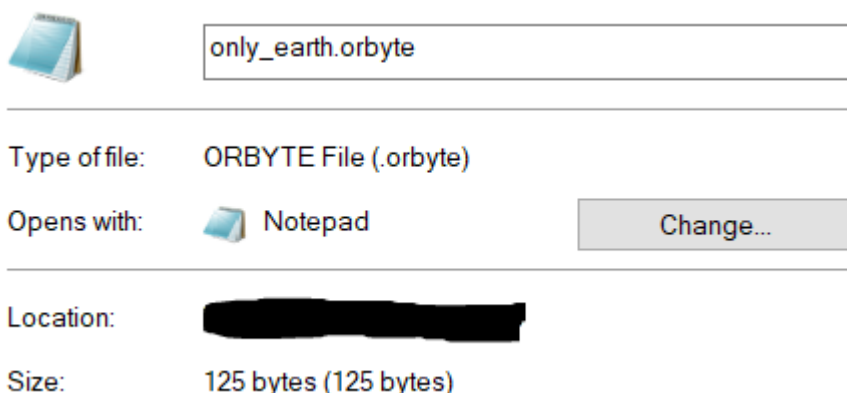


Figure 9 : Example .orbyte file with file size

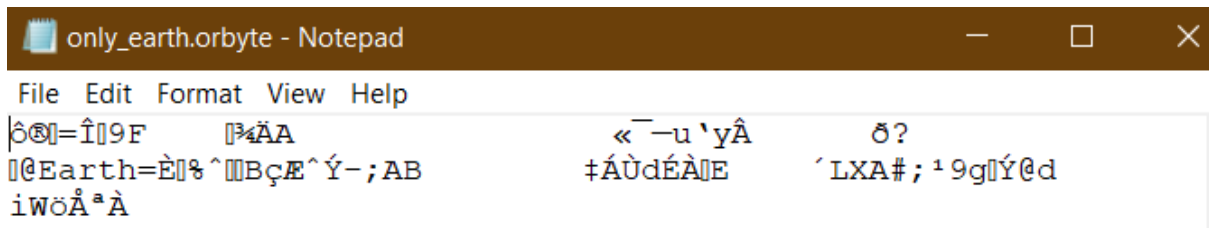


Figure 10 : Example .orbyte file viewed with notepad.

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.

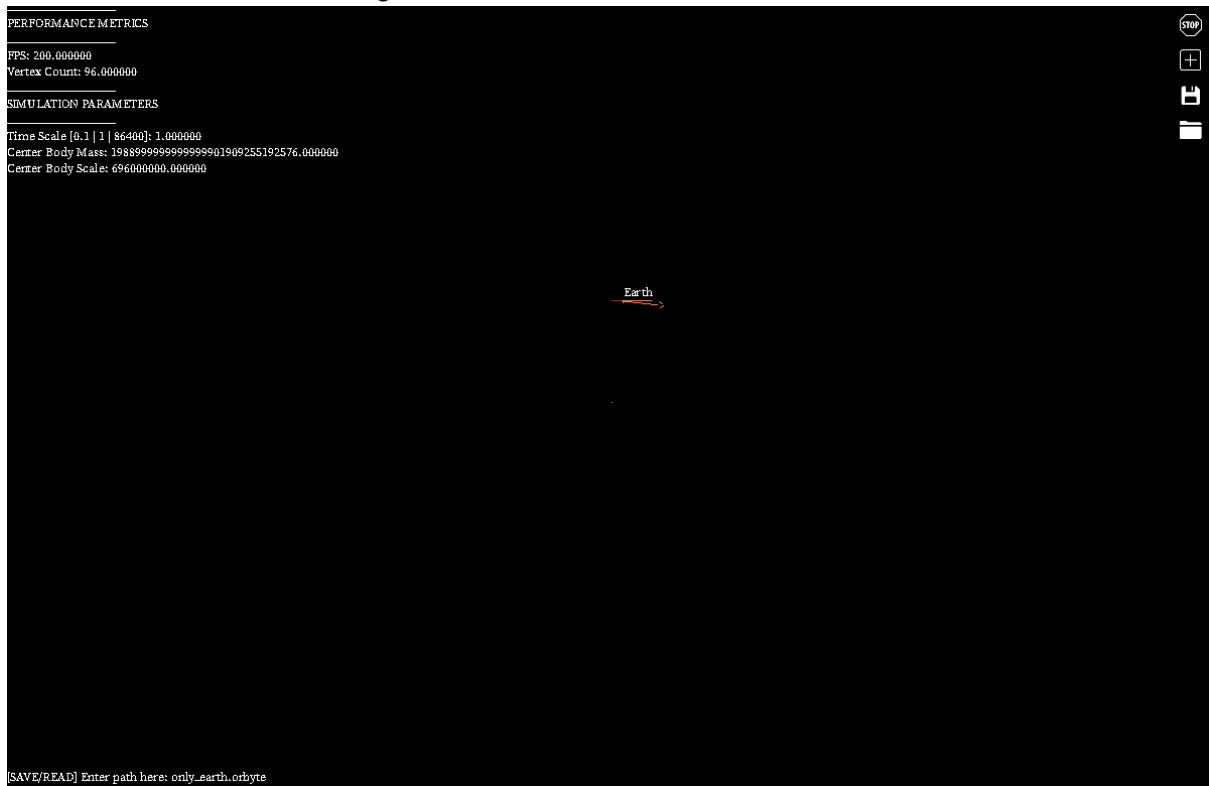


Figure 11 : Current state of simulation when saving.

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"
```


Firstname Surname

Candidate number: #####

Centre number: #####

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

R1

Test evidence

<Read NEA Workbook pg 51>

Video evidence

<Read NEA Workbook pg 52>

Failed tests

<Read NEA Workbook pg 52>

Testing qualitative objectives

<Read NEA Workbook pg 53>

Evaluation

Review against objectives

<Read NEA Workbook pg 55 to 56>

Analysis of independent feedback

<Read NEA Workbook pg 56>

Potential improvements

<Read NEA Workbook pg 57>

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]