

Computing

# OpenGL N-Body Simulator

**A2 Coursework**

Byron Theobald

March 2016

Northgate Sixth Form

Typeset with L<sup>A</sup>T<sub>E</sub>X

# Contents

<b>1. Analysis</b>	<b>4</b>
1.1. Background . . . . .	4
1.1.1. Project Objective . . . . .	4
1.1.2. Prospective Users . . . . .	4
1.2. Requirements and Processes . . . . .	4
1.2.1. Graphics Processing . . . . .	5
1.2.2. Calculations and Trigonometry . . . . .	5
1.2.3. Performance and Multithreading . . . . .	6
1.2.4. Saving of Scenarios . . . . .	7
1.2.5. Scale . . . . .	8
1.2.6. Summary: Requirements and Processes . . . . .	8
1.3. Data Management . . . . .	9
1.3.1. Data Sources and Destinations . . . . .	9
1.3.2. Data Dictionary . . . . .	10
1.3.3. Data Flow Diagrams . . . . .	13
1.4. Objectives . . . . .	15
1.4.1. System . . . . .	15
1.4.2. Processing . . . . .	15
1.4.3. User . . . . .	16
1.5. Potential Solutions . . . . .	17
1.5.1. C++, OpenGL, GLFW . . . . .	17
1.5.2. Pascal, Graph, Forms . . . . .	19
1.5.3. C++, OpenGL, SFML . . . . .	21
1.5.4. Conclusion . . . . .	22
1.6. Management and Tools . . . . .	23
1.6.1. Linux and Development Environment . . . . .	23
1.6.2. Git - Version Control . . . . .	23
1.6.3. GDB - Debugging . . . . .	24
1.6.4. Valgrind - Profiling and Memory Analysis . . . . .	24
1.6.5. L <sup>A</sup> T <sub>E</sub> X- Document Authoring . . . . .	26
<b>2. Prototyping</b>	<b>27</b>
2.1. Simulation Efficiency . . . . .	27
2.2. Simulation Prototyping . . . . .	28
2.2.1. Leapfrog Intergration . . . . .	29
<b>3. Design</b>	<b>31</b>
<b>4. Implementation</b>	<b>32</b>
<b>5. Manual</b>	<b>33</b>

<b>6. Evaluation</b>	<b>34</b>
<b>7. Code Listing</b>	<b>35</b>
7.1. Early Prototypes . . . . .	35
7.1.1. Euler (Trig) . . . . .	35
7.1.2. Euler (Vector) with $\Delta E$ . . . . .	37
7.1.3. Leapfrog (Vector) with $\Delta E$ . . . . .	38
<b>APPENDICES</b>	<b>39</b>
<b>A. Dialogue</b>	<b>39</b>

# 1. Analysis

## 1.1. Background

Mr Snowden is a Physics Teacher at Northgate High School, teaching the subject at levels from KS3 to A-Level, there are also several other teachers in the department. During lessons, there are some times where it can be useful to the students to be able to see a model of the subject that they are learning about, while the majority of subjects that are taught have a range of models available, both real and software models. However one of the subjects where models are quite limited is in orbital mechanics and circular motion, with most being just simple animations and not a fully interactive or real-time simulation. The main disadvantage of this is that the students are not able to easily see how different modifications to a system will affect the outcome, something that can help to improve the understanding of a particular topic in some scenarios.

### 1.1.1. Project Objective

Mr Snowden requires a teaching tool that can provide a graphical sandbox to create and simulate various scenarios to fortify and replace the currently used tool-set of simple animations and power-point presentations that are used in teaching. This would allow the teacher in a physics lesson to quickly show students how a situation would develop, making use of SI units in order to effectively show students how different parameters (Mass, Size and Velocity) would affect a situation.

### 1.1.2. Prospective Users

While Mr Snowden is intended to be the primary user, other teachers could also have access to the tool as it could be put on the shared network drive, making the tool easy to access and used as a teaching aid in lessons, the tool will also be accessed by students, allowing the tool to be used as a revision resource for individual students on the school computers or at home. Because of this special attention should be paid to make sure that the program is as compatible as possible

## 1.2. Requirements and Processes

The interface must be intuitive and self-explanatory in order to ensure that it is easy for teachers to understand, meaning that keyboard short-cuts should be commonly known and any buttons should be clearly labelled. It must also be able to run well on the

school computers, most of which are running on low end hardware (Dual-Core HT Intel i3 / Quad-Core Intel i5, 2-4GB RAM) using a 32-Bit Windows operating system, as this is what all computers in the school are running for internal compatibility reasons. This applies a limit of 4GB to the total system RAM but a maximum per program usage of 2GB, meaning that the program will need to handle memory efficiently and potentially have a limit on the number of bodies (In the range of 1000s) to avoid using too much memory and causing the system to crash, a sensible limit would be 500MB. The number of bodies that this would support can be estimated however actual memory usage will likely be higher due to graphics library overhead, meaning that memory profiling should be used. The program must not require installation on the system or any secondary redistributes to be installed containing dynamic libraries.

### **1.2.1. Graphics Processing**

While the resources of these systems are limited, keeping the program to being 2D should help to improve performance as there will be less demand on the integrated GPU (Graphics Processing Unit), meaning that it requires less system resources to display the simulation. This means that an external graphics library would need to be used in order to interface with this hardware and still function across different platforms. (The two main GPU Vendors, AMD and Nvidia, both have different interfaces for using GPUs for compute applications)

Having a 2D model also means that far less calculations need to be made for the simulation, making it faster, but it also reduces the load GPU in terms of rendering, As 3D would require shading and lighting to make depth perceivable. Input and camera movement would also be far more complex.

### **1.2.2. Calculations and Trigonometry**

The application must be able to simulate the paths of at least three bodies, as this will encounter the n-body problem ( $n = \text{Any positive integer}$ ), which in essence means that any interaction between two bodies can simply be predicted, however there is no direct solution for three or more bodies, and must be simulated through iterative process between each body to every other body. The main calculation that will be used between bodies is:

$$F = \frac{Gm_1m_2}{d^2}$$

This calculation produces a result for the gravitational force between objects, taking into account the mass of the two planets and the distance between them, based on a universal gravitational constant. ( $6.67408 \times 10^{-11} m^3 kg^{-1} s^{-2}$ ) While this equation only works between two bodies, the equation can be repeated in an iterative fashion in order to calculate the forces between all bodies in a system, the force can then be resolved to a single dimension for X and Y respectively, and then using a value for mass, acceleration can be calculated. By multiplying the acceleration by the change in time ( $\Delta t$ ) of the simulation you can propagate a velocity ( $v$ ) and change the position ( $p$ ) of a body.

$$F_y = F \sin \theta \quad F_x = F \cos \theta$$

The main issue that could crop up is the computational cost of the trigonometric functions, while the instruction set used by processors in nearly all modern desktop computers (x86) has specific instructions for these functions. The issue is that these still take longer than simple mathematical operators, which generally only take single instruction cycles. The difficulty is however that it will be heavily dependant both on the particular CPU running the code, but also the compiler used as well as the implementation used by the programming languages libraries.

There is also the issue that signs of calculated components to be flipped depending on which quadrant the simulated particle is in, adding further computational complexity. Trigonometric functions are also required to calculate the angle that the force is acting relative to the global X axis.

Because of these difficulties, it is worth investigating alternative methods for the calculation of the forces, as there could be far simpler methods that reduce the required complexity of code.

### 1.2.3. Performance and Multithreading

*Since the Mid-2000s, CPU Clock speeds have all but stopped increasing in the consumer market, with most coming in at the 2GHz to 3GHz range. Some high end desktop processors are able to be effectively over-clocked, with processors like the AMD FX-9590 capable of 5GHz as long as adequate cooling is provided, the current world record is 8.794GHz, achieved by an AMD FX-8350, however this with the use of Liquid Nitrogen to keep it cool.*

*While clock speeds are effectively limited by the stability that can be achieved in the material being used (Silicon) and thermal properties, the focus has now switched to increasing the core count of CPUs, as well as improving their power efficiency and increasing their transistor count by shrinking the size of the transistors on die. (Consumer Intel CPUs are currently at 14nm, with 10nm on road map for 2017 and 5nm by 2021)*

*Most Modern CPUs can be found with a core count of 2 or 4, with technologies such as hyper-threading allowing for multiple threads to be executed per core, effectively increasing the number of 'logical cores' by a factor of 2. High end desktop and workstation CPUs can be found with 6-12 physical cores, allowing for 12-24 consecutive threads. (Server CPUs are now coming with upwards of 18 physical cores, giving them 36 threads. These systems can also support multiple CPUs on the same motherboard, up to 8, allowing for effectively 288 consecutive threads on a single server.)*

Mutli-threading leads to is the potential to speed up the execution of the program by computing multiple parts of it at the same time. All of the computers in the school are new enough to have at a minimum dual core Intel i3 with hyper-threading, giving them a total of 4 'logical cores', this means that there is definitely some potential to improve the execution speed of the program.

This opens up new challenges in the form of ensuring that the programming is '*thread safe*', this means that the program is not going to *â€šstep on its own toesâ€š*, for example modifying data while another thread is reading it causing lock-ups, or race conditions where one thread completes a task before another and the program fails. When done properly the improvements to performance can be quite impressive.

#### **1.2.4. Saving of Scenarios**

The system will also need the ability to save the current scenario to a file, most likely a plain comma separated value file with a different file extension such as .sav, this will allow a teacher to set up a scenario inside the program outside of lesson time and save it for a future lesson, at which point it can be loaded into the program and used as a quick demo. These files could also be provided to students for use in their revision. Data for every body would need to be stored, namely Mass, Velocity and Position, constants could also be stored which would enable them to be changed in particular scenarios.

### 1.2.5. Scale

It is unlikely that true scale would translate particularly well to a program like this without visual aids added to the program, as distances are extremely massive, and planets are extremely small in comparison to the sun, let alone the distances involved, for example, Earth has a diameter of only 12700 km, however it orbits the sun at a height of 150 Million km. At a zoom where both the sun and earth will be visible, the earth will be too small to make out, even on extremely high resolution monitors.

### 1.2.6. Summary: Requirements and Processes

- Intuitive and clean interface making use of common keyboard short-cuts.
- The program must be compatible with as many computers as possible, as it must be able to be run by students outside of school as well as on the school computers. Care must be made to ensure that code is portable and can be easily recompiled for other operating systems, such as Linux or Mac OSX.
- It must run on 32-bit computers and operating systems.
- The program must be able to simulate n-bodies, with the only limit being acceptable performance and the amount of memory being used. The body limit will be based on the amount of memory being used. (Max 500MB)
- The user should be able to set up a scenario to save and then use at a later date.
- The user will have complete freedom to change to scale the scenario how they please and will be given complete control of constants such as the gravitational constant, as this would make smaller scale simulations more feasible.
- The program will make use of simple 2D graphics to keep system resource requirements low and boost performance. (3D is not in scope for this project)
- A graphics library should be used that allows efficient usage of a computer's GPU in order to accelerate the rendering of the simulation.
- Multi-threading could be investigated in order to improve the performance of simulation and handling of rendering.
- Some limitations will be required on variables such as position and velocity in order to ensure that a minimum precision is kept and that the laws of physics are not violated. (Speed of Light)



## 1.3. Data Management

### 1.3.1. Data Sources and Destinations

Table 1: Data Sources and Destinations

Data Name	Source	Destination
Mouse Coord / Key Press	Mouse / Keyboard	Input Handling
Saved Scenarios	File System	Render Scenario
Object Attributes	User Input / Scenarios	Render / Sim Scenario / File

Direct user input will be relatively low volume, taken in using keyboard short-cuts and mouse control, the user operates the program in order to place down objects, adjusting the initial size, mass and velocity of bodies in order to set-up the scenario, the user can also stop and start the program and change the speed of time within the simulation as well as other constants.

The loading and saving of scenarios will be sending and receiving similar data to what is provided by user inputs, however the data for an entire scenario will be transferred in one go, this includes the mass and size of bodies, as well as the most recent resolved (X, Y), Force, Velocity and Position of each simulated body, various constants that are set should also be stored, such as the gravitational constant, which can be changed by the user.

During the running of the simulation, the program will aim to keep the frame-rate (screen update) at 60 frames per second, (Ensures very smooth animation and responsiveness), this means that the data volumes during the running of the simulation are very high, as each frame could be a simulation update, this includes updating the resolved (X, Y) forces, velocity, acceleration and position of each body, according to how much simulation time passed per each frame.

### 1.3.2. Data Dictionary

The main flow of data around the program will be body data, as there will need to be a container of some data type that will contain a large number of these bodies. Because this is a simulation, I want to make use of the common double data type, this is the IEEE Standard, 64 bit variable providing anywhere from 15-17 digits of precision depending on the specific hardware and software implementation used.

While a float variable is smaller and only 32-Bits long, it is generally considered that there will be minimal computational benefit from using float instead of double on CPU bound simulations. In the majority of language mathematical functions take in double variables. While these can still take float variables, the operation will still performed as a double variable.

The double variable will also be useful for providing extra precision when it comes to the simulation variables, allowing the simulation to be potentially more accurate, however there may be situations where float can be beneficial to the memory footprint of the program.

It is also worth noting that if the simulation was being written to run on a GPU, the variables would need to be stored as floats unless a GPU with a large amount of double-precision units was being used as the target, as most GPUs will only have a small number of double-precision cores, often resulting in their performance being lower than what could be achieved using a CPU.

The variables written in the table below are likely to change as this project develops, particularly once the specific implementation is picked, things like program flow are more likely to develop during the programming stage.

Table 2: Data Sources and Destinations

Data Name	Data Type	Description
Mass of Body	Double	The mass of a particular body, every body will have mass. Defined as an objects reluctance to move, allows for calculation of forces, acceleration and potentially collisions.
Size of Body	Double	The graphical size of an object, could be used for calculation of collisions.
Force between Bodies	Double	Direct calculation for force between two bodies. calculated for every body to every other body. (Vector)
Angle of Force	Double	This will contain the angle of the force acting on the body relative to global Y axis.
Force X/Y	Double	Force between bodies resolved to global X/Y axis.
Acceleration X/Y	Double	Acceleration can be calculated using the mass of the body and the X/Y force.
Initial Velocity	Double	An initial velocity that is specified by the user when a body is placed. (Vector)
Angle of Velocity	Double	This contains the angle of the initial velocity of a body, relative to the positive global Y axis.
Velocity X/Y	Double	The initial velocity will be resolved to global X/Y axis, this value is also modified when the change in velocity is calculated using per frame time and Acceleration X/Y.
Position X/Y	Double	Position is first populated with the position that the user places the body at, it is then modified depending on the X/Y Velocity, updating its position, this is also used when resolving forces and velocity, it is also used to calculate the distance between two bodies to calculate the vector force.
Distance between Bodies	Double	The distance between the two bodies in current calculation, calculated using the positions of the two objects being simulated.
Body	Complex Structure ( $\approx 120B$ )	This will define a particular body, containing all the variables that relate to a particular body, stored in some kind of structure / object.
Gravitational Constant	Double	This defines the gravitational constant which is used to calculate the force that gravity will exert on two bodies, real value is $6.67408 \times 10^{-11} m^3 kg^{-1} s^{-2}$ However this value will not be particularly useful and should be something larger to allow for smaller scale simulations. The user will be able to modify this themselves.
Time Multiplier	Double	This value will be used when calculating the simulation time per displayed frame to allow the simulation to be run faster or slower than real time

Table 3: Data Sources and Destinations Cont.

Data Name	Data Type	Description
Paused	Boolean	This variable will be a simple TRUE/FALSE describing if the simulation is paused, time will not advance when this is true.
Scenario	Complex Structure	Size is dependant on the number of bodies present in the simulation, Also contains constant variables that affect simulation, namely time multiplier and the gravitational constant, as well as if the simulation is paused, it may be more efficient and safer to have separate scenarios for simulation and rendering, copying the simulation scenario over to render when a frame has finished simulation.
Pi $\pi$	Constant	Pi is the ratio of a circle's circumference to its diameter, it is defined as 3.14159265358979323846 as a constant in the C++ cmath library. While Pi is not entirely necessary for the calculations used in the simulation, it may be useful for writing other functions, particularly when converting between Radians and Degrees.
$F = \frac{Gm_1m_2}{d^2}$	Formula	This equation can be used for calculating the force between two bodies due to gravity, F is a vector Force, G is the Gravitational Constant, m is the mass of a body, d is the distance between the bodies.
$F_y = F \sin \theta$	Formula	This equation allows the vector Force or Velocity to be resolved to a single axis/dimension. This resolves the y axis. This will be the main computational cost of the program.
$F_x = F \cos \theta$	Formula	This equation allows the vector Force or Velocity to be resolved to a single axis/dimension. This resolves the x axis. This will be the main computational cost of the program
$F = ma$	Formula	This equation is one of the first principles of physics, Newton's Second Law, F is force, m is Mass, a is Acceleration.
$p = mv$	Formula	This equation describes how momentum is related to mass and velocity, this equation can be performed on both the x and y axis in the event of a collision between bodies to calculate its final trajectory, momentum is conserved through collisions.

### 1.3.3. Data Flow Diagrams

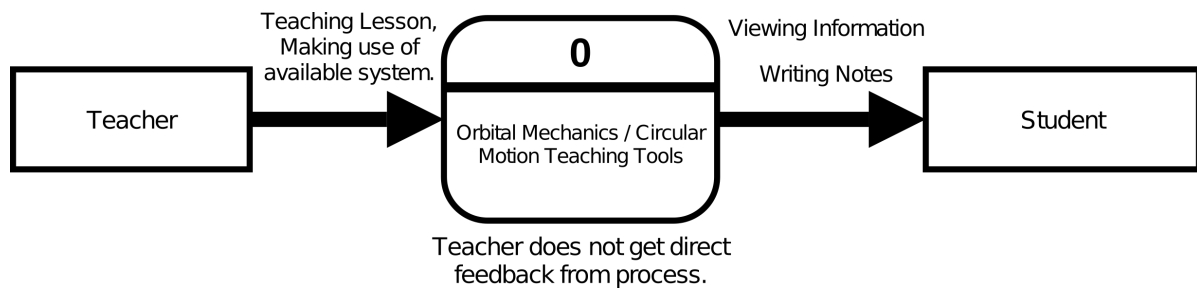


Figure 1: Current System, Level 0

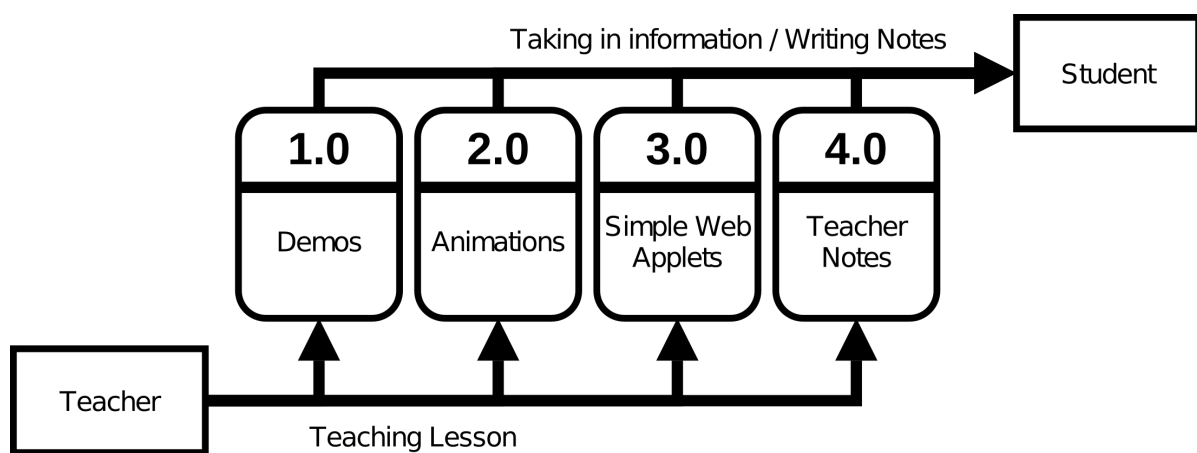


Figure 2: Current System, Level 1

The program can be used by either teachers or students, teachers can also provide preset scenarios for pupils to make use of in their study.

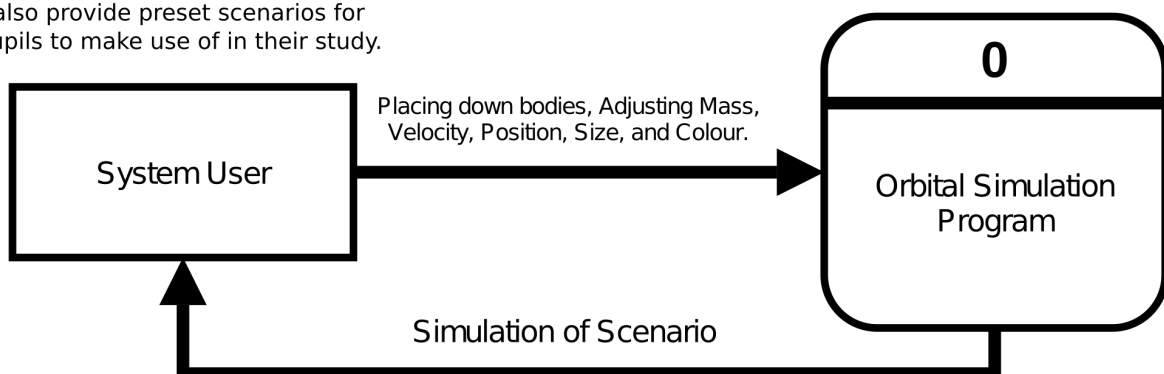


Figure 3: New System, Level 0

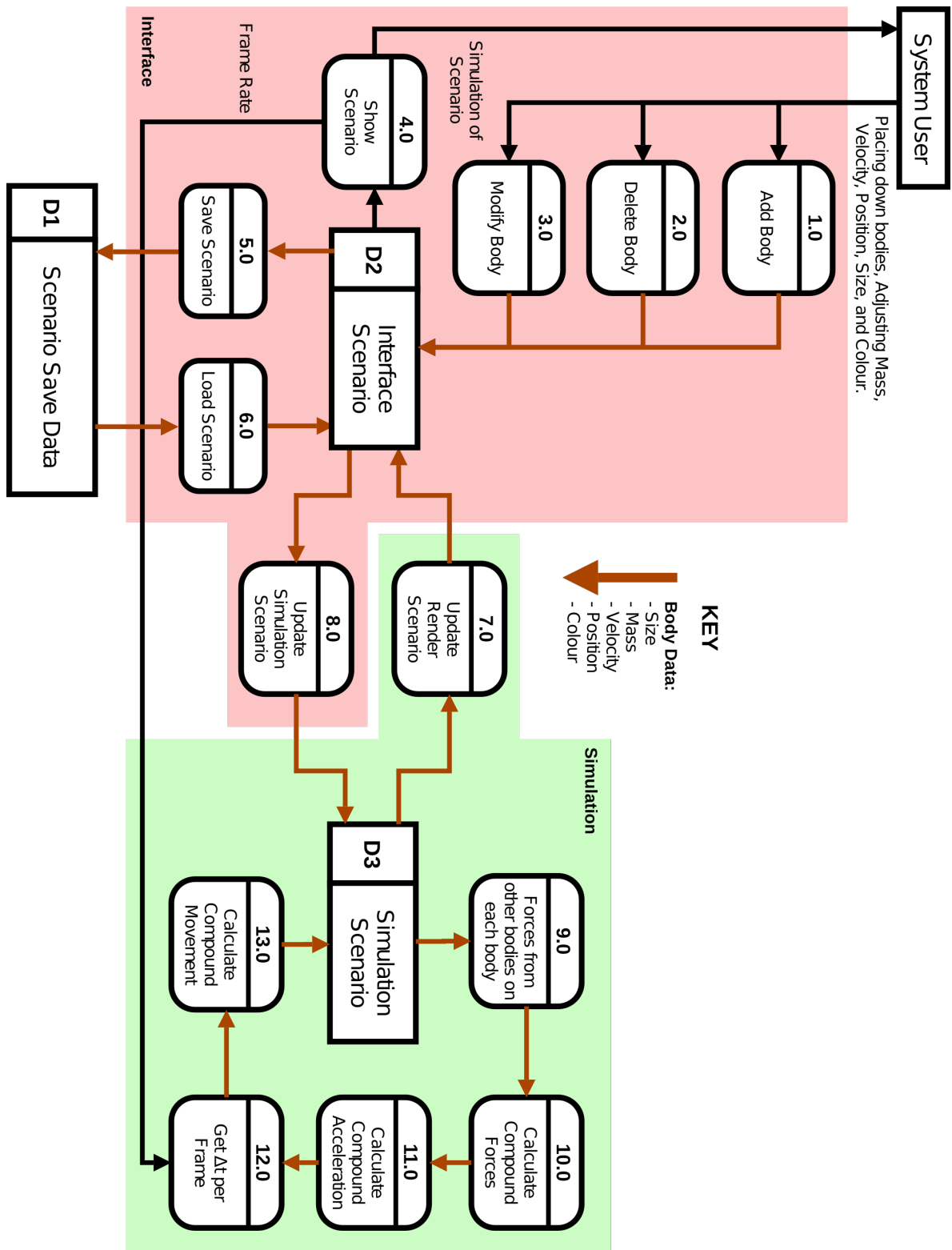


Figure 4: New System, Level 1

## 1.4. Objectives

### 1.4.1. System

1. The system must perform well under the kind of scenarios that are likely to need to be demonstrated by a teacher, for A2, this generally does not go over 2 bodies, however having a much larger number of bodies could be useful for showcase simulations.
2. Target performance is 60 frames per second with 100 bodies on screen.
3. The graphics used in the program should be rudimentary 2D vector shapes and lines of varying colours in order to effectively convey the information without becoming too crowded.
4. The program must be compiled to run under 32-bit Windows. it should also not require any external dependencies to be installed in order to run and should be able to be run from any folder on any computer.
5. The size of the scenario should be limited so that objects the precision on the position of the far bodies is not too large, which would potentially affect the simulation.

### 1.4.2. Processing

1. The application will use the gravitational force equation to calculate the force between two bodies and newtons second law to convert to acceleration.
2. Trigonometric functions are the currently known way to resolve vector forces to a global X and Y axis. Because of the computational cost these are likely to have an alternative should be investigated.
3. Collisions can be simulated using basic conservation of momentum equation, in order to calculate the resultant direction of travel.
4. The simulation should not be able to take up more than 500MB of system memory, meaning that memory usage must be profiled and a limit coded in.
5. The system should make use of dedicated hardware features such as the GPU to accelerate rendering, this will require external libraries for ease of porting
6. Multi-threading should be used to improve runtime performance where it is needed, as well as improve the overall user experience.
7. If multi-threading is used, the programming must be thread safe in order to prevent lock ups or race conditions, each thread must be aware of the state of other threads in order to prevent this.

### 1.4.3. User

1. The interface to the program should be basic and self explanatory, making use of the mouse for the placement (position) of bodies and definition of size, mass and velocity, the keyboard can be used for more precise entry of values.
2. It should be possible for the the user to save scenarios that have been set-up so that they can be accessed at a later date or shared with others. The data can be saved to a single file that can be loaded into the program, allowing teachers to set up a scenario outside lesson time to use as a demo.
3. The output of the program should be able to show up well on a projector, meaning that bodies must contrast highly with the background as well as with each other, it may be useful to allow the user to choose the colour of bodies.
4. The user should be able to change any of the values for bodies, position, mass, velocity, should all be possible to change on the fly even when the simulation is running.
5. The user should also be able to change variables such as the gravitational constant and time acceleration to suit smaller scale simulations which will be more visible and understandable to students.



## 1.5. Potential Solutions

### 1.5.1. C++, OpenGL, GLFW

C/C++ using the GLFW OpenGL library, this library provides a window management library, handling operating system specific tasks. OpenGL is mainly geared towards 3D rendering, however it is still possible to create a simple 2D scene in the software. The C++ standard library also contains support for multi-threaded applications.

#### Advantages

- I already have an understanding of the C Programming Language, the syntax used by C++ is near identical to C and has a more modern feature set as well as some extra features, such as support for object oriented programming.
- OpenGL makes direct use of the GPU for rendering of scenes, meaning that high performance will be easier to achieve than with a software based renderer (Using the CPU).
- The GLFW library contains support for multithreading, this can potentially be used to gain large performance boosts particularly for simulation. (Multithreading is also defined in C++ the standard library. (C++2011))
- GLFW takes a lot of the tedium out of programming for OpenGL, mainly in the creation and display of a window that fits in with the current operating system. It also contains support for keyboard and mouse inputs to the program. Traditionally, setting up a window for an OpenGL context would require a large amount of operating specific programming that does not warrant itself for small projects.
- GLFW is also a multi-platform library, making it significantly easier to port the program to other platforms should it be necessary.
- GLFW is an extremely minimal library, simply providing the utility to create OpenGL contexts to draw to and get the input from it. By including the GLFW header you also include the OpenGL headers as well, which are generally found in any standard development package.

#### Disadvantages

- On Windows systems, and external .dll (Dynamic Linked Library) will be required in the program directory in order for the program to run.
- The library may contain excess bulk that will increase the size of the program when compiled, even if the entire library isn't being used.

- There will be a large increase in the memory usage of the program due to the use of the library and set-up that is required, particularly for OpenGL, this will be in the range of 200MB.
- As the library is not written by me, there may be bugs in the code, however the library is open source and maintained by a relatively large group of people, meaning this would be quickly fixed.
- C++ is a complicated language and is known for having longer compile times than a language like C.
- GLFW has no inbuilt support for GUI, there are extra libraries available that could be used in order to create basic GUIs, such as AntTweakBar.
- The issue with both GLFW and AntTweakBar is that they are written with C style interfaces, this can make integrating into a C++ program in a clean way somewhat more difficult.
- Hardware will support a particular version of OpenGL and below, for example a GPU that supports OpenGL 3.0 will not support programs written for OpenGL 4.0, and most computers in the school will have a processor with new enough integrated graphics to support GLFW, which requires an OpenGL version above 3.2.
- OpenGL has some added complexity in terms of its use of matrix mathematics, something that can be something difficult to follow at times.

### 1.5.2. Pascal, Graph, Forms

Pascal contains an inbuilt unit known as graph, this allows the program to create a window and provide basic graphical output to the user.

#### Advantages

- Pascal is a language that I am familiar with and would be relatively understandable to write the program using Pascal, the more modern Free Pascal Compiler also supports Object Pascal, which is an object orientated subset of Pascal.
- The graph library is self-contained and part of the default pascal library included with the compiler, meaning that no libraries for external sources would be necessary for display output.
- I have a decent understanding of writing programs in Pascal from my studies for my AS qualifications.
- Lazarus, one of the main IDEs for Pascal has inbuilt support for Forms, these allow the developer to produce a GUI application, a small form could be created alongside the graph window for the modification of properties of bodies.
- Pascal does support multi-threading, which could be used to improve the speed of computation.
- Pascal can be compiled for any platform without the need to change code, assuming external libraries are not used.

#### Disadvantages

- Pascal is much less supported at present than languages such as C or C++, while the compiler still gets updates, it has much less of an active community, the most likely place where this would affect the program is in the optimisations that have been applied to the compiler and how it translates code.
- Pascal has been shown to have lower performance than C in certain examples, which could be a problem as the application will be quite demanding of performance.
- Pascal has no native way to obtain mouse input from the graph screen output, meaning that the program would either be limited to just keyboard inputs or an external library of some kind would likely be required in order to get mouse input.
- Pascal as a language is extremely verbose, using full words such as begin and end as opposed to braces, { and } as delimiters, this can lead to making the code

extremely difficult to read when there is a large number of these on the screen.

- Pascal also is not case-sensitive, which can lead to some confusion when a particular case styling is not kept throughout the code.
- Pascal has a large amount of functions included in the standard library, without needing to define any extra uses, leading to potentially larger binaries as these are linked in even if they are not used.
- The graph library is also not particularly efficient, and relies on CPU software bound rendering to draw to the display, meaning that it is unlikely that the target update rate could be achieved.

### 1.5.3. C++, OpenGL, SFML

SFML (Simple and Fast Multimedia Library) and C/C++ (Also has bindings for .NET, Java, Ruby, Python, Pascal and many other languages, this library is geared towards the development of 2D applications, it is a more modern alternative to SDL (Simple DirectMedia Layer) built in C++ from the ground up with object orientation in mind. SFML can be used a lightweight window manager for an OpenGL application or as a full multimedia library, making use of all of the features that it provides.

#### Advantages

- SFML is relatively modern and is built around the idea of object orientation.
- SFML provides a window management wrapper for the creation of OpenGL applications, similar to GLFW, however it provides many more features, such as a layer of abstraction on top of the OpenGL, Networking (TCP / UDP) and Audio (OpenAL) features.
- It is possible to choose which parts of the library that you want to use, split into the following:
  - Systems
  - Window - Depends on System
  - Graphics - Depends on System, Window
  - Audio - Depends on System
  - Networking - Depends on System
- SFML will work on any platform that has an OpenGL version above 3.0.
- The library is cross platform and can be compiled to run on Windows, Linux and OSX.
- Similar to GLFW, SFML removes the need to produce complex code that allows the basic set-up of an OpenGL context that fits in with the surrounding operating system.
- Multi-threading support is also provided, however there seem to be fewer features for controlling the flow of said runtime.

#### Disadvantages

- The Audio and Networking components of SFML are not required for this project, while it is not necessary to include them, it removes most of what sets SFML apart from GLFW, the only functionally different part would be graphics, which

contains pre-defined functions for loading images and drawing shapes, which is also unnecessary as this is relatively straightforward to using plain OpenGL, at this stage SFML is functionally identical to GLFW.

- Because of its relative newness (First released in 2007), SFML does not have nearly as much support available compared to GLFW, which was originally released in 2001.
- This library will also share many of the same disadvantages as GLFW due to its similarities.

#### **1.5.4. Conclusion**

Based on this analysis, I feel that Solution 1 would be best choice, namely because the minimal nature of the GLFW library, providing only the bare essentials for setting up an OpenGL context and system utilities such as multi-threading, it also can be programmed using C/C++, a language that I am familiar with and would be comfortable with writing a large scale program with, There are also tools that I can use that integrate well into the language to allow me to debug my code and step through it on a line by line basis.

While it would still be valid to use SFML and simply make use of the System and Window components and use it in a similar way to GLFW. It makes more sense to use the library that is more focused on the features that I plan to use.

## 1.6. Management and Tools

### 1.6.1. Linux and Development Environment

When it comes to the management and writing of the code, I will be using a bare-bones text editor with auto-complete plug-ins to make the experience somewhat smoother. A custom makefile will be used in order to create a custom build system which will make building an application with multiple source and header files much easier.

I plan on using the Linux operating system for the main development of the program, the main reason for this is that there are several tools that integrate very well into the Linux environment. As well as this, build tools and compilers are generally far better supported and more up to date on Linux.

In order to improve the organisation of code, different files will be split into separate directories source files will be found in *src/*, header files in *include/*, compiled object (.o) files will be stored in *bin/*. Having .o files in a separate directory means that the files can be kept after the program is linked, if any change is made to a particular source file only that file needs to be recompiled and the older files can just be linked together to produce a new executable.

Because I will be developing the software using Linux, I will need to work out an alternative build system to build an executable binary for Windows. It is possible to do this from within the Linux Environment. (e.g: CMake)

### 1.6.2. Git - Version Control

I will make use of tools such as *Git* (Originally written by Linus Torvalds, Creator of Linux) for the management and version control of the source code, this allows me to make changes to my code and upload it to a remote server. The code can be 'pulled' on other computers, changes can be 'staged' and 'committed', the resulting 'commits' can then be 'pushed' to the remote server. Git provides options to split changes to code into separate branches for making experimental changes to the code without making changes to the 'master' code.

Another feature of Git is the ability to automatically merge code, if a multiple changes are made to code on different computers (Such as forgetting to pull changes from remote.)

the modified code can be staged to a commit and the previous remote commit can be pulled from the server. Git will then attempt to automatically merge the code together in a way that preserves its function. (In the case that it cannot, it presents differences to you and asks you to merge the code manually.)

It is possible to completely revert code back to any previously commit, all of these features make Git an invaluable tool for the maintenance of source code, even for small projects. (It makes working on collaborative projects far easier also.) The fact that code is stored on an external server is an added bonus as it provides an off site backup and makes it possible to access code anywhere with an internet connection, but still make changes and commits to a local copy without a connection.

### **1.6.3. GDB - Debugging**

The GNU Debugger is an invaluable tool for the 'real-time' debugging of applications, assuming that the application has debugging symbols compiled in it is possible to run through code line by line stepping through code.

The tool allows individual variables to be looked at, allowing you to check that the program is running as intended. It will also return more information in the event of exceptions.

The main disadvantage to this tool is that even when the program is set to continuously run, it will run at a much lower speed than the program running outside the tool would run at, making this not a tool for profiling the performance of the application.

### **1.6.4. Valgrind - Profiling and Memory Analysis**

Valgrind is a tool which can be used for profiling certain aspects of the programs runtime, mainly looking at the programs use of memory.

The tool will print errors to a terminal in the case that it detects programming that could potentially be causing memory leaks to occur. After the program finishes or is exited the tool will print the total memory usage at the end of the program, it will also print out its analysis of potentially lost memory.



## Included Tools

- Memcheck - Memory error detector <sup>1</sup>
  - Illegal Access
  - Undefined Variables
  - Incorrect Memory Management
  - Overlapping Allocation
  - Negative Memory Sizes
  - Memory Leaks
- Cachegrind - Cache Interaction Simulator
- Callgrind - Call Profiling Tool <sup>1</sup>
  - Function Calls
  - Function 'Cost'
  - Includes Cache Simulator - Cachegrind
  - Other Events
- Helgrind - Multi-thread Synchronisation <sup>1</sup>
- DRD - Multi-threading Errors <sup>1</sup>
- Massif - Heap Profiler <sup>1</sup>
  - Memory Usage
  - Stack Usage
  - Long Term Leaks
- DHAT - Heap Allocation <sup>2</sup>
- SGCheck - Stack and Global Array Overrun
- BBV - Basic Block Vector <sup>2</sup>

---

<sup>1</sup>Will Use

<sup>2</sup>May Use

### 1.6.5. $\text{\LaTeX}$ - Document Authoring

Both this document and the manual for the application will be typeset using the  $\text{\LaTeX}$ word processor, Documents are written in comprehensive mark-up language which can then be compiled into a variety of formats, such as .pdf.

The system allows for extremely complex documents to be created, while keeping simple documents relatively easy to produce but keep the whole document to a consistent style, with automatic generation of a contents page. It also makes it easy to manage things like sources and references.

## 2. Prototyping

### 2.1. Simulation Efficiency

It is also worth noting that the method that I am using to simulate the orbits is generally known as being the brute force method, where every body in the simulation has all of its forces simulated based on every other body in the simulation, while this enables the most accurate method of simulation, it is not particularly efficient, and only really works well for small numbers of bodies, meaning that it would suit the purposes of this program.

There are some forces that will be calculated to be so insignificant that it will not majorly affect the outcome of the simulation. For example, the orbit of two bodies will not be massively affected by a third body that is extremely far away.

This principle is applied in the Barnes-Hut Algorithm, in which space is divided up into a quadtree. (octree for 3D simulation.) In this method, only particles/bodies that are close to the particle being simulated will be individually simulated. Particles that are further away will be simulated as one large particle based on its center of mass. This mainly increases the effectiveness if there are a large number of particles in a particular area, as this greatly reduces the number of interactions that need to be calculated.

While this method would have some benefits, the method is mainly designed for simulations with thousands of bodies or particles, something that is outside the scope for this project, it also makes use of a relatively complex algorithm that could be difficult to replicate in the time that I have available.

This may be something that is worth investigating at a later date should there be time, particularly if there is a need to simulate multiple thousands of bodies.

It may be possible to save some computational time by computing the values force forces inside a matrix. This would describe every particle to particle permutation that is possible and thus the forces between them in a undirected, weighted graph. While it won't really be used as a graph, it is a valid abstraction for the computation, removing duplication of a calculation that would decrease calculation performance.

## 2.2. Simulation Prototyping

The main thing that I wanted to check in the design of the project is that the mechanics that I plan to use will work correctly and as intended, due to the sheer volume of data involved, the easiest way to deal with this is to write a small program that runs through a simple scenario.

I started by writing a program that ran through a two body problem, using the intended algorithm and a fixed one second time step, I adjusted the gravitational constant to a more manageable number (1) that would work for small values of mass and velocity.

After the program was running, I set up a quick terminal output of the position of the body, these values were then pasted directly into a spreadsheet application and displayed on a graph, much to my dismay, it did not quite work correctly, the expected result of this should be a near perfect circle that loops back on itself, the system is theoretically closed and should remain constant.

However this output clearly shows that the object completes half an orbit, and then suddenly decides to veer off in completely the wrong direction, this results in the distance to the body becoming greater and the force diminishing.

When looking at the other values, particularly the angle of the orbiting body to the barycentre (0, 0, fixed mass), the angle flips its sign, from -1.5 Radians to 1.5 Radians, this causes the forces to also flip their sign and the forces then act in the opposite direction, causing acceleration away from the central body into infinity.

The fix for this was to check when the orbit crosses the X axis and flip the sign of the resultant forces when it does so that the system remains in orbit, this works in preventing this problem, however it manages to highlight a new issue.

As the body orbits, its height increases dramatically, and mainly on the negative X side of the graph, it is pretty clear that energy is not being conserved in this closed system and the orbiting body is gaining energy allowing it to reach a higher and higher orbit.

Upon further reading, it appears that this is simply a flaw of the system that I have elected to use, known generally as Cowell's Method, while this method is relatively simple to implement, it comes with significant disadvantages, the main one being is that a much smaller time base is required when bodies become extremely close together, meaning that a fixed time base will not be an option. The method does not maintain conservation of energy

and orbits will become higher over time due to error accumulated in the equations, the sheer amount of calculations that need to be done only worsen this effect, this is more obvious when looking at a highly eccentric orbit. while I understood that this would be the case, I did not consider that it would be quite this severe.

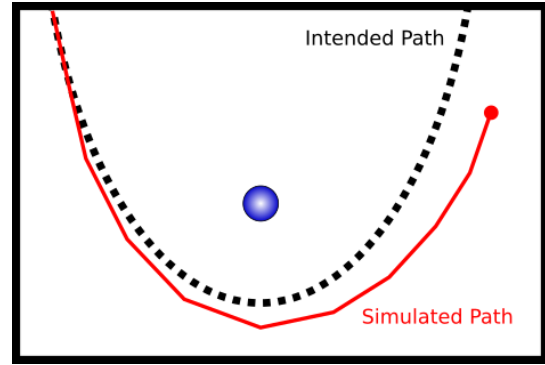


Figure 5: Integration Deviation

At a mathematical level, the algorithm is performing a method of numerical integration through the equation  $s = ut + \frac{1}{2}at^2$  which in essence will find the area under a velocity / time graph which equates to the distance travelled. This method performs this equation at every single time step, during the time the acceleration is constant and it is performed over the provided time step. This results in an issue when the orbiting body is close to the central body, as the distance travelled over the fixed time step becomes too large to correctly map to the intended orbit path, which causes the accumulation of error and instability over time. This is widely known as Euler Integration, Acceleration, Velocity and Position ( $a - v - p$ ) are all updated sequentially in each time step.

### 2.2.1. Leapfrog Intergration

After some research into this problem, it was clear that this is not a problem that can be solved while still retaining the simple sequential a-v-p calculation flow and would require some investigation into alternate methods. One of the numerical integration methods that is commonly used in gravitational particle simulations is Leapfrog integration, this method involves updating the velocity half time step out of phase with the acceleration and position, hence the name leapfrog.

Because it is somewhat difficult to step for half an iteration as integer loop counters are generally preferred for performance reasons, there are versions of these equations that are synchronised to update simultaneously. This effectively then becomes a calculation flow of  $\frac{1}{2}v - p - a - \frac{1}{2}v$ .

### 3. Design

## 4. Implementation



## 5. Manual

## 6. Evaluation

## 7. Code Listing

### 7.1. Early Prototypes

#### 7.1.1. Euler (Trig)

```
1 #include <iostream>
2 #include <cmath>
3 #include <unistd.h>
4
5 #define GRAVCONST 1
6 #define TIMESTEP 1
7
8 using namespace std;
9
10 // Types
11 typedef struct {
12     // Properties
13     double mass = 0.0;
14     // State
15     double force[2] = {0.0, 0.0};
16     double acceleration[2] = {0.0, 0.0};
17     double velocity[2] = {0.0, 0.0};
18     double position[2] = {0.0, 0.0};
19 } sBody; // Single Body;
20
21 typedef sBody bodyPair[2];
22
23 // Prototypes
24
25 // Functions
26 int main() {
27     bodyPair currentBodies;
28     int i = 0, xy;
29     double currentAngle;
30     double forceCalc, distance, diffX, diffY;
31     double forceX, forceY;
32     double deltaVelX, deltaVelY;
33     double deltaPosX, deltaPosY;
34
35     double deltaVelXs, deltaVelYs;
36     double deltaPosXs, deltaPosYs;
37
38     // Setup
39     // M0
40     currentBodies[0].mass = 100;
41     currentBodies[0].position[0] = 0.0;
42     currentBodies[0].position[1] = 0.0;
43     // M1
44     currentBodies[1].mass = 1;
45     currentBodies[1].position[0] = 0.0;
46     currentBodies[1].position[1] = 40;
47     currentBodies[1].velocity[0] = 1.58;
48     currentBodies[1].velocity[1] = 0.0;
49
50     //cout << "1x, 1y" << endl;
51
52     // Simulation
53     while (i < 500000) {
54         // Calculate Distance between Bodies
55         diffX = currentBodies[0].position[0] - currentBodies[1].position[0];
56         diffY = currentBodies[0].position[1] - currentBodies[1].position[1];
57         distance = sqrt(pow(diffX,2) + pow(diffY,2));
58
59         // Calculate Gravitational Force
60         forceCalc = -((GRAVCONST * currentBodies[0].mass * currentBodies[1].mass) / (pow(distance,2)));
61
62         // Find Angle Relative to X and resolve forces.
63         currentAngle = atan(diffY/diffX);
```

```

64 forceX = forceCalc * cos(currentAngle);
65 forceY = forceCalc * sin(currentAngle);
66 if(diffX < 0) {
67     forceX = -forceX;
68     forceY = -forceY;
69 }
70
71 // Put forces into body data
72 currentBodies[1].force[0] = -forceX;
73 currentBodies[1].force[1] = -forceY;
74
75 // Calculate Acceleration
76 currentBodies[1].acceleration[0] = currentBodies[1].force[0] / currentBodies[1].mass;
77 currentBodies[1].acceleration[1] = currentBodies[1].force[1] / currentBodies[1].mass;
78
79 // Calculate delta velocity
80 deltaVelX = currentBodies[1].acceleration[0] / TIMESTEP;
81 deltaVelY = currentBodies[1].acceleration[1] / TIMESTEP;
82
83 // Calculate position delta (S = ut + 0.5 * at^2)
84 deltaPosX = (currentBodies[1].velocity[0] * TIMESTEP + (0.5 * (currentBodies[1].acceleration[0] * pow(TIMESTEP,2))));
85 deltaPosY = (currentBodies[1].velocity[1] * TIMESTEP + (0.5 * (currentBodies[1].acceleration[1] * pow(TIMESTEP,2))));
86
87 // Update Velocity
88 currentBodies[1].velocity[0] += deltaVelX;
89 currentBodies[1].velocity[1] += deltaVelY;
90
91 // Update Position
92 currentBodies[1].position[0] += deltaPosX;
93 currentBodies[1].position[1] += deltaPosY;
94
95 cout << currentBodies[1].position[0] << ", " << currentBodies[1].position[1] << endl;
96 i++;
97 }
98 }

```

### 7.1.2. Euler (Vector) with $\Delta E$

```
1 #include <iostream>
2 #include <cmath>
3
4 using namespace std;
5
6 int main() {
7     // Body Data
8     double position_r[2], velocity[2], acceleration[2];
9     // Time Variables
10    double deltaT, simTime;
11    const double outputTime = 0.01;
12    double tc = outputTime;
13
14    // Setup Initial Parameters
15    cerr << "Enter_Timestep(s):_";
16    cin >> deltaT;
17    cerr << "Enter_Simulation_Time_(s):_";
18    cin >> simTime;
19
20    // Relative Position XY
21    position_r[0] = 1;
22    position_r[1] = 0;
23    // Velocity XY
24    velocity[0] = 0;
25    velocity[1] = 0.5;
26
27    // Calculate Initial System Total Energy
28    double e_k = 0.5 * (pow(velocity[0],2) + pow(velocity[1],2)); // Kinetic Energy
29    double e_p = - 1.0 / sqrt(pow(position_r[0],2) + pow(position_r[1],2)); // Potential Energy
30    double e_i = e_k + e_p; // Initial Energy
31
32    // Simulations
33    for (double t = 0; t < simTime; t += deltaT) {
34
35        // Calculate Relative Distance Squared
36        double r2 = pow(position_r[0],2) + pow(position_r[1],2);
37
38        // Calculate Acceleration
39        for (int c = 0; c < 2; c++) {
40            acceleration[c] = - position_r[c] / (pow(sqrt(r2), 3)); // a = p/r^3
41        }
42
43        // Calculate New Position and New Velocity
44        for (int c = 0; c < 2; c++) {
45            position_r[c] += velocity[c] * deltaT;
46            velocity[c] += acceleration[c] * deltaT;
47        }
48
49        // Output Data at Interval
50        if (t >= tc) {
51            cout << position_r[0] << "_ " << position_r[1] << endl;
52            //cerr << t << endl;
53            tc += outputTime;
54        }
55    }
56
57    // Calculate Final System Total Energy
58    e_k = 0.5 * (pow(velocity[0],2) + pow(velocity[1],2)); // Kinetic Energy
59    e_p = - 1.0 / sqrt(pow(position_r[0],2) + pow(position_r[1],2)); // Potential Energy
60    double e_f = e_k + e_p; // Final Energy
61
62    // Calculate Error
63    double error = (e_f - e_i) / e_i * 100;
64    cerr << "Percentage_Error:_ " << error << "%." << endl;
65    return 0;
66 }
```

### 7.1.3. Leapfrog (Vector) with $\Delta E$

```
1 #include <iostream>
2 #include <cmath>
3
4 using namespace std;
5
6 int main() {
7     // Body Data
8     double position_r[2], velocity[2], acceleration[2];
9     // Time Variables
10    double deltaT, simTime;
11    const double outputTime = 0.01;
12    double tc = outputTime;
13
14    // Setup Initial Parameters
15    cerr << "Enter_Timestep_(s):_";
16    cin >> deltaT;
17    cerr << "Enter_Simulation_Time_(s):_";
18    cin >> simTime;
19
20    // Relative Position XY
21    position_r[0] = 1.0;
22    position_r[1] = 0.0;
23    // Velocity XY
24    velocity[0] = 0.0;
25    velocity[1] = 0.5;
26
27    // Calculate Initial System Total Energy
28    double e_k = 0.5 * (pow(velocity[0],2) + pow(velocity[1],2)); // Kinetic Energy
29    double e_p = - 1.0 / sqrt(pow(position_r[0],2) + pow(position_r[1],2)); // Potential Energy
30    double e_i = e_k + e_p; // Initial Energy
31
32    // Calculate Initial Acceleration
33    double r2 = pow(position_r[0],2) + pow(position_r[1],2);
34    for (int c = 0; c < 2; c++) {
35        acceleration[c] = - position_r[c] / (r2 * sqrt(r2));
36    }
37
38    // Simulations
39    for (double t = 0; t < simTime; t += deltaT) {
40        // Calculate New Velocity and Position
41        for (int c = 0; c < 2; c++) {
42            velocity[c] += 0.5 * acceleration[c] * deltaT;
43            position_r[c] += velocity[c] * deltaT;
44        }
45
46        // Calculate New Acceleration and Velocity
47        r2 = pow(position_r[0],2) + pow(position_r[1],2);
48        for (int c = 0; c < 2; c++) {
49            acceleration[c] = - position_r[c] / (r2 * sqrt(r2));
50            velocity[c] += 0.5 * acceleration[c] * deltaT;
51        }
52
53        // Output Data at Interval
54        if (t >= tc) {
55            cout << position_r[0] << "_ " << position_r[1] << endl;
56            cerr << t << "\r";
57            tc += outputTime;
58        }
59    }
60
61    // Calculate Final System Total Energy
62    e_k = 0.5 * (pow(velocity[0],2) + pow(velocity[1],2)); // Kinetic Energy
63    e_p = - 1.0 / sqrt(pow(position_r[0],2) + pow(position_r[1],2)); // Potential Energy
64    double e_f = e_k + e_p; // Final Energy
65
66    // Calculate Error
67    double error = (e_f - e_i) / e_i * 100;
68    cerr << "Percentage_Error:_ " << error << "%" << endl;
69    return 0;
70 }
```

## A. Dialogue

I had several brief conversations with my teacher that were not documented during pre-planning stages of the project, these conversations consisted of outlining the idea for a teaching tool that could simulate a scenario set-up by a teacher in order to accurately demonstrate the fundamentals of circular and orbital motion, as a physics student I already have some understanding of what would be required from a project like this.

---

TO: Mr Snowden

In terms of my A2 Computing Project, I have a basic idea which I think would be workable and fit what you described previously, please let me know if you have any more ideas to expand the program. The program will feature relatively simple visuals, probably 2D, meaning it should run well on low end computers in the school, the program will simulate individual objects at the appropriate level in sped-up real time. You will be able to set up scenarios by introducing objects, varying the mass and velocity of the object before placement, multiple objects can be placed and they will interact with each other. (n-body problem could be interesting)

Creating the user interface for this will be somewhat of a challenge, likely I will try to integrate it all into a single window as it would remove the need for a secondary library for the user interface, information about what state an object is going to be placed down in (Mass, Velocity, Size, Fixed) As well as simulation state. (Time, Simulation Speed), A right click menu could contain certain options (Presets) but keyboard controls would be the primary method of control.

I have also considered implementing a system which would allow the saving and loading of different scenarios, allowing them to be set-up, saved and loaded at other times, which could be useful in a classroom situation, I would be interested to know if you would find this feature useful.

If you have any questions, let me know, I need to have a bit of a dialogue going in order to establish some groundwork, as the project progresses I will have questions for you in regards to particular decisions or compromises that need to be made along the way.

Many Thanks

Byron Theobald.

---

TO: Byron

A really useful innovation might be to have a graphical way of representing initial velocity, e.g. an arrow extending from each mass, representing the vector for initial velocity. I like the idea

of being able to save the scenario for later on, it would definitely increase the value of the tool.

A very difficult (but very useful) feature might be to be able to tick a box to display both gravitational field lines and lines of equipotential. This is an area of Physics which you haven't covered yet, but is fairly straightforward...

Thanks for trying a Physics application!

SDS

---

TO: Mr Snowden

Something I wanted to find out is how you would like the orbital simulator to handle collisions.

Based on my understanding of momentum, simply adding the mass of the two bodies and recalculating the velocity based on conservation of momentum would make sense.

Happy to hear your thoughts on this.

---

TO: Byron

Yes, provided momentum is conserved, everything will be fine. Remember that they will be inelastic collisions (ie KE not conserved).

Your idea below assumes no fragments are ejected - a reasonable assumption to avoid hideous complexity.

Best wishes, SDS

---

TO: Mr Snowden

I wanted to update you on the current progress of the orbital simulator, apologies for infrequency in that regard.

In its current state the code is sitting at just over 850 lines total.

So far the simulation portion is mostly complete, aside for the implementation of body collision detection and handling.

I have modified the simulation interface to allow it to be completely dynamic, meaning that it is much easier to define as many bodies as necessary in the code. While I have not tested the limits of this, aside from the memory and performance requirements there is no reason why there should be any limit to the number of bodies. (Part of my test plan revolves around benchmarking the performance of large numbers of bodies, up to 64. Back of the envelope



calculations suggest that at this number, each frame would take more than 250ms to compute, this is a severe reduction in performance but you are unlikely to need to use more than a few bodies.)

I have also nearly completed the interface to link together the multi-threaded nature of the program, allowing data to seamlessly be transferred between the renderer and the simulation, there is still some parts that need to be added, such as getting new changes made by the user sent to the simulation, ignoring any previously stored simulation data, user changes must take priority.

The next major task is to implement the renderer, this involves some understanding of matrix operations due to the nature of OpenGL programming, however I do not expect this to be a major issue as it is mostly abstracted.

After the renderer is implemented I will then focus on the UI and other utilities.

Attached is an animation of the raw simulation output showing a model sun-planet-moon system

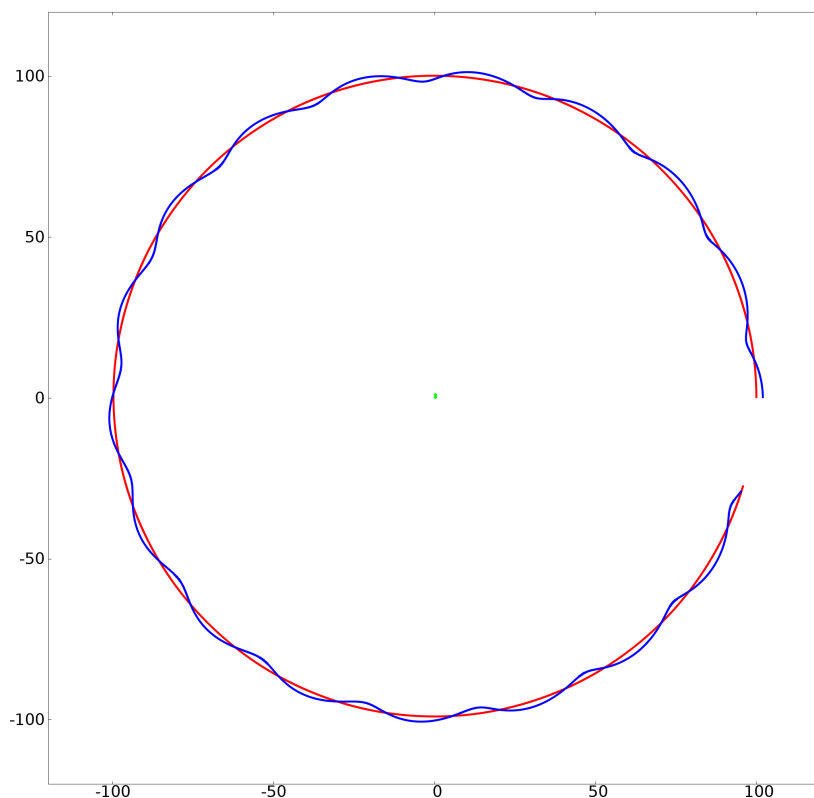


Figure 6: Three Body Sun-Planet-Moon System

---

TO: Mr Snowden

I implemented a basic version of the rendering module that i will be developing going forward.

Here is a preview:

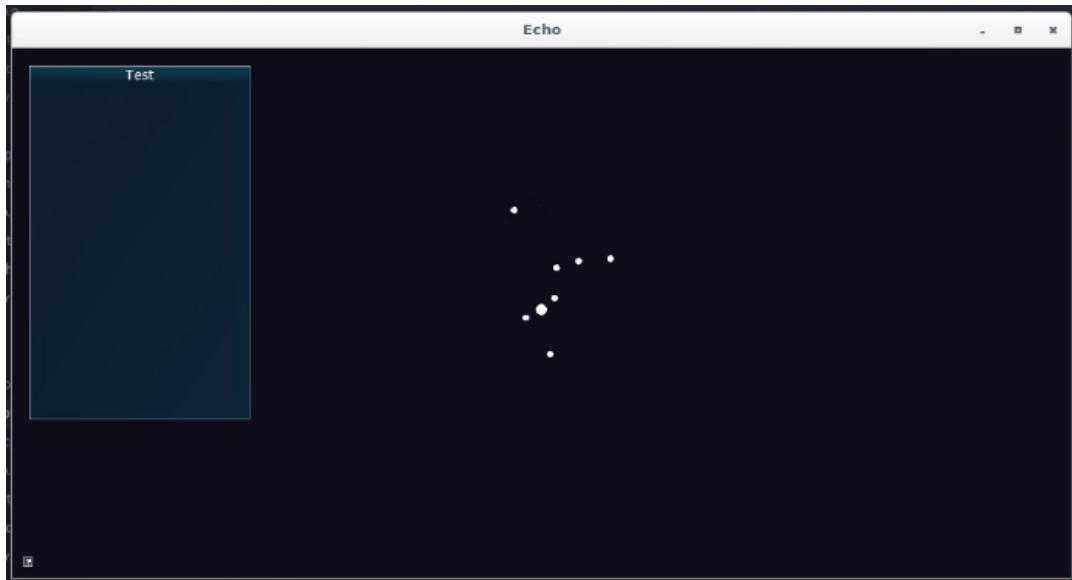


Figure 7: N-Body Graphics Test

Performance is looking extremely promising, will test with a very large number of bodies once I am able to create a distribution system.

---

TO: Byron

Thanks for the updates. this all looks very promising indeed - keep up the good work and keep me posted!

Best wishes, SDS

---

Prototype testing of the simulation showed that it can still support a massive amount of bodies and run smoothly, several orders of magnitude more than expected.

I had a few discussions with Mr Snowden about the potential applications of having large numbers of bodies, I agreed to implement the ability for the user to generate these superstructures; similar to the one shown below, through the interface.

This could allow the software to be used to simulate a very low body count galaxy collision.

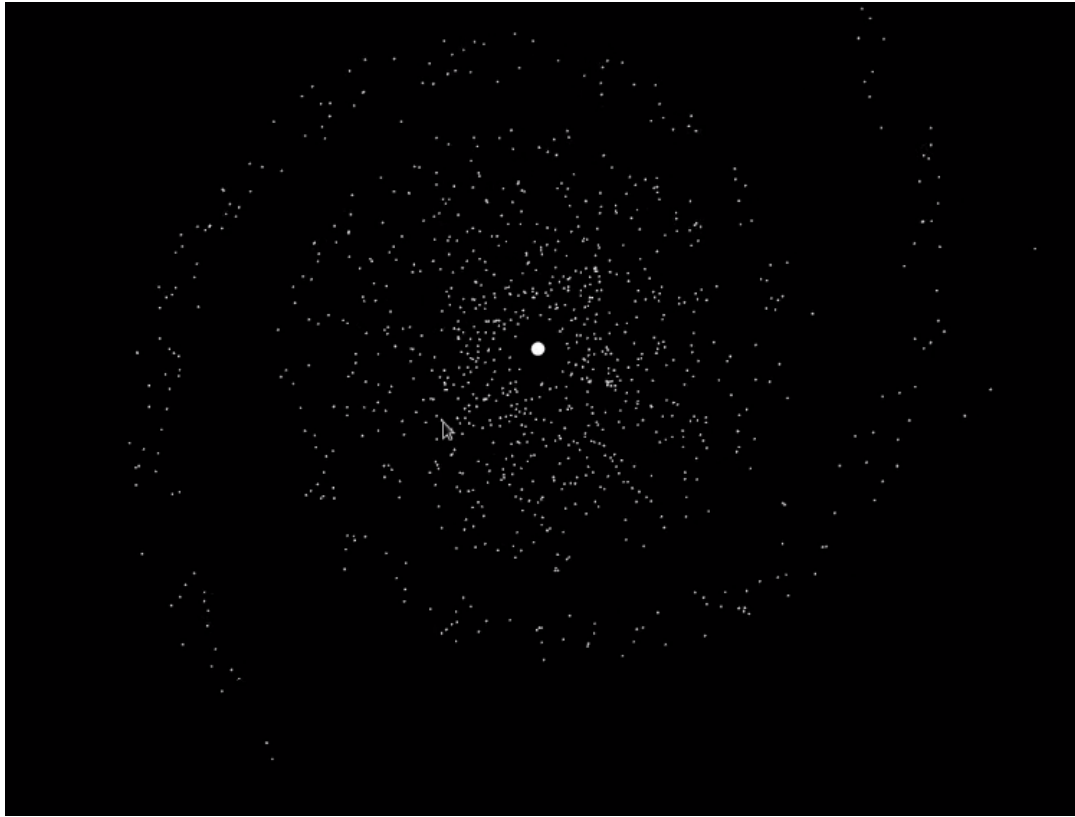


Figure 8: 1000 Body Superstructure

Something that we also decided on was the ability to turn on and off body collisions and combinations, as not having collisions results in the simulation staying more stable for longer.

While the earlier intention was to only consider the simulation for a few bodies, I will now consider the simulation for upto 1000 bodies, the benefit is that if the simulation is correct for 3 bodies, it will be correct for 1000.

The other benefit that this brings is the showcase aspect of it, something like this 'looks' really good to anybody who is studying or considering studying Physics, It is also far more likely to move into the area of inspiration for younger people, increasing their interest in Physics and academia in general. Spectacle is something that can easily affect the way that people think.