Torque To Me

Specification: A Modular Physics Engine

Richard Falconer (0502730),
Ben Hallett (0504569),
Alan Hazelden (0523756),
Leigh Robinson (0123489),
George Stanley (0525142).

October 24, 2008

Abstract

We are going to develop a platform-independent physics engine that will allow real-time simulation of rigid bodies in both 2D and 3D. The engine will be heavily componentised, utilising a solid and extensible object orientated design to facilitate its use in a wide range of interactive applications. Within this flexible framework we shall implement numerous published algorithms for collision detection and collision resolution, with the aim of assessing their strengths and weaknesses in particular scenarios. In addition to the core physics API and statistics we shall create a sandbox application that will facilitate the demonstration of the project by allowing an end-user to view and interact with live simulations.

1 Introduction

1.1 What is a Physics Engine?

A physics engine is, at its most basic level, a discrete software component that encapsulates some useful physical simulation algorithms for easy re-use. The exact nature of these algorithms may vary wildly with the intended application, giving rise to classes of engines that each deal with a subset of physical laws. Within these classes there are two further categories of physics engine, namely: *high precision* engines and *real-time* engines.
High precision engines focus on the quantitative quality of the simulations produced, but are not intended for use in time-critical applications, as high accuracy simulations are generally very computationally expensive. They are utilised to produce accurate simulation data for a very wide range of applications within both scientific fields and industry. Real-time engines, by contrast, only simulate what is absolutely necessary to produce a qualitatively realistic simulation, sufficient for use in interactive applications; most commonly games. Both the accuracy and types of simulation supported by this kind of engine are restricted by computational requirements to maintain interactive rates of simulation. Since the focus of these types of engines is to produce an interactive experience they tend to restrict their simulations to "everyday" phenomena such as Newtonian dynamics and fluids. It is these real-time, interactive simulations that this project shall focus upon.

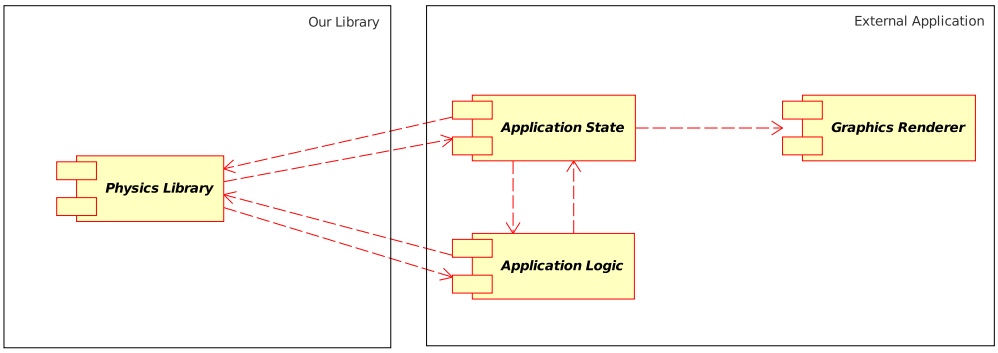

Figure 1: Component integration diagram highlighting how a physics component integrates within a host application.

Before physics systems became componentised into a discrete package they were bespoke and heavily tied to the original application making extension and maintenance difficult. With the separation of the physics components from the host application the way was clear for a true object-oriented framework to emerge to allow the physics components to be easily extended and re-used in other projects. Figure 1 shows the modern relationship between a typical application and a modular physics component. This change became especially important for real-time engines

as the available computational speed increases made year after year allow previously intractable simulations to be feasibly added. Many physics engine packages now exist, each having advantages and disadvantages over others in different situations due to the way in which they represent and handle collision detection and resolution. It is these such algorithm-level differences that our engine will be able to compare.

1.2 Motivations

1.2.1 Why are we creating a physics engine?

A physics engine provides an interesting software engineering challenge because although it is an easily componentised problem, these subcomponents can prove challenging to implement efficiently. A physics engine structure is naturally represented by the object-orientated paradigm, which is important as it facilitates teamwork and interoperability while providing experience in a modern programming style that is widely used on large scale projects throughout the industry.
In addition to providing experience with team development, a physics engine by its very nature is designed to be reused by a third party as part of a larger application. This provides experience in developing code that is easily extended and reused, again a vital skill in industrial development.
Furthermore, an application that implements the physics engine with a visual representation of the collisions is not only entertaining to develop and use but a very demonstrable end product. The project will actually be interactive and end users will be able to adjust the simulations to better understand exactly what has been implemented.

1.2.2 What will make our physics engine different?

There are already many open source implementations of physics engines both for two dimensional and three dimensional physics. Some examples include Box2D, Simple Physics Engine and the Open Dynamics Engine. What differentiates this project is that the engine is being designed from the outset to support the implementation and analysis of multiple algorithmic solutions to each subcomponent, such as broad-phase collision detection or contact resolution. The project will make it possible to easily compare two different implementations of algorithms and provide metrics to quantitatively analyse them.
It is outside the scope of this project to implement all of the features of the more established physics engines, especially within three dimensional environments. However it will implement most if not all of the primitive operations (collisions between boxes, spheres, and planes). This should allow a comparison between this physics engine and others in scenarios that only operate upon these primitive objects. The highly modular nature of our physics engine structure means that we can generally implement each algorithm without adversely affecting our ability to implement others. For this reason we do not have a strict ordering of algorithm implementations

in our development timeline; we plan to research their requirements then add as many as we can.

1.2.3 How to compare physics engines?

There are several factors that can dictate the effectiveness of a physics engine in a given scenario. The first is the type and algorithmic complexity of the algorithms employed. Naive implementations fail fairly quickly even with a moderate number of objects as their complexity is at best polynomial. Algorithms that have lower asymptotic complexity are always sought after, as the central goal is to maintain interactive simulation rates. As a consequence very good physics engines have algorithms that can achieve almost linear amortised running times.
Engines can also be judged on the quality of the simulation provided as measured by an appropriate metric. As mentioned, real-time physics engines have to use approximations to achieve their interactive rates. These optimisations not only reduce the stability of the simulation but also tend to skew the underlying physics. These inaccuracies could be quantitatively measured by analysing the physical parameters of the simulation. For instance, checking that momentum and energy are always conserved.
With a framework for implementing different algorithms, it will hopefully be possible to form some conclusions about exactly what scenarios a given type of algorithm is most suited to.

1.3 Customer

Nick Pope is one of the lead developers for the Warwick Game Design C++ Library (WGD-Lib). One area in which WGD-Lib is currently lacking is that it has no physics component, and Nick would like our project to result in something which could be integrated into it. We would not be responsible for this integration, but we must design and implement our system in such a way that it can be easily embedded into WGD-Lib, or any other application or library.

2 Project Objectives

Our objectives for the project can be summarised as:

> *We aim to create a highly modular and extensible physics engine component written in C++. It will efficiently support the simulation of rigid dynamic bodies under the influence of arbitrary forces (on a world and per body basis) and friction in both 2D and 3D environments. To support the creation and demonstration of the physics engine we intend to create a sandbox application that will allow for users to setup and interact with simulations. The sandbox application (or derivative of) will also allow*

us to conduct automated profiler tests to investigate the efficiency of the current widely used algorithms.

2.1 Core features

- Collision of primitive shapes and compound shapes

  Collision of primitive geometric shapes (2D and 3D) is the most basic level of rigid body dynamics simulation. Compound shapes made by simple intersection of sets of primitives will be a fairly straightforward extension.

- Modular architecture

  The framework needs to be sufficiently modular to allow the implementation of multiple algorithms for each problem, and ultimately to allow other developers to integrate the engine into their own applications. See Figure 2 for a high level relationship between the main components.

- Multiple broad-phase collision detection algorithms

  The so called "broad-phase" is the first attempt the engine makes at determining which objects are *possibly* colliding (rather than naively check all possible $O(n^2)$ pairs). A suitable broad-phase algorithm can greatly reduce the amount of computation required for the simulation by avoiding costly detailed collision detection. We will implement some (but probably not all) of the following algorithms:

  - Sort and sweep
  - Bounding volume hierarchies
  - Quadtrees/Octrees
  - Uniform grid
  - Spatial hashing
  - BSP trees

- Efficient narrow-phase collision detection for geometrical primitives

  The so called "narrow-phase" collision detection looks at the pairs of possibly colliding objects determined by the broad-phase and carries out expensive calculations to determine if they are indeed colliding. If they are deemed to be colliding then the interpenetration depth and contact normal are passed onto the contact resolver.

- Multiple collision resolution algorithms

  Collision resolution algorithms take a list of colliding objects and resolve the collisions. The choice of algorithm here will not only affect performance but

also the quality of the simulation, with regards to accuracy and stability. The following are some options we have already identified and will research their feasibility for the project in the coming weeks:

  - Resolving different amounts of penetration per-frame
  - For low-speed collisions, altering the coefficient of restitution
  - Non-linear penetration resolution
  - Generate contacts between nearby (non-touching) objects
  - Contact caching
  - Using Jacobian matrices to handle joints/constraints
  - Warm starting
  - Shock propagation

- Profiler allowing per-component performance statistics to be generated and analysed

  Within the physics library it will be important to keep track of statistics such as the amount of time taken to perform an operation or the number of bodies currently in the simulation. The profiler will be responsible for logging all this information to memory efficiently, and then providing mechanisms for other components to output this information. This information can later be processed into graphs and figures to aid with the analysis of algorithms within the system.

- Collision callbacks

  In addition to taking the position of objects in the world, it will be important for the physics engine API to be able to inform the application using it upon certain events within the simulation. For instance, a game application may require the library to alert it if an object collides with the player.

- Force/torque generators

  Force generators allow the application to define forces on bodies within the world. They can be used to implement attraction / repulsion between objects and more complex constructs such as Springs. Torque generators allow for the simulation of motors and other rotary actions.

- Simulation visualiser

  In order to test the Physics Library and ultimately to demonstrate its functionality, a visualiser will be required. This will take an initial state of the world, defined either in the program or within an external file, and then allow the user to see the movement of the objects in real time. It will also have the option to output relevant statistics from the profiler to the screen.

- Stable simulation of objects

  A challenging task within physics engines is making the simulation *stable*. Many of the optimisations that yield interactive simulation rates also introduce errors into the system, so that objects can resting upon one another can appear to move or collapse when you would expect them to remain upright. Different algorithmic approaches will need to be researched in order to try achieve a high level of stability while sacrificing as little of the speed as possible.

- . . . all the above should work in both 2D and 3D simulations.

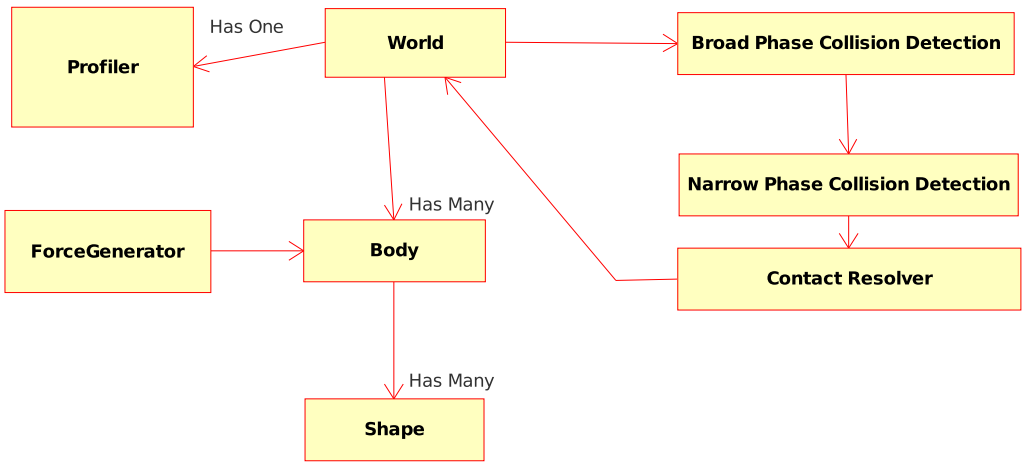

Figure 2: Major components (described in 2.1) of a real-time rigid body physics engine and how they are broadly related.

2.2 Further Work

The above constitutes a bare minimum of features that we must have in order for the project to be a success. In addition, we have identified other features which would further increase the utility of the project, but are more advanced and may not be feasible in the limited time available. We will have to closely monitor our work in this area to ensure that we do not waste time working on components that we cannot complete. We would hope to implement at least one or more of the following:

- Joints and constraints

  If we intend to provide a general purpose physics engine, it will be useful to be able to specify relationships between bodies: for example saying that two bodies are attached at (but can rotate around) a point. This enables the simulation of vehicles, ragdolls and other more complicated objects. To get acceptable performance, especially in 3D, it may be necessary to implement more advanced contact resolution techniques.

- Fully-featured sandbox application to accelerate testing and demonstrate the capabilities of the physics engine

  While a simple visualiser would suffice to confirm realistic looking behaviour, in order to adequately test all features, a more advanced sandbox application may be required. This would allow the construction of additional test scenarios and greater interaction with the simulation. While parts of the user interface may be shared between the 2D and 3D versions of a sandbox, it is possible that some sections would be more complex in 3D and would have to be dropped.

- Fluid dynamics and/or soft body simulation

  These can be to a certain extent simulated as a set of particles, with forces acting between the particles, which will both be supported by our engine. This may only be computationally feasible in 2D, or possibly not feasible at all without a large amount of optimisation.

- Advanced narrow-phase collision detection : arbitrary meshes

  Not all shapes can be easily represented as a union of primitive shapes (i.e. boxes and spheres). A more general approach would be to allow more complex shapes, such as convex polygon meshes. A naive implementation of these would likely give unacceptable performance, so we would have to turn to more advanced algorithms for computational geometry such as GJK or VClip. These would require a significant amount of work to both research and then implement.

3 Methodology

The project objectives lend themselves naturally to an agile development methodology, rather than a more static, planned approach. The aim is not to construct some monolithic application that must fulfil an immutable set of requirements. Rather, it is to implement a variety of features (in the form of various algorithms for the simulation of physical interactions) in a highly modular library. Consequently, requirements may change frequently as (for example) a new module depends on unimplemented functionality in another, or one feature is abandoned in favour of a more economical solution.
Our development methodology, then, must prioritise both inter-developer communication to ensure each team member fully understands every aspect of the project, and the flexibility and freedom to alter previous plans if and when they are discovered to be untenable (or unambitious). We must select tools that will not only aid us in collaborating to achieve our initial aims as stated herein, but will grant us the extensibility to go beyond them if desired. Above all else, if an agile methodology is to be maintained, testing must be rigorously carried out at every stage to allow us to identify problems at an early stage.
The principle drawbacks to this style of development - apart from the heavy time burden incurred by regular tests - are twofold. First, the emphasis on face-to-face communication detracts from the need to maintain formal documentation throughout the project (indeed, many agile methodologies actively discourage it), but our project will require such documentation for the end-product to be useful. Second, the ability to embellish and extend plans in the middle of development allows for a tendency towards feature creep, which may bloat the project and distract us from our primary objectives. Both of these problems are addressed within our strategy: in the first case, documentation is considered a primary objective as important as any feature, and has a team member dedicated to its maintenance; in the second case, our group organisation and collaboration techniques ensure that feature requests are dealt with in a sensible order that will prioritise the completion of the most important tasks. The project itself is actually reasonably feature-creep-proof thanks to its modular nature, in that a given new feature is unlikely to break older ones, and a single unimplemented feature will not render the entire product useless.

3.1 Group structure

Documentation Co-ordinator: Richard Falconer
The task of the Documentation Co-ordinator is to ensure that documentation is maintained in a suitable condition for the customer to easily understand and use the product. This individual also carries the responsibility of organising and overseeing the creation of other important documents (such as this one, and the project presentations), editing them for style, correctness, and consistency, and ensuring that they are completed on time.

Team A:
Leader: Leigh Robinson
Developer: George Stanley

Team B:
Leader: Alan Hazelden
Developers: Richard Falconer, Ben Hallett

The twin development teams do not operate in isolation; rather, they form small clusters of support, within which each developer will first go to their teammate(s) if they need to discuss an element of the project more immediately than a project meeting can be organised. Each team may be focussed on a different area of the project at a given time, allowing development to avoid serious bottlenecks. The team leaders are responsible for selecting and delegating tasks, ensuring that deadlines do not perpetually slip and alerting the rest of the team to serious issues should they arise.

3.2 Development Strategies

Agile software development methodologies heavily emphasise daily face-to-face communication between project members to keep everyone abreast of issues in the development, and ours is no exception. All project members will see one another and update each other on the project every day; this process is augmented through regular virtual contact via e-mail and instant messaging software. A more structured meeting at which all group members are present is to be organised weekly (or more frequently, as required) to assess and solve any problems that have arisen and to ensure compliance with the schedule and objectives.

3.2.1 Collaboration Issues

Co-ordinating the collaborative authoring of code by five group members simultaneously working on a sophisticated software engineering and documentation project gives rise to some not insignificant problems in organising and making available the work a given project member has done. Fortunately, however, these problems are also not insurmountable: there exist established tools for such collaboration, and by making use of them, much of the administrative overhead of the project is relieved.

- Subversion

  Subversion is a version-control system designed for managing large projects with a great deal of code. It is able to merge two distinct versions of files modified separately by group members into a single, complete file (or, failing that, to highlight the differences between them to ease the task of resolving the conflict manually). It serves as a backup system, and allows one to roll

back to previous versions if a newer one doesn't work for some reason. It (or something like it) is frankly essential for any group development effort.

Subversion is one of several competing technologies, other notable ones being CVS and GIT. CVS was discounted because of its age; Subversion is newer and provides more functionality, especially with regard to conflict resolution. GIT is newer than SVN and - arguably - more powerful. It is complicated to set up, however, and poorly supported by the other tools being used on this project (such as Redmine, below, and project members' IDEs). Ultimately Subversion was chosen because it strikes the correct balance of functionality and simplicity for this project.

- Redmine

  Redmine is a project management web application. It provides a group with tools to track both the development of features towards certain development milestones, and the emergence of bugs or problems with the project. It implements a forum for informal group communication and a wiki for more permanent developer documentation. It also integrates with the subversion repository, providing a more human interface for browsing the repository for specific files and viewing the most recent edits. Redmine can be configured to provide each project member with updates in real time via e-mail or RSS/Atom feeds when relevant changes occur to the project.

  This webapp will be critical to co-ordinating development efforts and preventing feature creep, since each feature works toward a particular milestone, and appears in a strict order. It will be easy to determine which features are important to the final version, and which were dreamed up later on in the project. The calendar features will also enable us to distribute the dates of meetings or deadlines.

3.2.2 Development tools

This project is to be developed in C++. This decision was reached for a number of reasons. Firstly, it is a language with which every group member was already familiar. Secondly, its object-oriented nature greatly facilitates the modular structure of the library we intend to create. It is also the language in which the customer intends to develop games in the future, and the language in which most games are produced, due to its power and speed, meaning that the library will be most useful written in C++.
As a long-established, well-used development language, C++ also has many other code libraries that make the development of certain features easier. The SDL and FTGL libraries will aid development of rich end-user tools such as the sandboxes, while Boost and TinyXML will provide less conspicuous features to the library itself, to name but a few.

3.2.3 Documentation

Although agile development has little time for documentation, ours will be considered a deliverable. With each project milestone, then, will come a requirement to have complete documentation for that state of the project. Our Documentation Co-ordinator will be tasked with ensuring this gets done by the relevant developer on time. Such formal customer documentation will be presented in a combination of LaTeXand doxygen-generated HTML, since both are standards for the production of such documentation. For certain tasks requiring highly specific documentation, we will also use Umbrello, a UML modelling tool which can generate class diagrams and use case diagrams automatically from the code. This not only greatly simplifies the task but ensures that the resulting diagrams are provably accurate representations of the real implementation.
Less-formal documentation, such as the information relayed between developers to maintain an understanding of others' work, will be light; most such information will be conveyed verbally. That which cannot be easily remembered or articulated will be recorded in the project wiki and in comments in the source code itself.

3.2.4 Testing

The most important element of any agile development methodology is frequent unit testing. Without such testing, it is impossible to identify problems with the code (and thus with the objectives as they stand), and the advantage of flexibility that the methodology provides us will not be realised. Therefore, we require that every component, once developed, is fully tested before being committed according to some test cases decided by the developer themselves, and that each milestone is tested by the group as a whole in a weekly meeting. Once again, the modular nature of the project facilitates such testing, since small components can be isolated and tested to allow for the swift detection and location of errors.
The nature of the project makes it difficult to test: its aim is verisimilitude, which is both qualitatively assessed and somewhat subjective. While this can be checked by the developers themselves, it will be best served by gathering as many opinions as possible. To this end, particularly significant testing sessions will involve demonstrating the current product to as many people as possible and recording their assessment of its attempt to provide a close simulacrum of real Newtonian dynamics.
The project may also be more quantitatively assessed on its stability and speed under various loads, since its aim is real-time simulation. This will be tested with a separate profiler class that measures the execution time of a set of calculations on a given state of the simulation. This will allow us to identify bottlenecks that are slowing the calculation down, and optimise them for better performance.

4 Timetable

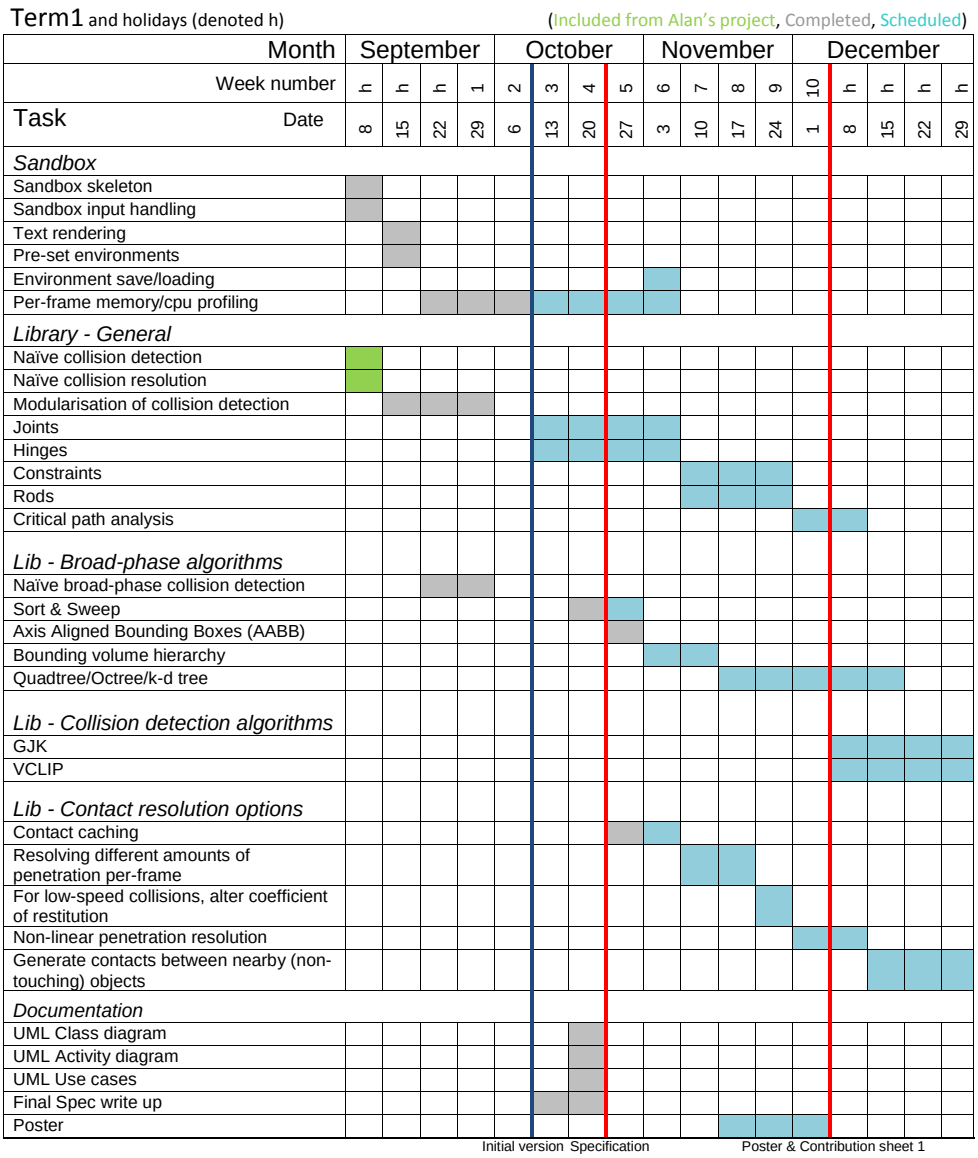

Figure 3: Gantt chart detailing term 1.

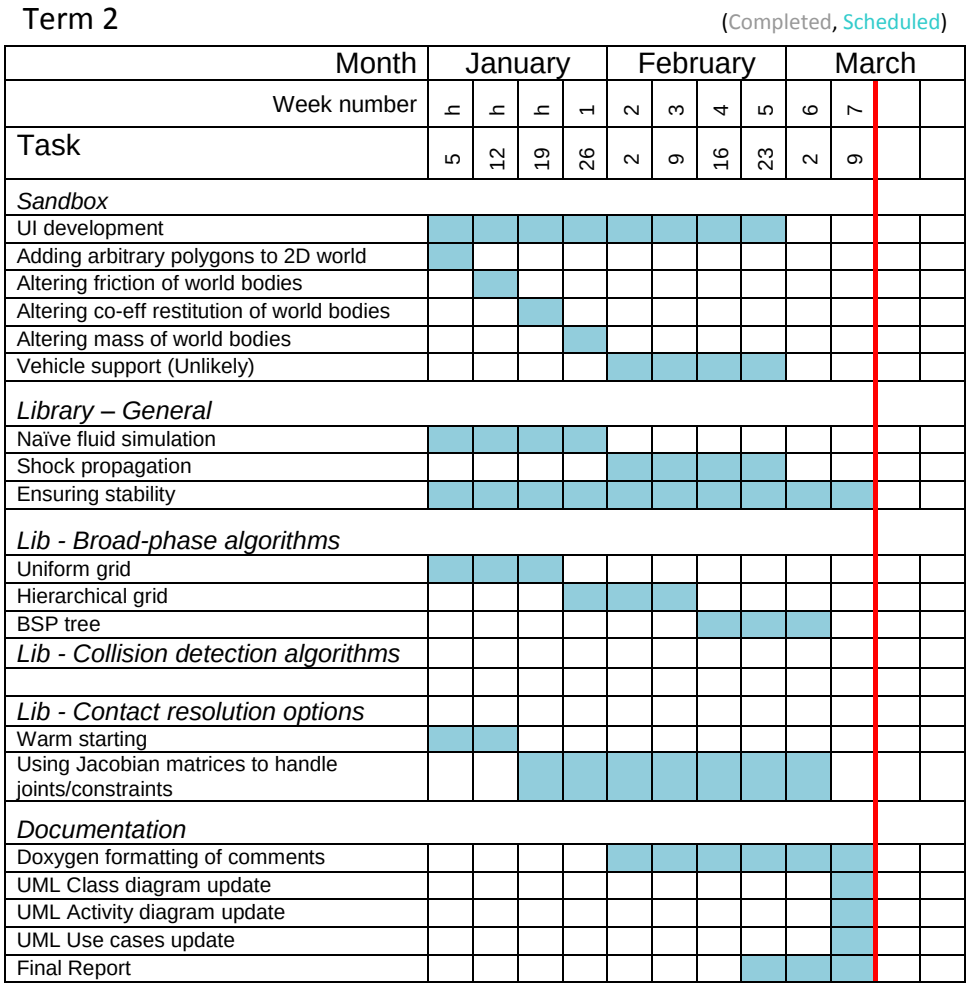

Figure 4: Gantt chart detailing term 2.