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**](#_996nhkt5neaj)4

[Background information](#_z5rfcvu4g4kg) 4

[Investigation](#_3uqdelmd4p4t) 4

[Interviews/ Observations](#_cj5q1nveosu1) 4

[Questionnaires](#_s0rdz1puhh61) 5

[Overview of problem](#_bffb6ng2nmkp) 7

[Limitations and constraints](#_iza6jx7bb8ku) 7

[Input, Process, Storage, Output (IPSO)](#_cx8xz5tqa30c) 8

[Data dictionary](#_eauprok6cvgk) 8

[Data volumes](#_820ddpmrqz6w) 9

[Context diagrams](#_gacmmw1s7bxe) 9

[Data Flow diagrams (DFDs)](#_9rqeboxflyh9) **Error! Bookmark not defined.**

[System flowcharts](#_p8a1npl3rplt) 9

[Entity Relationship diagram (ERD)](#_rck1rmowmn0h) 10

[Entity Attribute Model (EAM)](#_24bzt9ts6zvv) 10

[Objectives](#_o5d4wpuwks3s) 10

[Potential solutions](#_3ihg0h4skyxp) 15

[**Documented design**](#_hgl1fldq4hn4)15

[Database design](#_dlwshcmg54nm) 15

[Entity Relationship diagram (ERD)](#_e3i33etf9ztj) 16

[Entity Attribute Model (EAM)](#_nwp4zgz72d4d) 16

[Data Flow diagrams (DFDs) - to 3NF](#_wd2gb8xczb1c) 16

[Data dictionaries](#_woyyratqh5uf) 16

[SQL statements](#_i9efkrxknz0c) 17

[Overall System Design](#_ftqfdac176fd) 17

[Top-down diagram](#_ezxxnhlvmx1) 18

[Input, Process, Storage, Output (IPSO)](#_8qxc74jp5mdl) 18

[Data dictionaries](#_63nzsopuq8o2) 19

[Record structures](#_udgd5k1m5zu4) 20

[Data structures](#_10lq8ufqj9cb) 20

[OOP class design](#_j68xy5rxqt9e) 21

[Interface design](#_20mmmw7fw4hx) 22

[Algorithms](#_m0t983j3tc8x) 23

[Hardware selection/design (if appropriate)](#_kcxwapyfdgtf) 27

[Security and integrity of data](#_65b7jhy21s0u) **Error! Bookmark not defined.**

[**Technical solution**](#_tch8enov93k6)27

[Database solution](#_6nmjw1go3rdd) 34

[Evidence of setting up tables and relationships](#_19knckwrtg2d) **Error! Bookmark not defined.**

[System overview](#_y05b9yffxaq9) 34

[Evidence of classes](#_11nwrnt4xvre) 39

[Evidence of complete code listings and user interface](#_nvvjw2q0eiou) 59

[Evidence of procedures and variables](#_7a0309aepy1h) **Error! Bookmark not defined.**

[**Testing**](#_ktn1gre1wao2)60

[Testing strategy](#_6bfawt6yppdh) 60

[Test plan](#_ljwkjglenn7w) 60

[Test evidence](#_c9uux4mp4b1j) 70

[Video evidence](#_5fap8ll7h0vd) **Error! Bookmark not defined.**

[Failed tests](#_g8p7t3q2hzf) 70

[Testing qualitative objectives](#_dpdfqzcaqhk8) **Error! Bookmark not defined.**

[**Evaluation**](#_hqrhe2ykzmzx)71

[Review against objectives](#_5r0y44i355p4) 72

[Analysis of independent feedback](#_2sn03drvyw8z) 72

[Potential improvements](#_r7q4bz4wkorq) 72

[**References**](#_di7i6av840s0)73

# Analysis

## Background information

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

## Investigation

### Interview with a Physics Teacher

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

*How does the teaching of gravitational fields and orbits benefit from the use of graphical computer simulations?*

|| “Gravitational fields can be effectively taught through simulations, especially sandbox-style simulations, as a result of the ability to become hands-on with the Physics and transport gravity, which normally is not viable to show many demonstrations of in the lab, into a practical science for the students.” ||

*Can you think of any limitations with current simulations used in classes? (e.g. the Phet Orbit Simulation)*

|| “Relatively limited options in the PhET simulation for which satellites are in play and how many of them you would want (although this may become computationally expensive quite quickly).” ||

*What are some features you like to see in a 3D orbital simulation?*

|| “Free movement of the camera, multiple possible bodies and their interactions.” ||

*Should the simulation include planets, satellites, or both?*

|| “Both.” ||

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

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

### Analysis of the current industry standard

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

**Classroom**

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

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

A picture containing graphical user interface

Description automatically generated

**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 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.

A screenshot of a computer

Description automatically generated with medium confidence

## Overview of problem

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

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

## Limitations and constraints

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

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

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

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

## Input, Process, Storage, Output (IPSO)

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

## Data dictionary

*Abstract data types: Vector*

|  |  |  |  |
| --- | --- | --- | --- |
| **Data item** | **Data type** | **Validation** | **Sample data** |
| Velocity | Vector |  | (1, 1, 1) |
| Mass | Float |  | 5.972 × 10^24 |
| Position | Vector |  | (1, 1, 1) |
| Gravitational Constant | Float |  | 6.6743 × 10^-11 |
| Acceleration | Vector |  | (1,1,1) |
| Time Step | Float |  | 100 |

## Data volumes

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

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

## System flowchart



## Runge-Kutta 4

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

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.

Where:

From this we extrapolate the ODE Method:

**

This method is then used within each RK4 step:



## Objectives

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

Its ability to directly interface with graphics hardware and low-level nature will make it a suitable environment to develop a performant simulation, giving me control over how memory and other resources are being used, and freedom to implement graphics and UI how I choose to, in order to best accomplish the goal of making an educational general orbit simulation.

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

|  |  |  |
| --- | --- | --- |
| **No.** | **Objective** | **Performance criteria** |
| **Version 0.0** | **Fundamental Setup** |  |
| 0.0.1 | Create project and set up using SDL. | Successfully setup SDL dependencies. Compile with no errors. |
| 0.0.2 | Divide initialization steps into separate procedures and draw window. | Window is drawn and can be closed. |
| 0.0.3 | Get User Keyboard Input and log it to console | User can press keys and corresponding debug message will be outputted to console. |
| **Version 0.1** | **Orbit Object Implementation** |  |
| 0.1.1.0 | Set up Central Object Class (Accessors and Placeholder Methods) | Debugging procedures and Accessor methods within the class to access and display class attributes. |
| 0.1.1.1 | Set up Orbiting Body Class (Accessors and Placeholder Methods) | Debugging procedures and Accessor methods within the class to access and display class attributes. Must follow good OOP practices. |
| 0.1.1.2 | Set up child class for satellites (Accessors and Placeholder Methods) | Debugging procedures and Accessor methods within the class to access and display class attributes. |
| 0.1.2.0 | Implement Update Method for Orbit Body Class and Satellite Class | Mostly comments for what procedures need to occur each simulation step. Connect internal methods that should be called every frame, so that the entire update can be handled via one public method. |
| 0.1.3.0 | Create Simulation Class with attributes and accessors | Should contain necessary attributes, including a queue of orbiting bodies, a central body and various parameters. Implement all necessary methods / placeholders and include comments. |
| 0.1.3.1 | Outline Simulation Class Update Method | Method in place to handle entire application update. |
| **Version 0.2** | **Orbital Simulation Mathematics** |  |
| 0.2.1.0 | Implement Runge-Kutta 4 step in a procedure. | 2nd ODE solver set up so that orbits can be simulated in realtime (1 second = 1 second). Using mathematics covered above. |
| 0.2.2.0 | Integrate new RK4 method into the main program update procedure. | Connect RK4 method with Body update method and implement means to update all simulation bodies via simulation mainloop. |
| 0.2.3.0 | Create Abstract Data Type: Vector3 | Data type with accessor methods for x, y, z. |
| 0.2.3.1 | Create Magnitude Function in Vector Class | Function returns magnitude of the Vector. |
| 0.2.3.2 | Create Normalize Function in Vector Class | Function returns the normalized vector. |
| 0.2.3.3 | Overload operators for Vector3 struct | Implement vector addition, scalar-vector multiplication, dot product. |
| **Version 0.3** | **Simulation Step** |  |
| 0.3.1.0 | Configure simulation class such that only one orbiting body is in the simulation queue. To sense-check results, make the parameters of the central body that of the sun and the parameters of the orbiting body that of the earth. | Earth instantiated at the correct distance from the sun, with correct velocity. Calculate orbit period (projection) and sense-check results: result should have error under 2%. |
| 0.3.2.0 | In Orbital Body Update Method, pass parameters to RK4 step and start to calculate position as a function of time. (Note this will 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. | Pixel drawn at expected location. |
| 0.4.2.0 | Create method to draw a line of pixels between two points. | Draw a 3D shape on screen. |
| 0.4.3.0 | Create method to rotate a collection of 3D points about a centroid. | Rotating 3D shape visible on screen. Constant 3D rotation over time (via update method) with no scaling / distortion. |
| **Version 0.5** | **Satellite Child Class** |  |
| 0.5.1.0 | Instantiate one satellite in the simulation class main method. Set it’s “parent” to be the earth and set its parameters to be similar to that of the moon.  (*Note: We are presently forming a reductive version of our solar system, with only the sun, earth and moon, so that we can sense-check the results of our simulation.)* | Debug Methods to confirm attributes are set correctly. |
| **Version 0.6** | **Graphics II** |  |
| 0.6.1.0 | Draw the central body to the screen using custom graphics implementation. | Have a 3D shape drawn to the screen at the position corresponding to the origin. |
| 0.6.2.0 | Draw Orbiting bodies to the screen using the custom graphics implementation. Have the draw function in the update method for the orbiting body class. | Have a 3D shape drawn to the screen corresponding to the current position of the orbiting body. Drawn every frame. |
| 0.6.3.0 | Draw Satellites. | Have a 3D shape drawn to the screen corresponding to the current position of the satellite. Drawn every frame. |
| **Version 0.7** | **User Interface I** | **Draw User Interface using a library for GUIs in SDL.** |
| 0.7.1.0 | Draw sidebar with placeholder text corresponding to parameters for the simulation. | Text is drawn to the sidebar, occupying a portion of the side of the screen. |
| 0.7.2.0 | Add Placeholder buttons | Clicking button outputs a debug message to console. |
| 0.7.2.1 | Add Button to instantiate a new orbit body | Clicking button instantiates a new orbit with default parameters. |
| 0.7.2.2 | Add Button to pause simulation | Clicking button pauses simulation. |
| 0.7.3.0 | Add a panel that functions as an “inspector.”  *This will describe attributes of the object currently in focus.* | Placeholder panel. |
| **Version 0.8** | **User Interface II** |  |
| 0.8.1.0 | Connect simulation to the User Interface. UI should access and display orbiting body class attributes. | See object attributes shown for 1 orbiting body. |
| 0.8.2.0 | Allow user to interact with any orbiting body such that its information is shown on the extended UI. | Inspector window shows selected object’s properties |
| 0.8.3.0 | Add input fields to general simulation parameters. | Able to change simulation parameters via input fields. |
| 0.8.3.1 | Add input fields to inspector panel for orbit bodies. | Able to change orbit parameters via input fields. |
| 0.8.3.2 | Add Button within inspector to instantiate satellite orbit around current body. | Clicking button instantiates a new satellite around current body. |
| 0.8.3.3 | Add Button within inspector to reset selected orbit. | Clicking button resets orbit to initial parameters. |
| 0.8.3.4 | Add Button within inspector to delete selected orbit. | Clicking button removes body from orbit queue. Effectively deletes the orbit. |
| 0.8.4.0 | Add method to display orbit object vector attributes, such as Velocity and Force. (Including option to hide these.) | Arrows are draw to the screen representing the direction and magnitude of the vectors. |
| 0.8.5.0 | Add method to interact with orbit body such that the camera locks to it and follows its position around the simulation. | Camera’s centre |
| **Version 0.9** | **Data Storage** |  |
| 0.9.1.0 | Write simulation data to storage. | Show representation of simulation data in storage system in console. |
| 0.9.1.1 | Read simulation data from storage. | Show read data from storage solution in console. |
| 0.9.2.0 | Add ability to create new simulations & read / write to them. | Show new simulation in storage. |
| 0.9.2.1 | Add input method to select simulations to read from / save to. | Use input method to save to and read from a simulation. |

# Documented design

## Database design

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

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

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

### Entity Relationship diagram (ERD)



### Entity Attribute Model (EAM)



## Overall System Design

### UML Diagram



### Application Process Diagram



Figure : Process diagram detailing application flow

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

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

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

### Input, Process, Storage, Output (IPSO)

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

### Data dictionaries

|  |  |  |  |
| --- | --- | --- | --- |
| **Data Item** | **Data Type** | **Validation** | **Sample Data** |
| Simulation ID | Int |  | 1233052 |
| Orbit Body ID | Int |  | 4408723 |
| Central Body ID | Int |  | 5308921 |
| Orbit Body Position | Vector3 |  | (12000, 43000, 55000) |
| Orbit Body Velocity | Vector3 |  | (1, 1, 1) |
| Initial Orbit Position | Vector3 | X, y, z all floating point values and magnitude of position vector > 0. | (1000, 0, 0) |
| Orbit Body Mass | Double | Mass > 0 & Is Float | 120000 |
| Central Body Mass | Double | Mass > 0 & Is Float | 26000000 |
| Orbit Body Scale | Double | Scale > 0 & Is Float | 1 |
| Central Body Scale | Double | Scale > 0 & Is Float | 1 |
| Universal Time Scale | Double | 100 > UTS > -100 & Is Float | 10 |
| Orbit Body Name | String | Is String | “Kevin” |
| Central Body Name | String | Is String | “Not Kevin” |
| Date Created | String |  | “12/11/2022” |
| Orbit Body Vertices | Vector3 Array |  | [(1, 1, 0), (1, 2, 0), (2, 2, 0)] |
| Orbit Body Edges | Vector Array |  | [(0, 1), (1, 2)] |

### Data structures

#### Vector 3

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

**STRUCTURE Vector3**

**Float x**

**Float y**

**Float z**

**END STRUCTURE**

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

#### Edge

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

**STRUCTURE Edge**

**Integer a**

**Integer b**

**END STRUCTURE**

#### OrbitBodyData

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

**STRUCTURE OrbitBodyData**

**String name**

**Vector3 center**

**Float scale**

**Vector3 velocity**

**Float mass**

**END STRUCTURE**

### OOP class design



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

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

### 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 : 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.*

**

#### 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.*

**



#### 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.



#### 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.



#### Hashing Algorithm

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



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

### Libraries

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

# Technical solution

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

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



Figure : 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:  DATACONTROLLER  - orbiting\_bodies: VECTOR<BODY>  - sun: CENTRALBODY  - path\_source: STRING | - init()  - commit\_to\_text\_field()  - close\_planet\_inspectors()  - click()  - Update\_Clock()  - clean\_orbit\_queue()  - add\_specific\_orbit()  - 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]](#footnote-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]](#footnote-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  - 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> | + Init()  + CreateText()  + CreateIcon()  + GetTextParams()  + AddTextToRenderQueue()  + AddIconToRenderQueue()  + Get\_Screen\_Dimensions()  + Get\_Number\_Of\_Points()  + pixel()  + line()  + draw()  + free() |
| 7: Camera | - camera\_rotation : VECTOR3  + position : VECTOR3  + clipping\_z : FLOAT | + Camera()  + RotateCamera()  + rotate()  + WorldSpaceToScreenSpace() |
| 8: GTexture[[6]](#footnote-6) | - MTexture : SDL\_TEXTURE  - renderer : SDL\_RENDERER  - font : TTF\_FONT  - mWidth : INTEGER  - mHeight : INTEGER | + GTexture()  + GTexture(GTexture source)[[7]](#footnote-7)  + loadFromFile()  + loadFromRenderedText()  + reset\_texture()  + free()  + render()  + getWidth()  + getHeight() |
| 9: Button | - position : VECTOR3  - width : INTEGER  - height : INTEGER  - left\_wall\_offset : INTEGER  - function : FUNCTION<VOID()>  # enabled : BOOLEAN | # CallFunction()  # AttachFunction()  + Button()  + SetDimensions()  + SetPosition()  + SetEnabled()  + Clicked() |
| 10: FunctionButton  (INHERITS FROM BUTTON) | - icon : ICON | + FunctionButton()  + CheckForClick()  + SetEnabled()  + free() |
| 11: Icon | - texture : GTEXTURE  + pos\_x : INTEGER  + pos\_y : INTEGER  + path\_to\_image : STRING  + visible : BOOLEAN  + dimensions : VECTOR<INTEGER> | + Icon()  + Render()  + SetPosition()  + SetDimensions()  + GetDimensions()  + free() |
| 12: Text | # texture : GTEXTURE  + pos\_x : INTEGER  + pos\_y : INTEGER  + text : STRING  + visible : BOOLEAN | + Text()  + Text(Text t)  + Set\_Text()  + Set\_Position()  + Set\_Visibility()  + 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 performance[[8]](#footnote-8) across all hardware is ensured.

****

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



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

Text

Description automatically generated

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

Text

Description automatically generated with medium confidence

Figure : Clock in Simulation GUI

### Orbit Queue

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

Text

Description automatically generated

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



### Orbit Body Class

A picture containing shape

Description automatically generated

Figure : 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.

Text

Description automatically generated

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

#### The Update Method

Text

Description automatically generated

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



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

**A picture containing text

Description automatically generated**

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.

Text

Description automatically generated

#### Orbit Body Geometry

A screenshot of a computer

Description automatically generated with medium confidence

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.

Shape, circle, polygon

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

Figure : Orbit Body Geometry

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

**Text

Description automatically generated**

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

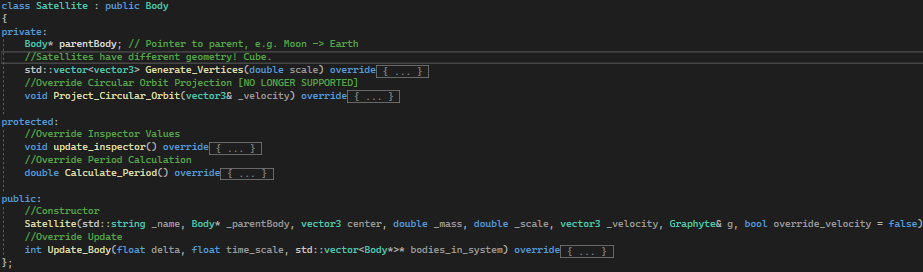
Following object-oriented programming conventions, numerous accessor methods have been implemented in this class and others in the project so that private and protected variables can be read.

Text

Description automatically generated

### Satellites

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



They are instantiated by a “parent” **Body** via a user interaction with a **FunctionButton** accessible in the Body’s inspector.



Text

Description automatically generated

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

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

#### Polymorphism

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

The most significant visual difference between the 1st generation[[10]](#footnote-10) orbits and 2nd generation orbits are the geometries generated in the virtual **Generate\_Vertices** method. Text

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

Figure : 1st (top) and 2nd (bottom) gen orbit geometries (NOT TO SCALE)

Engineering drawing

Description automatically generated with medium confidence The other significant override within the child class **Satellite** is the **Calculate\_Period** method:

Text

Description automatically generated

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.

Text

Description automatically generated

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

Text

Description automatically generated

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

Text

Description automatically generated

## 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**]

Text

Description automatically generated

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.

Text

Description automatically generated

#### Text

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

#### Arrow

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

Text

Description automatically generated

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.

#### 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.

Text

Description automatically generated

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

#### Function Buttons

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

Text

Description automatically generated

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).

Text

Description automatically generated

#### Double Field Value

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

Text

Description automatically generated

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

Text

Description automatically generated

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.

Text

Description automatically generated

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.



#### 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.

Text

Description automatically generated

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.



Below shows the validation method.



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.

Text

Description automatically generated

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.

Text

Description automatically generated

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:

Text

Description automatically generated

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

## Simulation Storage Solution

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

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

### OrbitBodyData

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



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.

Text

Description automatically generated

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

Text

Description automatically generated

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.

Text

Description automatically generated

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:

Text

Description automatically generated

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

Text

Description automatically generated

### SimulationData

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

Text

Description automatically generated

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.

Graphical user interface, text

Description automatically generated with medium confidence

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 Filesection.

#### Usage

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

Text

Description automatically generated

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

Text

Description automatically generated

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

Text

Description automatically generated

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:
   1. Bytes For Name (64 bits : double)
   2. Name:
      1. Character (8 bits : char)
   3. Position (64 \* 3 bits : vector3)
   4. Mass (64 bits : double)
   5. Scale (64 bits : double)
   6. 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.

Text

Description automatically generated

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.

Text

Description automatically generated

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

#### Example : Saving Earth

Graphical user interface, text, application

Description automatically generated

Figure : Example .orbyte file with file size

Graphical user interface

Description automatically generated

Figure : 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.

A picture containing text

Description automatically generated

Figure : 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

A screenshot of a computer

Description automatically generated

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

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

# Testing

## Testing strategy

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

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

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

## Testing plan

### Key

*“STND”: Standard (correct) data*

*“INCR”: Incorrect data*

*“BNDR”: Boundary data[[13]](#footnote-13)*

*“AE”: As Expected*

*“ERR”: Error / Incorrect result*

*“PR”: Considered Pre-requisite.*

### Plan

|  |  |
| --- | --- |
| Test Number | 1 |
| Objective / Version Number: | 0.0.1 - 0.0.3 |
| Description: | Testing to ensure that key presses are registered by Orbyte. (Anything preceding 0.0.3 considered prerequisite for this test). |
| Test Data: | I : Left Arrow Key [STND]  II : Right Arrow Key [STND]  III : Space Bar [STND]  IV : “G” Key [INCR] |
| Expected Result: | I : Debug Message  II : Debug Message  III : Debug Message  IV : No Message |
| Actual Result: | I : AE  II : AE  III : AE  IV : AE |
| Action needed? | None |

|  |  |
| --- | --- |
| Test Number | 2 |
| Objective / Version Number: | 0.6.2.0[[14]](#footnote-14) [PR: 0.4.2.0, 0.4.1.0, 0.3, 0.2, 0.1.3.1, 0.1.3.0, 0.1.2.0, 0.1.1.1, 0.1.1.0] |
| Description: | Instantiate Orbit Body in simulation with default parameters (through code). This is to test a combination of modules and the **Body** class. |
| Test Data: | Body earth = Body("Earth", {0, 1.49E11, 0}, 5.97E24, 6.37E6, { 30000, 0, 0 }, Sun.mu, graphyte, false);  //Name: "Earth"  //Radius of orbit: 1.49E11m  //Mass: 5.97E24kg  //Scale: 6.37E6m  //T\_Velocity: 30000ms^-1  //Mu: Sun's mu (See Analysis) |
| Expected Result: | Orbit body instantiated with name: “Earth” Displayed vertically above the sun. The period of the orbit should be close to 365 days. |
| Actual Result: | *See corresponding supporting evidence in subsequent section.*  Orbit Body instantiated in correct place, confirming that Body constructor works as expected. [AE]  Graphyte correctly draws the 3D geometry. [AE]  Period of orbit predicted within acceptable range. [ERR : I *| Technically within acceptable range, however nonetheless default parameters will be changed for more accuracy.*] |
| Action needed? | ERR : I 🡪 Change default orbit constructor parameters for earth to have more accurate values. |

|  |  |
| --- | --- |
| Test Number | 3 |
| Objective / Version Number: | 0.6.3.0 [PR: 0.5, 0.4, 0.3, 0.2, 0.1, 0.0.2, 0.0.1] |
| Description: | Second generation orbit body instantiated with name “Moon”. Checking to see that the object is instantiated near the parent (“Earth”), the period of orbit is near 27 days, and that the geometry is unique. |
| Test Data: | Name: “Moon”  Radius: 3.844E8m  Mass: 7.3E22kg  Scale: 1.7E5m  Velocity: 1024m/s |
| Expected Result: | 3D object (Cube) instantiated in orbit near the Earth. Should have a period of near 27 days. |
| Actual Result: | *See corresponding supporting evidence in subsequent section.*  Moon instantiated correctly. [AE]  Graphyte correctly renders the mesh wireframe for satellites (Cube). [AE]  Predicted period equal to accepted values for period of moon’s orbit around earth: 27.3 days. [AE] |
| Action needed? | None. |

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

|  |  |
| --- | --- |
| Test Number | 5 |
| Objective / Version Number: | 0.7.2.2 [PR: 0.3, 0.2] |
| Description: | Test the simulation “pause” button. Should set the timescale to 0 so that all bodies do not orbit, clock does not advance but UI etc. is still usable. |
| Test Data: | Button press. |
| Expected Result: | Simulation time-scale is set to 0. All values within body inspectors should not change and simulation GUI should still be operational. |
| Actual Result: | *See corresponding supporting evidence in subsequent section.*  Simulation time-scale set to 0 [AE], however text field showing current time scale not updated. [ERR: I]  Simulation clock does not progress, due to time scale value being successfully set to 0. [AE]  GUI still functional [AE]. |
| Action needed? | ERR : 1 🡪 Change “toggle\_pause” method within Simulation so that it updates the text field. |

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

|  |  |
| --- | --- |
| Test Number | 7 |
| Objective / Version Number: | 0.8.3.0 [PR: 0.7, 0.6, 0.3, 0.1] |
| Description: | Test “Time Scale” Input field within the Simulation Parameters section of the GUI.  Test multiple different values including negative real numbers, and check console & simulation for confirmation.  This tests the simulation’s ability to cope under different time scales and the DoubleFieldValue validation method. |
| Test Data: | I : 1 [STND]  II : -1 [STND]  III : 0.1 [STND]  IV : 86400 [STND]  V : -86400 [STND]  VI : 0 [BNDR]  VII : 8.64E7 [BNDR]  VIII : -8.64E7 [BNDR]  IX : “4EB9” [INCR]  X : “8&9” [INCR] |
| Expected Result: | I : Simulation time scale becomes 1 second per second  II : -1 second per second (reverse)  III : 0.1 seconds per second (slow)  IV : 1 day per second (fast)  V : -1 day per second (fast reverse)  VI : Paused  VII : Really fast simulation progression.  VIII : Really fast simulation regression.  IX : Rejection & Value NOT updated  X : Rejection & Value NOT updated |
| Actual Result: | *See corresponding supporting evidence in subsequent section.*  I : AE  II : AE  III : AE  IV : AE  V : AE  VI : AE  VII : AE  VIII : AE  IX : AE  X : [ERR : I] “8&9” incorrectly accepted as a valid input. |
| Action needed? | ERR : I 🡪 Amend DoubleFieldValue Validate method. Correct regular expression filtering inputs. |

|  |  |
| --- | --- |
| Test Number | 8 |
| Objective / Version Number: | 0.8.3.0 [PR: 0.7, 0.6, 0.3, 0.1] |
| Description: | Test “Centre Body Mass” Input field under Simulation Parameters section of GUI.  This tests the input field as well as the ability of bodies in orbit to adjust their physics simulation to the new value for “mu”, due to the updated mass of the body they orbit. |
| Test Data: | I : “1.989E30” [STND]  II : “5.972E24” [STND]  III : “-1.989E30” [STND]  IV : 0 [BNDR]  V : “3E32” [BNDR]  VI : “1.0.0.0” [INCR]  VII : “1$$./” [INCR] |
| Expected Result: | I : Mass set to value & orbits change.  II : Mass set to value & orbits change.  III : Mass set to value & orbits change.  IV : Mass set to 0 and orbits no longer affected by centre body.  V : All orbits dragged towards centre.  VI : Rejection & Value not updated.  VII : Rejection & Value not updated. |
| Actual Result: | *See corresponding supporting evidence in subsequent section.*  I : AE  II : AE  III : AE (albeit interesting to watch)  IV : AE  V : AE  VI : AE  VII : AE |
| Action needed? | None. |

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

|  |  |
| --- | --- |
| Test Number | 10 |
| Objective / Version Number: | 0.8.3.1 [PR: 0.0 – 0.8.2.0] |
| Description: | Testing name input field within Orbit Inspector.  Tests StringFieldValue & Graphyte system. |
| Test Data: | I : “Earth” [STND]  II : “16” [STND]  III : “” [INCR] |
| Expected Result: | I : Name of orbit changed to “Earth”  II : Name of orbit changed to “16”  III : Rejected & Name of orbit not changed. |
| Actual Result: | *See corresponding supporting evidence in subsequent section.*  I : AE  II : Rejected [ERR : I]  III : Accepted [ERR : II] |
| Action needed? | ERR : I 🡪 Amend StringFieldValue validation regular expression to include digits.  ERR : II 🡪 Amend StringFieldValue validation regular expression to not allow empty strings. |

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

|  |  |
| --- | --- |
| Test Number | 12 |
| Objective / Version Number: | 0.8.3.1 [PR: 0.0 – 0.8.2.0] |
| Description: | Testing mass input field within Orbit Inspector. Dynamically changing mass of body in orbit.  *To see the effects of changing the mass of a 1st Generation orbit (The earth). A 2nd Generation orbit (The moon) will be instantiated. This has the added benefit of verifying that the polymorphism within the child class* ***Satellite*** *works correctly.* |
| Test Data: | I : “5.972E26” [STND]  II : 0 [BNDR]  III : “Number” [INCR] |
| Expected Result: | I : Mass of Earth changed. 2nd Gen. Orbit to experience greater acceleration towards Earth.  II : Mass of Earth changed. 2nd Gen. Orbit to experience no force due to Earth, continue with tangential velocity at that time.  III : Rejection & Mass / 2nd Gen. Orbit unchanged. |
| Actual Result: | *See corresponding supporting evidence in subsequent section.*  I : AE  II : AE  III : AE |
| Action needed? | None. (Though acceleration arrows are going to be increased in size so that they are easier to see). |

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

|  |  |
| --- | --- |
| Test Number | 14 |
| Objective / Version Number: | 0.8.3.1 [PR: 0.0 – 0.8.2.0] |
| Description: | Testing velocity input field within Orbit Inspector. Changing velocity of body in orbit. |
| Test Data: | I : (0, 0, 30000) [STND]  II : (0, 0, 0) [BNDR]  III : (“A”, “B”, “C”) [INCR] |
| Expected Result: | I : Body velocity changed to only be in the position Z axis.  II : Body no longer has angular velocity; will move towards centre body.  III : Rejection & Velocity unchanged. |
| Actual Result: | *See corresponding supporting evidence in subsequent section.*  I : AE  II : AE  III : AE |
| Action needed? | None. |

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

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

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

|  |  |
| --- | --- |
| Test Number | 18 |
| Objective / Version Number: | 0.8.5.0 [PR: 0.0 – 0.8.2.0] |
| Description: | Right-Click on orbit to snap camera to that body’s orbit. |
| Test Data: | Right-Click on orbit body tag. |
| Expected Result: | Camera should follow the orbit body’s x and y position, maintaining a fixed z distance. |
| Actual Result: | *See corresponding supporting evidence in subsequent section.*  AE. Camera follows orbit of selected body. |
| Action needed? | None. |

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

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

## Test evidence

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

### Evidence Supporting Test 1

Console Output:

Shape, rectangle

Description automatically generated with medium confidence

### Evidence Supporting Test 2

Graphical user interface, text, application

Description automatically generated

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

Shape

Description automatically generated

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

Graphical user interface, text, application, chat or text message

Description automatically generated

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

### Evidence Supporting Test 3

Text

Description automatically generated

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

Text

Description automatically generated

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

Text

Description automatically generated

### Evidence Supporting Test 4

Console output upon button press:



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

### Evidence Supporting Test 5

Text

Description automatically generated with low confidence

Figure : Paused simulation

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

Console output:

A screenshot of a computer

Description automatically generated with low confidence

### Evidence Supporting Test 6

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

### Evidence Supporting Test 7

Console output showing tests: VIII, IX & X.

Text

Description automatically generated

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

### Evidence Supporting Test 8

Console output showing a rejection of the final test.

Text

Description automatically generated

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

### Evidence Supporting Test 9

Console output showing rejection of incorrect test value:

Text

Description automatically generated

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

### Evidence Supporting Test 10

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

### Evidence Supporting Test 11

Console output showing rejection of incorrect data test:

Text

Description automatically generated

*Note: bad input in this case was “”, hence nothing shown.*

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

### Evidence Supporting Test 12

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

### Evidence Supporting Test 13

Console output showing rejection of incorrect data test:

Text

Description automatically generated

Video Evidence : <https://youtu.be/YYHx9QyPKIQ>

### Evidence Supporting Test 14

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

### Evidence Supporting Test 15

Console output showing result of button press & satellite instantiation:

Text

Description automatically generated

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

### Evidence Supporting Test 16

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

### Evidence Supporting Test 17

Console output showing result of button press:

Text

Description automatically generated

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

### Evidence Supporting Test 18

Video Evidence : <https://youtu.be/CttMRlxoA7k>

### Evidence Supporting Test 19

Console output showing result of saving simulation:

Text

Description automatically generated

.Orbyte file saved to:

Graphical user interface, application

Description automatically generated

Video Evidence : <https://youtu.be/WhX0AquuPHc>

### Evidence Supporting Test 20

Console output showing result of opening file:

Graphical user interface, text

Description automatically generated

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

## Qualitative testing

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

* The user interface would be clearer if “fewer significant figures (were) displayed”. They suggested rounding larger numbers and showing fields to fewer decimal places.
* They requested that each UI field “show the units”, such as metres or seconds.
* Mr Rice suggested that it would be beneficial to be able to “reset the clock” within the simulation for timing purposes.
* They were “impressed by its (the simulation’s) ability to simulate the orbit of the ISS”.
* Mr Rice affirmed that the simulation could be used to “verify relationships learnt in A Level and GCSE” physics.
* The user was confident that the simulation could be “used in the classroom for demonstrations, once all bugs[[15]](#footnote-15) are fixed”.
* The user was impressed by the “sophistication of the simulation”, particularly the simulation of 2nd generation orbits (satellites).

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

## Stress Testing

To test the performance of Orbyte, this section covers how the FPS and memory usage of the simulation change with the instantiation of more orbits. See Analysis Of Performance under the Evaluation section for discussion of results shown below.

### Specifications:

Processor: Intel Core i7 @ 2.20 GHz

Memory: 32GB

### Results:

#### 100 Orbits

A picture containing text

Description automatically generated

Figure : Orbyte simulating 100 orbits.

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

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

# Iterative Development Process

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

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

## Rotation issues

Text

Description automatically generated

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

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

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

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

## Instantiating Orbits

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

The orbiting\_bodies vector attribute within the Simulation class used to contain instances of Body objects. This was not the intended design as it led to a Body being instantiated once, before being instantiated again via a copy constructor when added to the back of the “orbit queue” or vector. This caused excessive use of memory and damaged the performance of the simulation, while also causing confusion when accessing orbits via the GUI due to the multiple instances of any one orbit.

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



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

## Performance Problem When Rendering Geometry

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

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

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

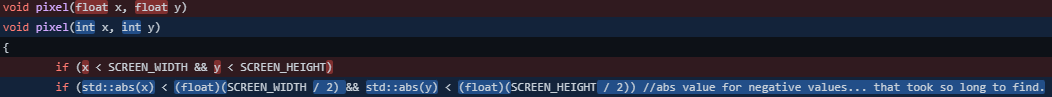


Figure : Git Diff showing new conditions that account for negative x and y values

# Evaluation

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

## Review against objectives

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

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

|  |  |  |  |
| --- | --- | --- | --- |
| **No.** | **Objective** | **Performance criteria** | **Evaluation** |
| **Version 0.0** | **Fundamental Setup** |  | Version implemented successfully. |
| 0.0.1 | Create project and set up using SDL. | Successfully setup SDL dependencies. Compile with no errors. | Orbyte uses SDL and can be built with no errors. **COMPLETED** |
| 0.0.2 | Divide initialization steps into separate procedures and draw window. | Window is drawn and can be closed. | Initialization of SDL and Graphyte is handled by an individual method that is called by the main application method. Window is drawn and functions as expected. **COMPLETED** |
| 0.0.3 | Get User Keyboard Input and log it to console | User can press keys and corresponding debug message will be outputted to console. | Switch statement within application mainloop handles user input and registers expected key-presses successfully. **COMPLETED** |
| **Version 0.1** | **Orbit Object Implementation** |  | Version implemented successfully. |
| 0.1.1.0 | Set up Central Object Class (Accessors and Placeholder Methods) | Debugging procedures and Accessor methods within the class to access and display class attributes. | Simulation has a central body, functioning as the “largest” or most massive body in a system. This object is the fundamental source of gravitational attraction within the simulation. **COMPLETED** |
| 0.1.1.1 | Set up Orbiting Body Class (Accessors and Placeholder Methods) | Debugging procedures and Accessor methods within the class to access and display class attributes. Must follow good OOP practices. | Orbit bodies form the basis of every simulated orbit. Handling their own update methods and any child “2nd generation orbits.” The class contains multiple key attributes and corresponding accessor / mutator methods, following good OOP convention. **COMPLETED** |
| 0.1.1.2 | Set up child class for satellites (Accessors and Placeholder Methods) | Debugging procedures and Accessor methods within the class to access and display class attributes. | **Satellite** inherits from **Body** and successfully demonstrates polymorphism through the overloading of inherited methods. **COMPLETED** |
| 0.1.2.0 | Implement Update Method for Orbit Body Class and Satellite Class | Mostly comments for what procedures need to occur each simulation step. Connect internal methods that should be called every frame, so that the entire update can be handled via one public method. | Orbit Body update method present and handles all internal processes that need to occur every frame. Calls RK4 step so that position can be calculated as a function of time, and updates body mesh for graphical representation.  Satellite inherits the Update method from Body and overrides it, making amendments to the code for satellite functionality. **COMPLETED** |
| 0.1.3.0 | Create Simulation Class with attributes and accessors | Should contain necessary attributes, including a queue of orbiting bodies, a central body and various parameters. Implement all necessary methods / placeholders and include comments. | Simulation class successfully set up. Only one is instantiated during the application’s execution. Attributes include an orbit body vector, a central body and necessary simulation parameters such as time scale & screen dimensions. **COMPLETED** |
| 0.1.3.1 | Outline Simulation Class Update Method | Method in place to handle entire application update. | Also referred to as the “application mainloop.” Handles all frame-to-frame processes of Orbyte, from calling the update method on Orbit Bodies, to calling Graphyte’s draw method. The simulation class encapsulates the entire functionality of Orbyte. **COMPLETED** |
| **Version 0.2** | **Orbital Simulation Mathematics** |  | Version implemented successfully. |
| 0.2.1.0 | Implement Runge-Kutta 4 step in a procedure. | 2nd ODE solver set up so that orbits can be simulated in realtime (1 second = 1 second). Using mathematics covered above. | Runge-Kutta 4 underpins the simulation. Without the second order ordinary differential equation solver calculating position as a function of time, the simulation’s mechanics and graphics would not be possible. It successfully simulates orbits in realtime, as well as larger, smaller or even negative time scales. **COMPLETED** |
| 0.2.2.0 | Integrate new RK4 method into the main program update procedure. | Connect RK4 method with Body update method and implement means to update all simulation bodies via simulation mainloop. | RK4 implementation used to calculate position as a function of time for bodies in orbit around the central body, and 2nd generation orbits around their parents and the central body.  **COMPLETED** |
| 0.2.3.0 | Create Abstract Data Type: Vector3 | Data type with accessor methods for x, y, z. | Vector3 struct implemented and used heavily throughout the simulation for calculations involving 3D vectors. **COMPLETED** |
| 0.2.3.1 | Create Magnitude Function in Vector Class | Function returns magnitude of the Vector. | Vector3 struct as parameter to the magnitude function. Used frequently throughout the simulation calculations and graphical representation of orbits. **COMPLETED** |
| 0.2.3.2 | Create Normalize Function in Vector Class | Function returns the normalized vector. | Vector3 struct as parameter to the normalize function. Returns a vector of magnitude 1 in the direction of the specified vector3. **COMPLETED** |
| 0.2.3.3 | Overload operators for Vector3 struct | Implement vector addition, scalar-vector multiplication, dot product. | Vector3 struct overloads many core c++ operators for usage in mathematical calculations. Extremely useful throughout the simulation. **COMPLETED** |
| **Version 0.3** | **Simulation Step** |  | Version implemented successfully. |
| 0.3.1.0 | Configure simulation class such that only one orbiting body is in the simulation queue. To sense-check results, make the parameters of the central body that of the sun and the parameters of the orbiting body that of the earth. | Earth instantiated at the correct distance from the sun, with correct velocity. Calculate orbit period (projection) and sense-check results: result should have error under 2%. | Earth can be instantiated such that it is the correct distance away from the sun (verified via debug console output and period calculations). Added to the list of orbits successfully. **COMPLETED**  *Simulation queue:* ***PARTIALLY COMPLETED?*** *Throughout the design stage and within the technical solution, the “orbiting\_bodies” vector of orbits has been referred to as a queue. It has been justified earlier, however a textbook “queue” such as a circular queue has not been used on account of it not being performant to repeatedly dequeue and requeue bodies that are known to remain in the simulation between updates. Nonetheless, the vector is used as first-in, first-out and so is referred to as a queue.* |
| 0.3.2.0 | In Orbital Body Update Method, pass parameters to RK4 step and start to calculate position as a function of time. (Note this will be modified time, depending on the time-step parameter of the simulation.) | Debug methods each update to confirm values are changing, verification of correct calculation will be done at a later stage. | Discussed above. The **Body** update method calls the protected rk4 step that calculates the body’s position as a function of time. It successfully does so and has been demonstrated in testing to do so accurately and performantly. **COMPLETED** |
| 0.3.3.0 | Write methods for the Orbital Body Class to output the current position. | Find the time period of the earth’s orbit in the simulation and compare to known values. Note the error (if any), and then refer to previous steps to fix bugs (if any). | Correctly calculates the period of orbits. Verified through testing and instantiation of known orbits such as those of the planets in our solar system, and comparing both projected orbit periods and simulated orbit period (the time in-simulation that a complete orbit takes) to the known values. Minimal error seen. **COMPLETED** |
| **Version 0.4** | **Graphics I** |  | Version implemented successfully. |
| 0.4.1.0 | Create method to draw a pixel to the screen given an x and y coordinate, using the SDL libraries. | Pixel drawn at expected location. | Graphyte is capable of drawing a pixel anywhere on the screen. **COMPLETED** |
| 0.4.2.0 | Create method to draw a line of pixels between two points. | Draw a 3D shape on screen. | Graphyte is capable of drawing a collection of pixels, generated from lines and points. As such it is able to draw 3D geometry such as the diamond used to represent orbit bodies, or a cube. More complex geometry is possible through a more detailed mesh. **COMPLETED** |
| 0.4.3.0 | Create method to rotate a collection of 3D points about a centroid. | Rotating 3D shape visible on screen. Constant 3D rotation over time (via update method) with no scaling / distortion. | Rotation matrix successfully applied to a point in 3D space so that shapes can be rotated every frame. **COMPLETED** |
| **Version 0.5** | **Satellite Child Class** |  | Version implemented successfully. |
| 0.5.1.0 | Instantiate one satellite in the simulation class main method. Set it’s “parent” to be the earth and set its parameters to be similar to that of the moon.  (*Note: We are presently forming a reductive version of our solar system, with only the sun, earth and moon, so that we can sense-check the results of our simulation.)* | Debug Methods to confirm attributes are set correctly. | The functionality of satellites exceeded expectations. Throughout their implementation the simulation struggled to accurately simulate their orbits and it was common for satellites to escape the orbit of their parents. At lower time-scales the simulation is more accurate, and therefore the orbit of satellites is more reliable. In testing, the moon was instantiated to orbit the earth and it was instantiated correctly, with an accurate orbit period. Ultimately the satellite class has been a success testament to the strength of the simulation. More can be done to improve their simulation. Perhaps calling the rk4 multiple times to interpolate between larger steps caused by higher time-scales will fix inaccuracies, however this will come with a very large performance overhead for larger time-scales. **COMPLETED** |
| **Version 0.6** | **Graphics II** |  | Version implemented successfully. |
| 0.6.1.0 | Draw the central body to the screen using custom graphics implementation. | Have a 3D shape drawn to the screen at the position corresponding to the origin. | Central body drawn to screen as a diamond shape, used also for 1st generation orbits. **COMPLETED** |
| 0.6.2.0 | Draw Orbiting bodies to the 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. | Orbiting bodies drawn to the simulation window as 3D diamonds. Bodies able to be moved over the course of their orbit, frame-to-frame moved to the position calculated by the ODE solver. **COMPLETED** |
| 0.6.3.0 | Draw Satellites. | Have a 3D shape drawn to the screen corresponding to the current position of the satellite. Drawn every frame. | Satellites (2nd generation orbits) represented by a cube that is successfully drawn to the screen. **COMPLETED** |
| **Version 0.7** | **User Interface I** | **Draw User Interface using a library for GUIs in SDL.** | Version implemented successfully. |
| 0.7.1.0 | Draw sidebar with placeholder text corresponding to parameters for the simulation. | Text is drawn to the sidebar, occupying a portion of the side of the screen. | Simulation sidebar forms a crucial part of the GUI. Seen throughout all tests in the Testing section, the simulation parameters, clock and performance metrics are always visible on the left hand side of the screen. **COMPLETED** |
| 0.7.2.0 | Add Placeholder buttons | Clicking button outputs a debug message to console. | Buttons ultimately implemented successfully. The placeholders and debug messages proved useful in finding problems when not registering button presses etc. **COMPLETED** |
| 0.7.2.1 | Add Button to instantiate a new orbit body | Clicking button instantiates a new orbit with default parameters. | New orbit body button seen in the top right of the simulation. Proved to work as expected during testing. Vital part of the user interface and it gives users the freedom to add as many orbits as they want to, contributing to Orbyte’s “sandbox” nature. **COMPLETED** |
| 0.7.2.2 | Add Button to pause simulation | Clicking button pauses simulation. | Used to stop the simulation from updating the position of orbit bodies and halt the clock’s progression, so that changes can be made to the configuration of the simulation before it is resumed. Provides good structure to making changes to the simulation:   1. User pauses simulation. 2. Changes parameters. 3. Resumes.   This avoids any problems with changing the radius of the orbit and not having time to change the velocity before the orbiting body moves. By pausing the simulation, multiple fields can be edited before the next step is calculated. This button is possibly the most frequently used in the simulation. **COMPLETED** |
| 0.7.3.0 | Add a panel that functions as an “inspector.”  *This will describe attributes of the object currently in focus.* | Placeholder panel. | The inspector is what sets Orbyte apart from a demonstration. It is no longer a display of orbits, as the inspector turns it into a tool. By displaying orbital characteristics such as velocity, radius, acceleration and mass, the simulation provides more possibility for configuration and potential complexity than any other seen in the analysis section. **COMPLETED** |
| **Version 0.8** | **User Interface II** |  | Version implemented successfully. |
| 0.8.1.0 | Connect simulation to the User Interface. UI should access and display orbiting body class attributes. | See object attributes shown for 1 orbiting body. | Inspector shows parameters for selected orbit body. **COMPLETED** |
| 0.8.2.0 | Allow user to interact with any orbiting body such that its information is shown on the extended UI. | Inspector window shows selected object’s properties | With multiple bodies in orbit, the user is able to click one of the bodies in orbit in order to open its inspector. This way different orbits can be inspected, whilst being simulated together. **COMPLETED** |
| 0.8.3.0 | Add input fields to general simulation parameters. | Able to change simulation parameters via input fields. | Crucially, in order to interact with the simulation, the user must be able to configure parameters via input fields. The user is able to change time scale and the mass or scale of the centre body. **COMPLETED** |
| 0.8.3.1 | Add input fields to inspector panel for orbit bodies. | Able to change orbit parameters via input fields. | The greatest level of customisability in an orbit simulation comes from being able to change the attributes of a body in orbit: its position in metres, its initial velocity, its size and its name. Orbyte facilitates all of this through input fields in its inspector, that can be opened by selecting a body in orbit. **COMPLETED** |
| 0.8.3.2 | Add Button within inspector to instantiate satellite orbit around current body. | Clicking button instantiates a new satellite around current body. | 2nd Generation orbits are instantiated via a 1st generation orbit body’s inspector. The “add satellite” button is identical to the main “add orbit” button, except under the inspector. This facilitates adding orbits such as the moon, around the earth, with the earth and moon both orbiting the sun. **COMPLETED** |
| 0.8.3.3 | Add Button within inspector to reset selected orbit. | Clicking button resets orbit to initial parameters. | Under the orbit body inspector, it is possible to reset the orbit to its last configured initial position and velocity. This is helpful in case any unexpected behaviour develops that needs to be undone. **COMPLETED** |
| 0.8.3.4 | Add Button within inspector to delete selected orbit. | Clicking button removes body from orbit queue. Effectively deletes the orbit. | The orbit body inspector contains a button used to delete the selected orbit. Done by removing the body from the orbit queue and not rendering it. This is helpful as it removes clutter from the simulation and frees up processing resources as fewer bodies have to be updated each update. **COMPLETED** |
| 0.8.4.0 | Add method to display orbit object vector attributes, such as Velocity and Force. (Including option to hide these.) | Arrows are draw to the screen representing the direction and magnitude of the vectors. | Arrows are incredibly useful, both for debugging the simulation calculations by illustrating the vectors calculated as a result of the rk4 step, but primarily utilised as a part of the UI to visualize what is affecting the journey of a body in orbit. The acceleration of an object is drawn as a double headed arrow and the velocity is shown as a single headed arrow. **COMPLETED** |
| 0.8.5.0 | Add method to interact with orbit body such that the camera locks to it and follows its position around the simulation. | Camera’s centre | Due to the “to scale” nature of the simulation’s graphical representation of orbits, it is helpful to be able to follow an object in orbit with it at the centre of the screen, so it can be zoomed in on. By right clicking an orbit body, it is snapped to the centre of the screen. This makes it easier to view the geometry of the orbit body and its satellites’ paths.  The only issue with the current system is that the follow method only facilitates looking at the simulation along the z axis. Rotating the simulation with the arrow keys leads to the camera not following the orbit body. This could be fixed given more time by only using the object’s screen space position and then converting that to a world space position for the camera.  **PARTIALLY** **COMPLETED** |
| **Version 0.9** | **Data Storage** |  | Version implemented successfully |
| 0.9.1.0 | Write simulation data to storage. | Show representation of simulation data in storage system in console. | Simulations can be saved and written to a .orbyte file so that they can be opened another time. **COMPLETED** |
| 0.9.1.1 | Read simulation data from storage. | Show read data from storage solution in console. | Simulations can be read from a .orbyte file so that previous configurations and simulation states can be accessed and reopened. Especially useful for “templates” that can be made by one user and used by another. **COMPLETED** |
| 0.9.2.0 | Add ability to create new simulations & read / write to them. | Show new simulation in storage. | Path input field specifies which file to write to / create, and save button writes to that path. **COMPLETED** |
| 0.9.2.1 | Add input method to select simulations to read from / save to. | Use input method to save to and read from a simulation. | Path input field specifies which file to read from and open button reads from the file. **COMPLETED** |

## Analysis of independent feedback

When showing the key user, Mr Rice (see Analysis), the current state of the simulation, he was impressed by the degree of sophistication, configurability and 3D nature of Orbyte. As discussed earlier in the Qualitative Testing section, the end user was confident that Orbyte could be used in a classroom to verify the relationships and principles covered at A Level and GCSE. When shown that all orbits affect each other and it is possible to instantiate satellites that orbit 1st Generation Bodies, the end user praised the intricacy of the simulation. Despite the version that was shown to them containing a few bugs at the time, they expressed confidence in the simulation and verified that simulated orbit behaviour was as they would have expected to see.

## Analysis of performance

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

## Potential improvements

### Optimisations

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

#### Drawing Pixels

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

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

# 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/#](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]

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. [↑](#footnote-ref-1)
2. This could ultimately be implemented as a structure. [↑](#footnote-ref-2)
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. [↑](#footnote-ref-3)
4. “mu” is defined as the gravitational constant multiplied by the focus of the orbit: G x Mlarge [↑](#footnote-ref-4)
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. [↑](#footnote-ref-5)
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. [↑](#footnote-ref-6)
7. Copy constructor used for debugging. Adds no functionality in the class but was crucial in fixing rendering bugs. [↑](#footnote-ref-7)
8. Performance and simulation optimisation is discussed further on. [↑](#footnote-ref-8)
9. Treated as a “queue” in so far as it operates as first in, first out. It is not necessary to access a specific element by index per sae, however by removing the need to re-enqueue objects every frame, significant performance has been preserved. [↑](#footnote-ref-9)
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. [↑](#footnote-ref-10)
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. [↑](#footnote-ref-11)
12. Path to .orbyte files for reading or writing are entered in the global path input [↑](#footnote-ref-12)
13. “Boundary Data” will be interpreted loosely from test to test. In some situation the conventional idea of giving data on the edge of a range is inapplicable. Technically for many input fields the boundary data would be the maximum representable double. The simulation was explicitly designed to allow any VALID data. [↑](#footnote-ref-13)
14. Though many version points have been skipped, they are all being tested. These tests are targeted at the current version of Orbyte in its entirety, not individual components in the past. The iterative development process and testing that occurred throughout is discussed later. [↑](#footnote-ref-14)
15. “Bugs” referring to some of the problems found in testing that were not fixed yet. All issues are now fixed, as shown in testing. [↑](#footnote-ref-15)