Large Polygon Collider

A Real-Time Modular Physics Engine

THE UNIVERSITY OF WARWICK

Introduction

Aim
To create a modular real time physics engine library in C++.

What is a physics engine?
A Physics engine is a library that is used by other applications via an API to simulate collisions between bodies in an environment. It is responsible for resolving everything from simple collisions between primitive objects to handling vehicles and fluids.

Real-time physics versus high precision physics
Physics simulations can either be performed in real time or offline. Offline simulations (or High-Precision systems) aim to be completely physically accurate and suitable for scientific models of physical interaction. Real time physics engines on the other hand trade some accuracy for speed, so that although they generate visually convincing simulations, the physics may not be entirely realistic. This is necessary in the context of an application like a game where the amount of time to compute physics is bounded to ensure that the environment remains interactive.

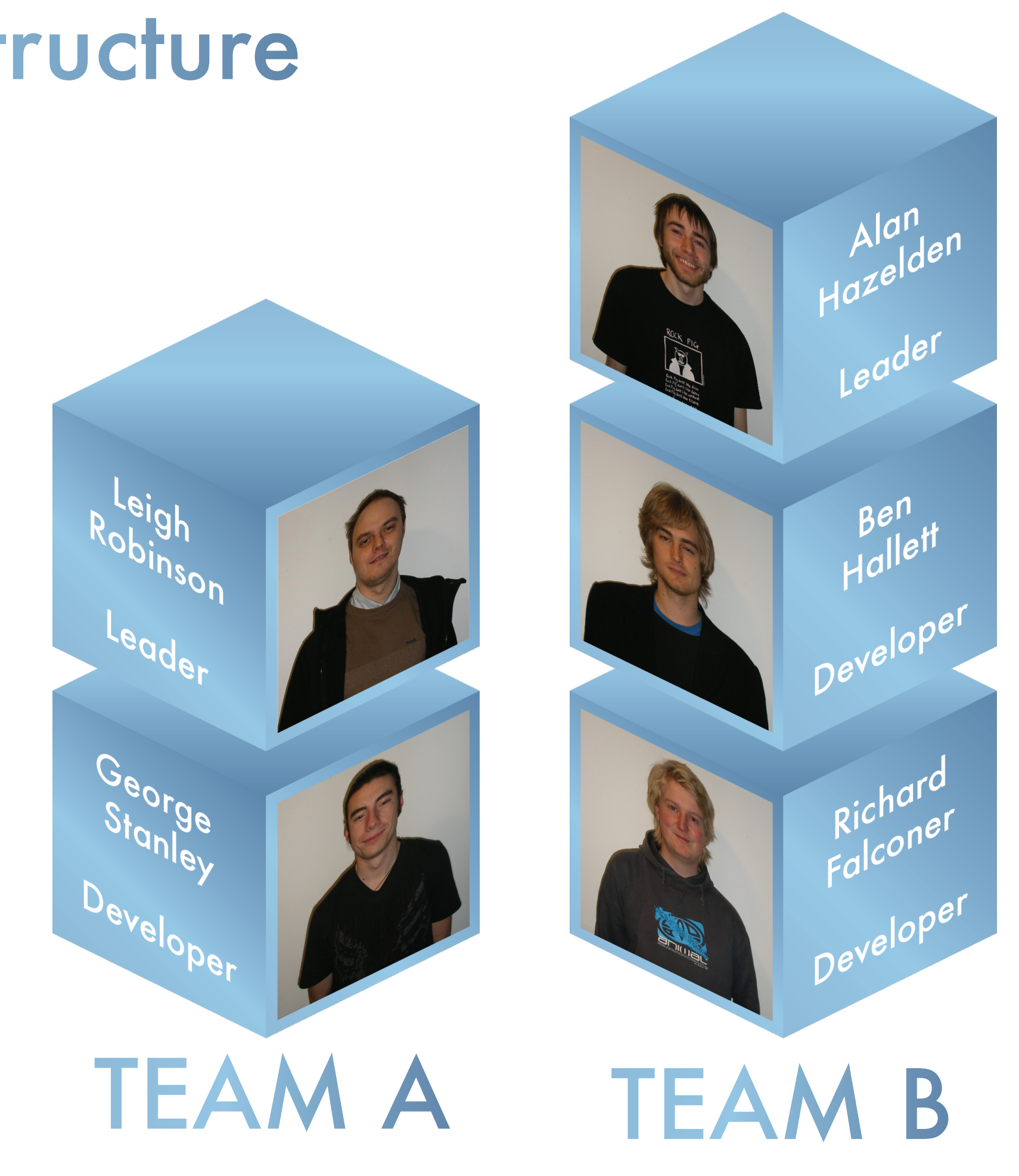
Uses of a real time physics engine
- Games
- Real time 3D modelling / CAM
- Computer Aided Design
- Animated movies

Requirements
- 2 dimensional and 3 dimensional rigid body physics handled by the same library
- Joints and constraints for dynamic behaviour (springs, ropes, rods, pivots, hinges, etc...)
- Callback system for integration with third party applications
- Profiler for determining performance of modules
- Infrastructure for implementing and comparing different algorithms
- Cross platform support - primarily for both Windows and Linux
- Sandbox application to demonstrate functionality

Customer
- Customer is putting together a complete game engine
- Our physics engine would provide a component of that engine
- This will require well documented and rigorously tested APIs
- Especially important is the cross platform nature of the project

Motivation
- Interesting software development project with numerous avenues to consider for extension
- Potential to be useful to other developers, especially in small game projects
- Scope for research into different algorithms and implementations
- The highly interactive nature of the sandbox - realtime physics simulations are FUN!

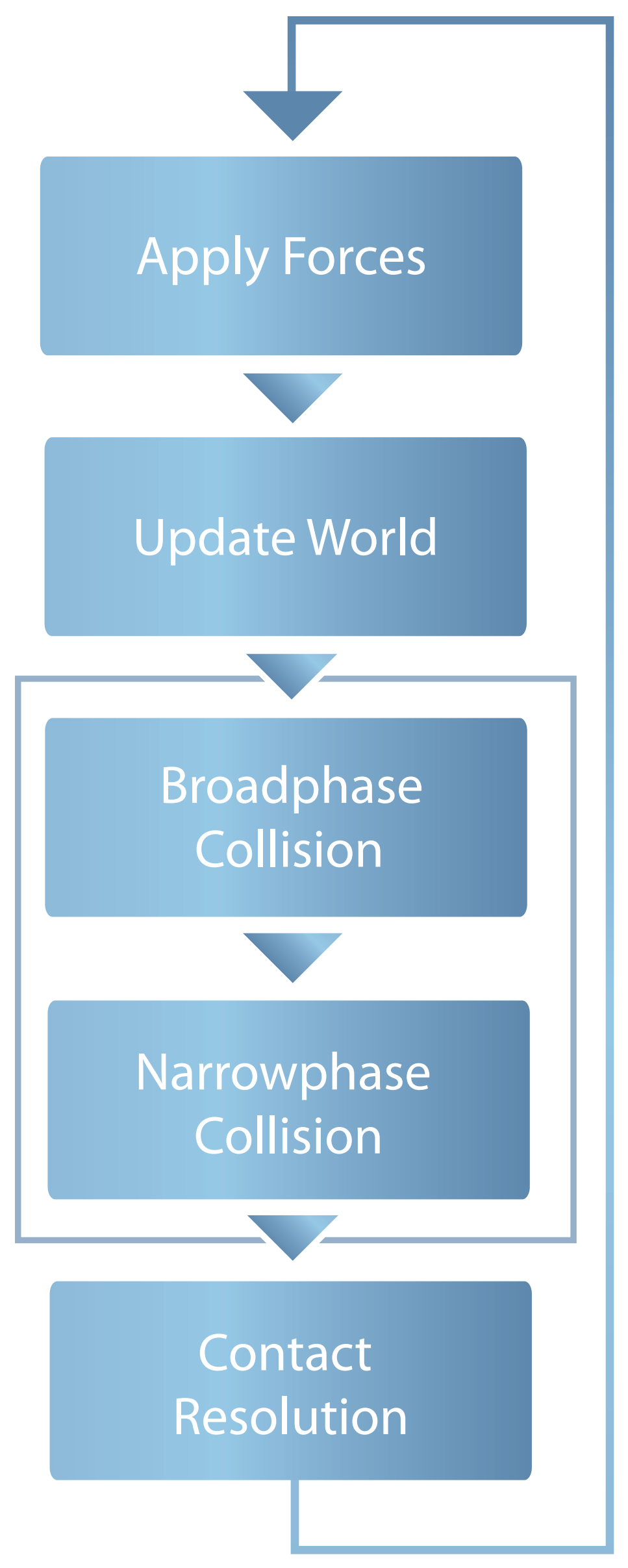
Group Structure



Design & Implementation

The flow of the physics system is detailed in the opposite figure. As shown there are 5 distinct phases that need to be completed for each physics update, namely:

- **Apply Forces**
  This is the first stage of the simulation pipeline. Its sole job is to iterate through the dynamic bodies present in the world and apply any forces to them such as gravity, etc. The `ForceGenerators` all implement a standard interface that says nothing about how the force is generated - allowing for complete flexibility.
- **Update World**
  This simply takes the current time step and performs numerical integration calculating the new velocities, positions and orientations from the current forces.
- **Broadphase Collision**
  This component detects bodies that might be colliding. Naive checking generates $O(n^2)$ possible collisions between n objects that need to be checked. This can be greatly reduced by utilising the spatial distribution of the bodies. There are numerous approaches to accomplish this, including: sweep-and-sort, quad/oct-trees, and spatial hashing techniques.
- **Narrowphase Collision**
  This component is responsible for taking the pairs of possible collisions generated by the broadphase and checking if they are actually colliding. There are many ways to accomplish this depending on the representation of the underlying collision geometry. Efficient techniques borrow heavily from computational-geometry to accelerate the checking process but still remain relatively expensive hence the initial broadphase step to keep the number of such checks to a minimum.
- **Contact Resolution**
  The collisions discovered in the previous stages must be resolved. Meaning that any inter-penetrations (due to the discrete time-step) must be corrected and a physically correct collision response (taking into account both angular and linear velocity, contact friction and restitution) must be created while still being computationally tractable for realtime applications. This stage is key to *stability* of the physics simulation and will ultimately determine the utility of the produced physics library.



A goal from the outset of the project was to re-use as much of the code as possible and not replicate common functions for both 2D and 3D. With this in mind we adopted a heavily templated design allowing almost seamless creation of these common functions but retaining the ability to specialise where required.

Development Methodology

Our project objectives lend themselves naturally to an agile development methodology, rather than a more static, planned approach. This agile methodology:

- Allows flexibility to adapt earlier designs when they become unworkable.
- Prioritises face-to-face developer communication over the creation of heavy documentation.
- Iterates over design, build and testing for each new feature added to the functional whole.
- Requires frequent testing of the entire system.
- Ensures a functional, evolving system at every step of the process.
- Requires regular contact with (and feedback from) the customer.

Development Teams
The twin development teams do not operate in isolation; they form small clusters for collaborative development. 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.

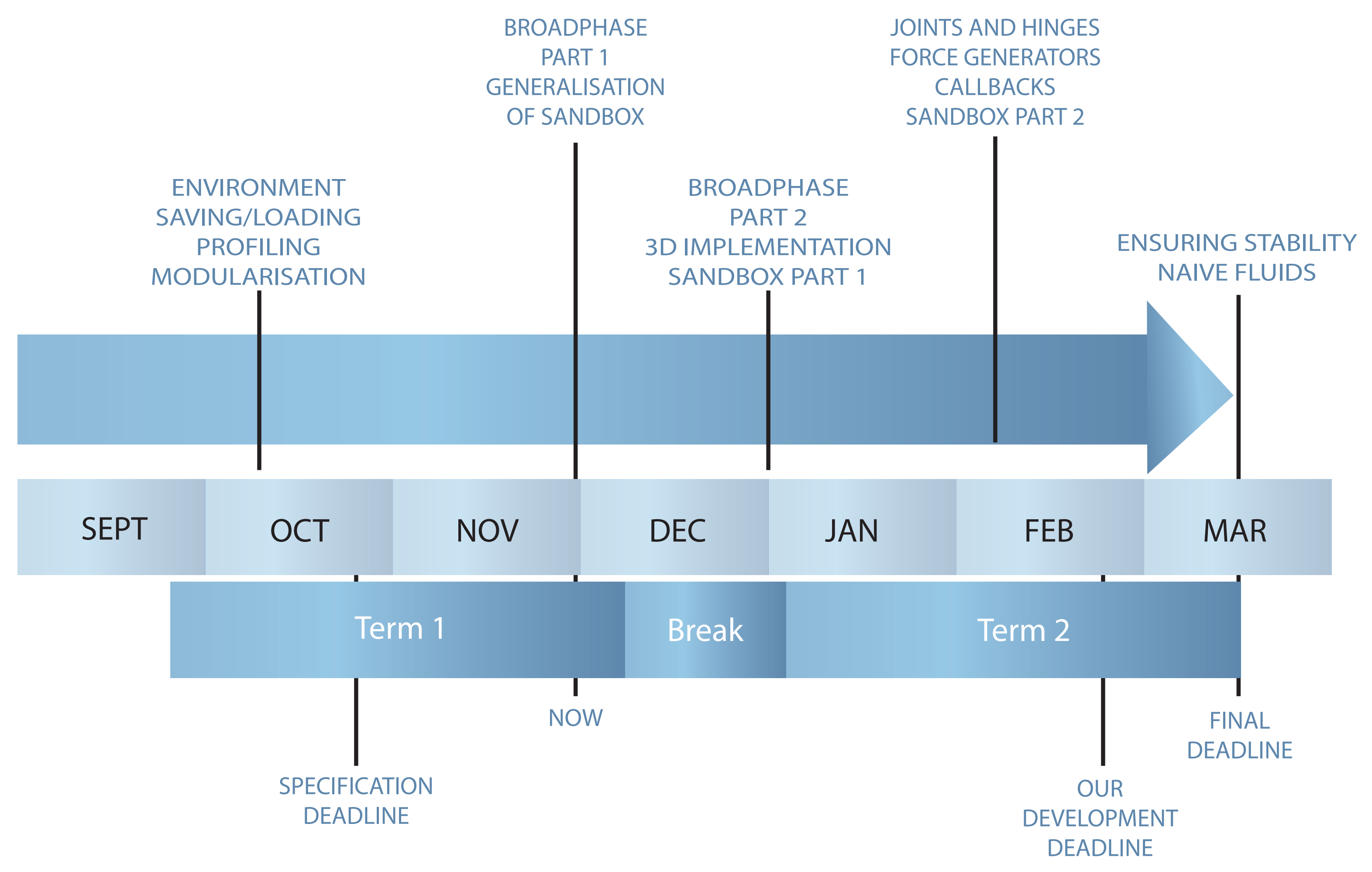
Development Tools

Subversion is our version-control system; it manages and backs-up our code, allowing merges and roll-backs to always maintain a working system.

Redmine is our project management web application. It lets us track progress towards development milestones and the bugs or problems with the project, as well as facilitating communication and (light) documentation.

C++ is used as a mature and very powerful programming language with almost exclusive support in the application domain. Many libraries are available and easily included, such as SDL and OpenGL for graphical output and FreeType and FTGL for text rendering.

Timeline



At the start of the project, many team members had limited experience with C++, especially some of the more advanced features like templates - which are used throughout the project to support genericity between 2D and 3D. A short series of lectures and workshops on some of these topics was presented by Alan to bring these members up to speed.

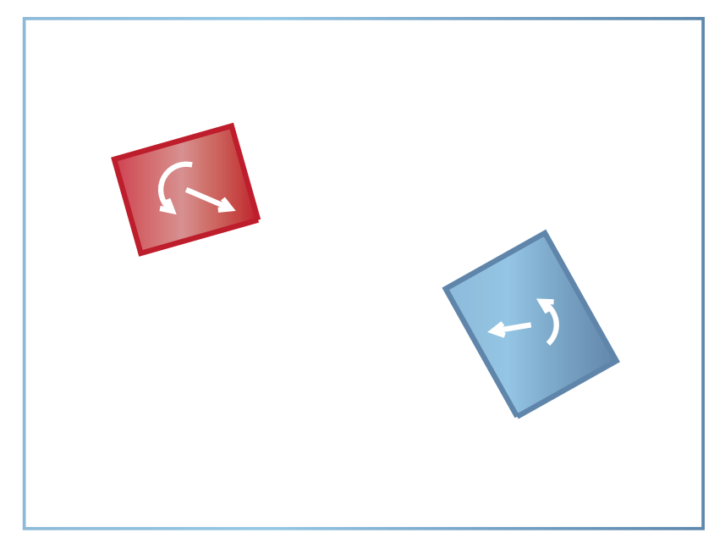
Our initial timetable did not take into account the disparity between our workloads for other modules in Term 1 and in Term 2. We have shifted some objectives to term 2 to compensate for this. We also underestimated the time taken to set up a working text rendering system. We partially implemented a range of text libraries before finally settling on FTGL.

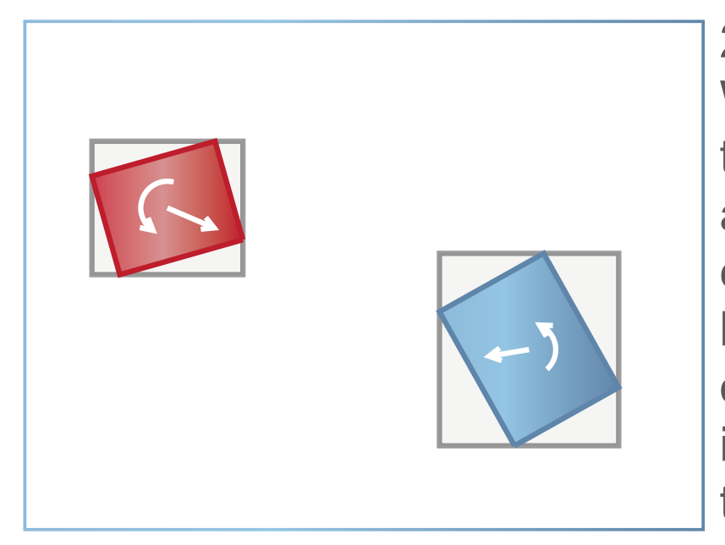
When formalising project goals, we had a wide range of discussions on our interpretation of the project goal. At one point we considered expanding our spec to include implementing the Physics Abstraction Layer (PAL) API. This would have allowed comparison between our implementation and other leading physics engines. Additionally, we thought it would allow us to easily integrate with Scythe Physics Editor, which would provide an 'out of the box' 3D visualiser. Further research revealed that implementing Scythe would be substantial work on top of implementing the required PAL interfaces. It was decided that these 'extras' deviated the project from its core goals too much and were subsequently dropped.

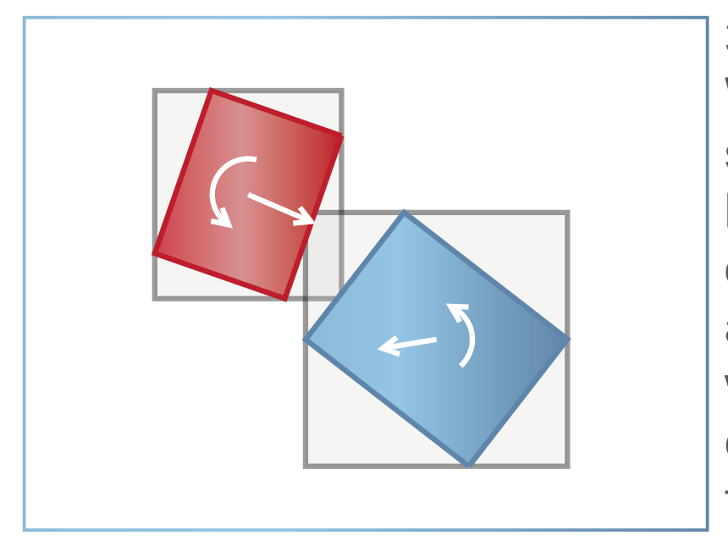
Scythe integration was motivated by a desire to not spend an excessive amount of time on a potentially difficult 3D visualising and world editing sandbox. We got round this by deciding to make our current 2D visualiser more generic, at the expense of some features planned for the full 3D visualiser. Our 3D sandbox has now been subsumed by the increased scope of the 2D sandbox. By adding a simplistic 3D camera to the 2D sandbox we avoid a lot of code duplication at the cost of losing some 3D world manipulation that would take too long to implement and generally be beyond our scope.

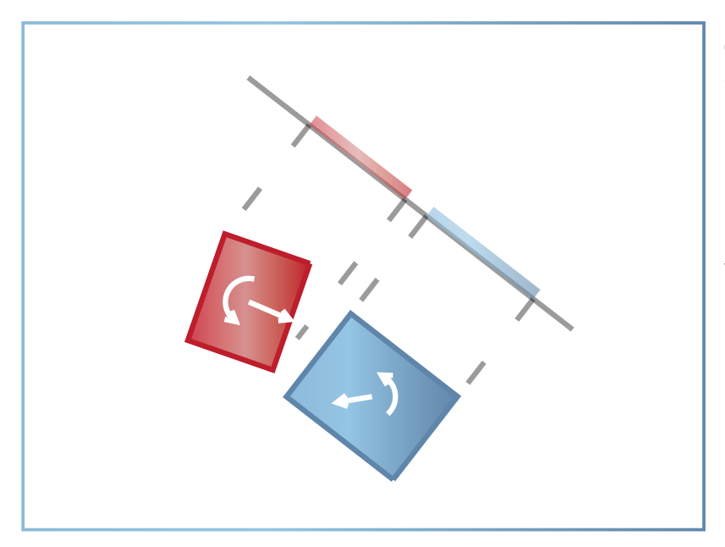
A scope we have expanded however is that of the 2D sandbox. Previously our aim was purely a very simple visualiser/sandbox, with no real usable UI, only somewhat obscure events hard bound to keyboard keys - nothing more than a testing sandbox. After discussion and seeing projects such as Phun, we have decided to make a far more user friendly sandbox with clear mouse-based controls.

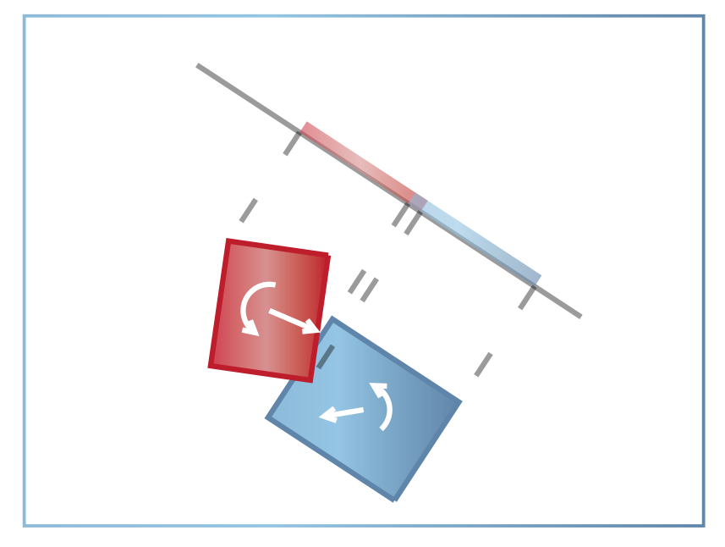
Collision Example

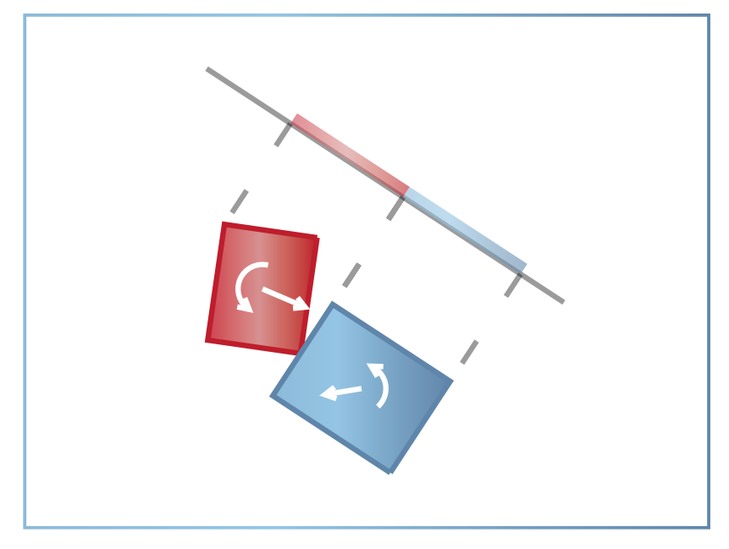

1
Here we have two rectangular bodies, each with its own position, velocity, and rotation speed.

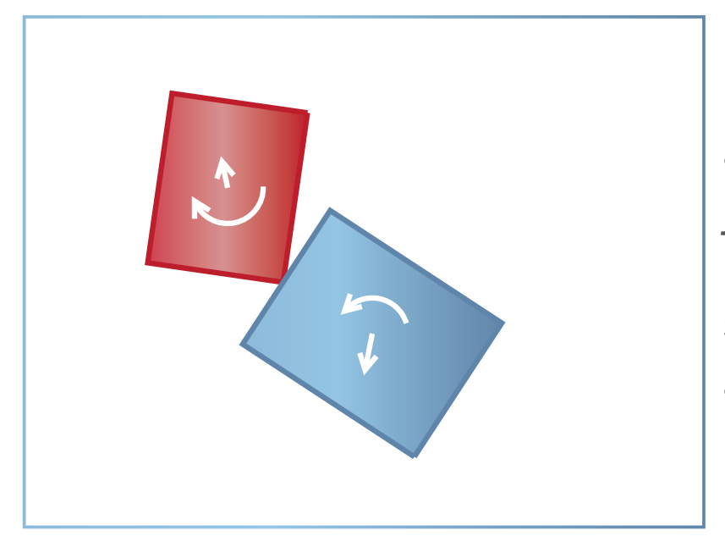

2
We perform a bounding box test to determine if the objects are potentially colliding. In the example here the bounding boxes for the shapes are not overlapping, so it would be impossible for the shapes to touch one another.

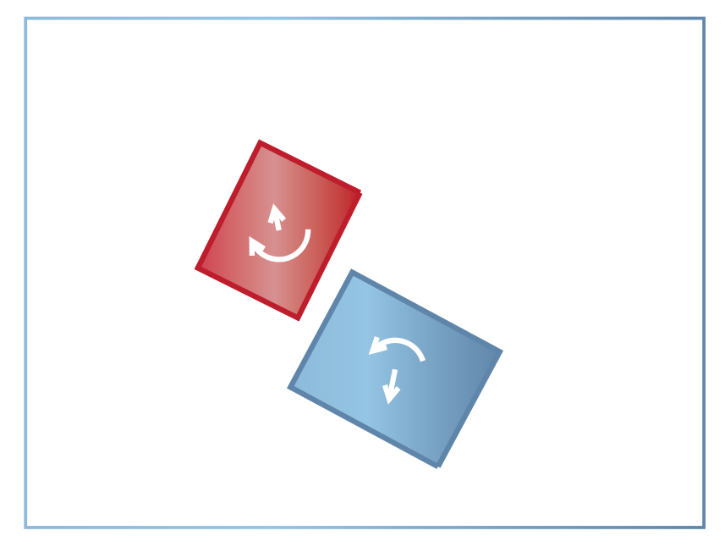

3
We have moved forward several frames and the bounding boxes are now overlapping. This signals that a more comprehensive test will have to be performed to check whether the objects themselves are colliding.


4
We confirm whether they are colliding or not by finding a separating axis: a line onto which we project the shapes. If these projections are not overlapping, the shapes are separated.


5
In the next frame, we find that after moving the objects to their new positions we can no longer find a separating axis: the objects are interpenetrating.


6
The first thing we do is to update the positions of the objects so that the penetration is removed.


7
Now we adjust the velocity and rotation speed of the objects (realistically - note the new rotations and velocities) so that they will be moving apart after the collision.


8
The objects now separate and continue in their new directions.